Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

А. В. Титков, С. А. Черепанов

СОЗДАНИЕ ВЕБ-ПРИЛОЖЕНИЙ

Учебное пособие

Томск
«Эль Контент»
2014

УДК 004.42:004.738.52(075.8)
ББК 32.973.202я73
Т 452

Рецензенты:
Ефимов А. А., канд. техн. наук, доцент кафедры
автоматизации обработки информации ТУСУРа;
Соловьев Д. А., канд. техн. наук, директор Nikolas group.

Титков А. В.

Т 452 Создание веб-приложений : учебное пособие / А. В. Титков, С. А. Черепанов. — Томск : Эль Контент, 2014. — 72 с.

ISBN 978-5-4332-0182-8

В учебном пособии рассматриваются процессы создания динамического клиент-серверного приложения для работы через сеть Интернет. В довольно краткой, но содержательной форме описываются такие аспекты разработки, как работа клиент-серверных программ, написание скриптов на языке Python, создание моделей, представлений, шаблонов и использование системы авторизации и регистрации для фреймворка Django, написание сценариев для различных анимаций на языке JavaScript, в том числе с применением библиотеки jQuery.

Данное пособие направлено на то, чтобы читатель после полного ознакомления с материалом был способен самостоятельно создавать веб-приложения невысокой степени сложности, при этом максимально придерживаясь принятых правил и стандартов в процессе разработки.

УДК 004.42:004.738.52(075.8)
ББК 32.973.202я73

ISBN 978-5-4332-0182-8

© Титков А. В.,
Черепанов С. А., 2014
© Оформление.
ООО «Эль Контент», 2014

ОГЛАВЛЕНИЕ

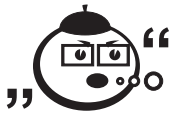
ВВЕДЕНИЕ

Сейчас, в век развития не только информационных технологий, но и сетевого взаимодействия между многочисленными программами и их пользователями, Интернет предоставляет замечательную возможность для разработчиков использовать единую, относительно стандартизированную среду для своих программ. Браузеры, которые есть на компьютере практически каждого пользователя, позволяют сразу, без установки какого-либо стороннего программного обеспечения, открывать нужные пользователю веб-страницы, пользоваться услугами различных сервисов и получать доступ практически к любой информации. В итоге многие и крупные, и мелкие сервисы предлагают свои услуги именно в качестве веб-приложений. Самыми популярными примерами последних лет являются успешные запуски таких компаний, как Фейсбук, Гугл и отечественного Вконтакте, которые позволяют пользователям получать доступ к своим страницам практически с любого устройства, которое имеет подключение к сети Интернет.

Практически всегда при создании веб-сервисов используются те или иные фреймворки или библиотеки, которые позволяют значительно ускорить разработку и избавить от шаблонного кода. Два примера таких решений будут продемонстрированы в пособии. Первым будет описан фреймворк Django для написания веб-приложений с помощью языка программирования Python, затем будет приведено краткое изложение принципов работы библиотеки jQuery, которая написана на языке JavaScript. Также будут показаны основы клиент-серверных приложений, базовые возможности языков программирования Python и JavaScript.

Соглашения, принятые в книге

Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.



Эта пиктограмма означает цитату.

Контрольные вопросы по главе

Глава 1

КЛИЕНТ-СЕРВЕРНЫЕ ПРИЛОЖЕНИЯ И ОСНОВЫ ЯЗЫКА PYTHON

1.1 Основы работы клиент-серверных приложений

Любое веб-приложение работает на клиент-серверной архитектуре — это значит, что у каждой программы (иными словами, у каждого сайта) есть две части — клиентская и серверная. Первая работает на компьютере непосредственно пользователя и реагирует на все пользовательские действия, вторая ждет от клиента запросов, по поступлению которых начинает формировать нужный ответ и отсылать его обратно клиенту. Обычно клиент нужен для обеспечения красивого интерфейса, который был бы удобен для конечного пользователя, а сервер для проведения основных вычислений и хранения информации. Чаще всего на множество клиентских приложений приходится всего несколько серверов (продемонстрировано на рис. 1.1), что значительно облегчает обслуживание, ведь большинство ошибок будет, скорее всего, лишь на нескольких машинах, непосредственно принадлежащих производителю.

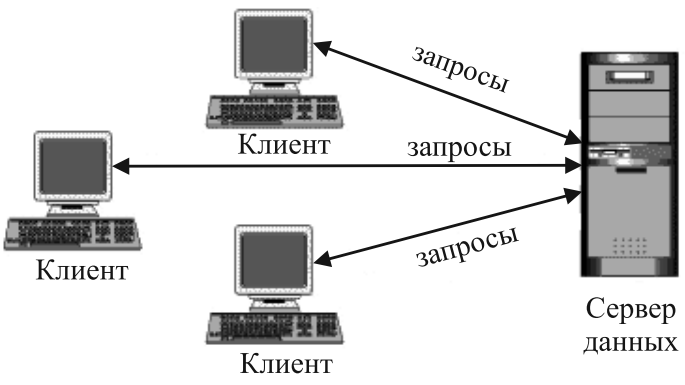


Рис. 1.1 – Общая схема работы клиент-серверного приложения

У описанной архитектуры есть как преимущества, так и недостатки, которые описаны ниже.

Преимущества:

- Отсутствие дублирования кода программы-сервера программами-клиентами.
- Сделать приложения кроссплатформенным при такой архитектуре гораздо легче, ведь нужно оптимизировать лишь клиентский код.
- Так как все вычисления выполняются на сервере, то требования к компьютерам, на которых установлен клиент, снижаются.
- Все данные хранятся на сервере, который, как правило, защищен гораздо лучше большинства клиентов. На сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа.

Недостатки:

- Неработоспособность сервера может сделать неработоспособной всю вычислительную сеть.
- Поддержка работы данной системы часто требует отдельного специалиста — системного администратора.
- Высокая стоимость оборудования [1].

Веб-сайты являются частным случаем такой архитектуры, когда общение между клиентом и сервером происходит через сеть Интернет по протоколу HTTP. Роль клиентской части выполняет браузер, который реализует пользовательский интерфейс, формирует запросы к серверу и обрабатывает ответы от него.

Серверная часть получает запрос от клиента, выполняет вычисления, после этого формирует веб-страницу и отправляет её клиенту по сети с использованием протокола HTTP.

Ярким примером веб-приложения является система управления содержимым статей Википедии: множество её участников могут принимать участие в создании сетевой энциклопедии, используя для этого браузеры своих операционных систем (будь то Microsoft Windows, GNU/Linux или любая другая операционная система) и не загружая дополнительных исполняемых модулей для работы с базой данных статей.

В настоящее время набирает популярность новый подход к разработке веб-приложений, называемый Ajax. При использовании Ajax страницы веб-приложения не перезагружаются целиком, а лишь догружают необходимые данные с сервера, что делает их более интерактивными и производительными.

Для создания веб-приложений на стороне сервера используются разнообразные технологии и многие языки программирования, такие, как PHP, C++, ASP.NET, Perl, Ruby, Python или JavaScript (с помощью платформы NodeJS).

Клиентская часть обычно создается с помощью языка разметки HTML для определения того, *что должно быть* на странице, каскадных таблиц стилей для того, чтобы описать, *как контент должен выглядеть*, и языка сценариев JavaScript для того, чтобы *создавать динамические изменения* контента и стилей.

Как было описано, веб-приложения активно используют протокол HTTP для передачи данных. Говоря простыми словами, HTTP представляет из себя некий

стандарт, которому должна соответствовать любая передаваемая информация. Сама структура каждого HTTP-сообщения довольно проста:

1. Стартовая строка (Starting line) — определяет тип сообщения.
2. Заголовки (Headers) — характеризуют тело сообщения, параметры передачи и прочие сведения.
3. Тело сообщения (Message Body) — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой.

Примером стартовой строки запроса является следующее:

```
GET /article/id35 HTTP/1.0
```

Здесь описывается метод, которым шлется запрос (всего есть несколько видов методов, но вам прежде всего нужно знать два: GET и POST). Метод GET чаще всего используется, когда клиенту просто нужна какая-либо информация с сервера, например просто получить веб-страницу для просмотра, а метод POST нужен, если клиент не просто получает какую-либо информацию, но и передает для обработки и сохранения, например когда заполняет форму с контактными данными.

После метода через пробел идет интересующий пользователя адрес, так называемый *URI*, *Uniform Resource Identifier*, в данном случае скорее всего сервер по такому адресу вернет страницу статьи с *id* равным 35. Исторически здесь передавался путь до нужного файла, который хранится на сервере, но сейчас технологии позволяют создавать абстрактные URI (какой приведен в данном примере), которые привязаны не к хранящимся на сервере файлам, а к некоторому программному коду, выполняющемуся по запросу. В данном случае будет выполнен исходный код, который разыскивает в базе данных статью с *id* равным 35, формирует HTML-документ, который отображает содержимое этой статьи, и возвращает HTTP-ответ с этим документом.

Третьим параметром указывается версия используемого протокола.

В заголовках обычно указываются различные характеристики и параметры сообщения, например тип передаваемых данных, размер сообщения и тому подобное.

В теле же содержатся непосредственно данные сообщения, например если была запрошена веб-страница, то в теле будет находиться HTML-код этой страницы.

HTML по своей сути представляет из себя просто определенный формат данных. Для того, чтобы явно себе представить, что это такое, поставьте перед собой задачу передать только в текстовом формате документ, который содержит не только текст, но и заголовки различных уровней (главы, подглавы, разделы подглав и т. д.), таблицы, списки и подобные стандартные для более-менее сложной документации элементы. Скорее всего, вы стали бы придумывать некоторый стандарт, который бы с помощью ключевых символов говорил, что этот текст — заголовок второго уровня (подглава), а тот текст — это список с пятью элементами. Собственно говоря, таким образом вы придумали бы аналог HTML! Чтобы понять на примере, как такую задачу решает именно данный язык разметки, внимательно изучите следующий HTML-код:

```
<h1>Глава 2. Права и свободы человека и гражданина</h1>
<h2>Статья 17</h2>
<ol>
```

```
        <li>В Российской Федерации признаются и гарантируются
права и свободы человека и гражданина согласно общепризнанным
принципам и нормам международного права и в соответствии
с настоящей Конституцией.</li>
          <li>Основные права и свободы человека неотчуждаемы
и принадлежат каждому от рождения.</li>
        <li>Осуществление прав и свобод человека и гражданина
не должно нарушать права и свободы других лиц.</li>
    </ol>
    <h2>Статья 18</h2>
    <p>Права и свободы человека и гражданина являются
непосредственно действующими. Они определяют смысл, содержание
и применение законов, деятельность законодательной
и исполнительной власти, местного самоуправления
и обеспечиваются правосудием.</p>
```

А теперь прочтите описание основных тегов в языке *HTML*:

1) *h1* — заголовок первого уровня;
2) *h2* — заголовок второго уровня;
3) *ol* — нумерованный список, содержащий в себе несколько элементов;
4) *li* — один элемент нумерованного или маркированного списка;
5) *p* — абзац (от англ. *paragraph*).

Заметьте, что все вышеупомянутые теги оборачивают свои названия в угловые скобки «<» и «>», а также имеют не только открывающие теги, но и закрывающие, которые отличаются тем, что после первой угловой скобки содержат символ косой черты «/» (например, в отличие от открывающего тега <*h*1>, закрывающий будет выглядеть следующим образом: </*h*1>).

Основы языка программирования Python

Python — язык со строгой динамической типизацией. Что это означает?

Есть языки со строгой типизацией (Pascal, Java, C и т. п.), у которых тип переменной определяется заранее и не может быть изменен, и есть языки с динамической типизацией (Python, Ruby, VB), в которых тип переменной трактуется в зависимости от присвоенного значения.

Языки с динамической типизацией можно разделить еще на 2 вида. Строгие, которые не допускают неявного преобразования типа (Python), и нестрогие, которые выполняют неявные преобразования типа (например VB, в котором можно легко сложить строку '123' и число 456).

Сферы применения языка Python:

- Компания Google использует Python в своей поисковой системе, и создатель Python, Гвидо ван Россум, работал в этой корпорации до декабря 2012 года. Сейчас ван Россум трудится в Dropbox Inc.
- Такие компании, как Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm и IBM, используют Python для тестирования аппаратного обеспечения.

- Служба коллективного использования видеоматериалов YouTube в значительной степени реализована на Python.
- NSA использует Python для шифрования и анализа разведданных.
- Компании JPMorgan Chase, UBS, Getco и Citadel применяют Python для прогнозирования финансового рынка.
- Популярная программа BitTorrent для обмена файлами в пиринговых сетях написана на языке Python.
- Популярный веб-фреймворк App Engine от компании Google использует Python в качестве прикладного языка программирования.
- NASA, Los Alamos, JPL и Fermilab используют Python для научных вычислений [2].

Создание переменных в языке осуществляется простым присваиванием в любой части программы.

Синтаксис операции присваивания:

```
<variable> = <value>,
```

где `<variable>` — имя переменной, а `<value>` — её значение.

Комментарии в Python создаются символом решетки «#», после которой до конца строки следует текст комментария.

Пример комментария:

```
# я комментарий, я могу начинаться с новой строки
x = 3.0 # и могу продолжить строку с кодом
```

Для возможности использования русского языка в *начале скрипта* обязательно нужно добавить комментарий с указанием кодировки.

Использование символов Unicode в программе на Python:

```
str = \переменная с русскими символами/
# так работать нельзя, нет указания кодировки!

#coding:utf-8
str = \переменная с русскими символами/
# а вот так можно, стоит указание кодировки в начале
# ƒайла :)
```

Также следует учитывать случай, если внутри какой-либо строки содержатся нелатинские символы. В таком случае для корректной работы Python необходимо указывать, что текущая строка должна обрабатываться с учетом символов Юникода. Тут будет достаточным перед каждой строкой просто добавлять символ «*u*».

Пример создания строки *Unicode*:

```
str1 = u\строка с символами Юникода/
str2 = \latin symbols, unicode is not required/
```

Для *ввода* какого-либо *значения* с клавиатуры в Python до версии 3.0 желательно использовать функцию *raw_input()*. Эта функция является примерным аналогом *readln* из языка Pascal.

Внимание! Несмотря на существование функции *input()* схожего действия, использовать ее в программах не рекомендуется, так как интерпретатор пытается выполнить вводимые с ее помощью синтаксические выражения, что является серьезной уязвимостью в безопасности программы.

Для вывода значения на экран (аналог оператора *writeln* в языке Pascal) можно воспользоваться оператором *print*.

Пример работы *print()*:

```
x = 23
# теперь в переменной x хранится число 23
print(x)
# вывести на экран 23
x = 'foo'
# а теперь в переменной x хранится строка ''foo''
print(x)
# вывести строку ''foo''
del x
# теперь переменная x не определена в программе
print(x)
# и при выводе возникнет исключение
y = None
# присвоили переменной y значение ... None
print(y)
# вывести None на экран
```

Следующее, что необходимо знать, — как строятся *базовые алгоритмические единицы* — ветвления и циклы. Для начала, необходима небольшая справка. В Python нет специального ограничителя блоков кода, их роль выполняют *отступы*. То есть то, что написано с одинаковым отступом, — является *одним командным блоком*. Поначалу это может показаться странным, но после легкого привыкания нетрудно понять, что эта «вынужденная» мера позволяет получать куда более читабельный код.

Условие задается с помощью оператора *if*, который заканчивается «:». Альтернативные условия, которые будут выполняться, если первая проверка «не прошла», задаются оператором *elif*. Наконец, *else* задает ветку, которая будет выполнена, если ни одно из условий не подошло.

Пример работы оператора ветвления:

```
x = 15
if (x < 10) or (x > 20):
    print ''x is bad''
elif 10 < x < 20:
    print ''x is good!''
else:
    print ''x is wierd''
# x is good!
```

Циклы. Простейшим случаем цикла является цикл *while*. В качестве параметра он принимает условие и выполняется до тех пор, пока оно истинно.

Пример работы цикла *while*:

```
x = 0
while x <= 3:
    print x
    x += 1
# выведет:
# 0
# 1
# 2
# 3
```

Обратите внимание, что поскольку и *print x*, и *x += 1* написаны с одинаковым отступом, обе последние строки считаются телом цикла. Второй вид циклов в Python — цикл *for*.

Синтаксис цикла *for*:

```
for переменная in список:
     команды
```

Переменной будут присваиваться по очереди все значения из массива. Вот простой пример: в роли списка будет выступать строка, которая является не чем иным, как списком символов.

Пример работы цикла *for* для строки:

```
x = "Hey!"
for char in x:
     print char

# выведет:
# H
# e
# y
# !
```

Таким образом, можно разложить строку по символам.

Что же делать, если нужен цикл, повторяющийся определенное число раз? Очень просто, на помощь придет функция *range*. На входе она принимает от одного до трех параметров, на выходе возвращает список чисел, по которому мы можем «пройтись» оператором *for*. Вот несколько примеров использования функции *range*, которые объясняют роль ее параметров.

Примеры работы функции *range()*:

```
range(10)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(2, 12)
# [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
range(2, 12, 3)
# [2, 5, 8, 11]
range(12, 2, -2)
# [12, 10, 8, 6, 4]
```

И маленький пример с циклом, результат которого эквивалентен примеру из описания цикла while.

Совместное использование цикла *for* и функции *range()*

```
for x in range(4):
    print x

# выведет:
# 0
# 1
# 2
# 3 [3].
```

Внутри любого цикла возможно использование операторов *continue* и *break*. Оператор *continue* прекращает выполнение *текущей* итерации и начинает новую (то есть цикл не прекращается), а оператор *break* полностью завершает работу цикла.

Примеры использования операторов *continue* и *break*:

```
i = 0
while i < 10:
     i += 1
     if i == 2:
       print "пропустить 2"
       continue
     if i == 4:
       print "i равно 4. Пора закончить работу"
       break
     print i

# Выведет:
# 1
# пропустить 2
# 3
# i равно 4. Пора закончить работу
```

Кортежи. Кортежи (англ. *tuple*) используются для представления неизменяемой последовательности разнородных объектов. Они обычно записываются в круглых скобках, но если неоднозначности не возникает, то скобки можно опустить.

Примеры работы с кортежами:

```
t = (2, 2.05, "Hello")
print t
# (2, 2.0499999999999998, 'Hello')
(a, b, c) = t
print a, b, c # здесь скобки были опущены
# 2 2.05 Hello
z, y, x = t # опять скобки опущены
print z, y, x
# 2 2.05 Hello
a = 1
```

```
b = 2
a,b = b,a
print a,b
# 2 1
x = 12,
x
# (12,)
```

Как видно из примера, кортеж может быть использован и в левой части оператора присваивания. Значения из кортежа в левой части оператора присваивания связываются с аналогичными элементами правой части. Этот факт как раз и дает такие замечательные возможности, как массовая инициализация переменных и возврат множества значений из функции одновременно. Последний пример демонстрирует создание кортежа из одного элемента (его часто называют синглтоном).

Списки. В Python отсутствуют массивы в традиционном понимании этого термина. Вместо них для хранения однородных (и не только) объектов используются списки. Они задаются тремя способами.

Простое перечисление.

Примеры работы со списками:

```
a = [2, 2.25, "Python"]
print a
# [2, 2.25, 'Python']

# Преобразовать строку в список
list = list("help")
print list
# ['h', 'e', 'l', 'p']
```

Создать список можно с помощью генератора, который позволяет инициировать значения в одну строку кода. В данном случае создастся список, содержащий кубы всех нечетных чисел от 0 до 19.

Пример генератора списков:

```
list = [x ** 3 for x in range(20) if x%2==1]
print list
# [1, 27, 125, 343, 729, 1331, 2197, 3375, 4913, 6859]
```

Для работы со списками определен ряд операторов и функций (в каждом примере переменная *list* является представителем списка):

- *len(list)* — Длина последовательности *list*.
- *x in s* — Проверка принадлежности элемента последовательности. В новых версиях Python можно проверять принадлежность подстроки строке. Возвращает True или False.

Пример работы оператора вхождения *in*:

```
x not in s == not x in s
```

- *list1 + list2* — Конкатенация последовательностей.
- *s*n* или *n*s* — Последовательность из *n* раз повторенной *list*. Если $n < 0$, возвращается пустая последовательность.

- *list[i]* — Возвращает *i*-й элемент *list* или *len*(*list*) + *i*-й, если $i < 0$.
- *list[i:j:d]* — Срез из последовательности *list* от *i* до *j* с шагом *d* будет рассматриваться ниже.
- *min(list)* — Наименьший элемент *list*.
- *max(list)* — Наибольший элемент *list*.
- *list[i] = x* — *I*-й элемент списка *list* заменяется на *x*.
- *list[i:j:d] = t* — Срез от *i* до*j* (с шагом *d*) заменяется на (список) *t*.
- *del list[i:j:d]* — Удаление элементов среза из последовательности.

Кроме того, для списков определен ряд методов:

- *list.append(x)* — Добавляет элемент в конец последовательности.
- *list.count(x)* — Считает количество элементов, равных *x*.
- *list.extend(s)* — Добавляет к концу последовательности последовательность *list*.
- *list.index(x)* — Возвращает наименьшее *list*, такое, что *list[i] == x*. Возбуждает исключение ValueError, если *x* не найден в *list*.
- *list.insert(i, x)* — Вставляет элемент *x* в *i*-й промежуток.
- *list.pop(i)* — Возвращает *i*-й элемент, удаляя его из последовательности.
- *list.reverse()* — Меняет порядок элементов *list* на обратный.
- *list.sort([cmpfunc])* — Сортирует элементы *list*. Может быть указана своя функция сравнения *cmpfunc*.

Для преобразования кортежа в список есть функция *list*, для обратной операции — *tuple*.

Об индексировании списков и выделении подпоследовательностей следует еще раз упомянуть отдельно (этот механизм работает аналогично и для строк). Для получения элемента используются квадратные скобки, в которых находится индекс элемента. Элементы нумеруются с нуля. Отрицательное значение индекса указывает на элементы с конца. Первый с конца списка (строки) элемент имеет индекс –1.

Примеры индексации списков:

```
s = [0, 1, 2, 3, 4]
print s[0], s[-1], s[3]
# 0 4 3
s [2] = -2
print s
# [0, 1, -2, 3, 4]
del s [2]
print s
# [0, 1, 3, 4]
```

Сложнее обстоят дела со срезами. Для получения срезов последовательности в Python принято указывать не номера элементов, а номера «промежутков» между ними. Перед первым элементом последовательности промежуток имеет индекс 0,

перед вторым −1 и так далее. Отрицательные значения отсчитывают элементы с конца строки.

В общем виде срез записывается в следующем виде:

```
список[начало: конец: шаг]
```

По умолчанию начало среза равно 0, конец среза равен *len* (список), шаг равен 1. Если шаг не указывается, второй символ «:» можно опустить.

С помощью среза можно указать подмножество для вставки списка в другой список, даже при нулевой длине. Это удобно для вставки списка в строго определенной позиции.

Пример изменения списка с помощью срезов:

```
l = range(12)
print l
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
print l[1:3]
# [1, 2]
print l[-1:]
# [11]
print l[::2]
# [0, 2, 4, 6, 8, 10]
l[0:1]=[-1,-1,-1]
print l
# [-1, -1, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
del l[:3]
print l
# [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Словари. Словарь (хэш, предопределенный массив) — изменяемая структура данных, предназначенная для хранения элементов вида «ключ:значение». Главное отличие от списка или кортежа — это то, что позиционируются элементы не по числовым индексам, а по ключевым словам. Все легко показывается на примере.

Пример работы со словарями. *Создать словари*.

```
h1 = {1:"one", 2:"two", 3:"three"}
h2 = {0:"zero", 5:"five"}
h3 = {"z":1, "y":2, "x":3}

# Цикл по паре ключ-значение
for key, value in h1.items():
    print key, " ", value
# 1 one
# 2 two
# 3 three

#Цикл по ключам
for key in h2.keys():
    print key, " ", h2[key]
# 0 zero
# 5 five
```

```
#Цикл по значениям
for v in h3.values():
    print v
# 2
# 3
# 1

#Добавление элементов из другого словаря
h1.update(h3)

#Количество пар в словаре
print len(h1)
#6 [4].
```

Функции. Язык Python предоставляет возможность оформления блоков программного кода в виде функций, параметризуемых аргументами, которые им передаются. Ниже приводится общий синтаксис определения функции.

Пример создания функций:

```
def functionName(argument1, argument2):
    # нельзя забывать добавление двоеточия
    # аргументы к функции перечислять через запятую
    print argument1, argument2
    return True
    # с помощью оператора return можно определить то,
    # должна возвращать что функция
```

Часть *argument1, argument2* не является обязательной; для передачи нескольких аргументов их необходимо отделить друг от друга запятыми. Любая функция в языке Python возвращает некоторое значение — по умолчанию это значение *None*; если явно не возвращается значение с помощью инструкции *return value*, где *value* — это возвращаемое значение. Возвращаемое значение может быть единственным значением или кортежем возвращаемых значений. Возвращаемое значение может игнорироваться вызывающей программой, в этом случае оно просто уничтожается.

Примечательно, что инструкция *def* действует как оператор присваивания. При выполнении инструкции *def* создаются новый объект функция и ссылка на объект с указанным именем, которая указывает на объект функцию. Поскольку функции — это объекты, они могут сохраняться в виде элементов коллекций и передаваться в качестве аргументов другим функциям, в чем вы сможете убедиться в последующих главах.

Пример функции, которая принимает от пользователя строку и пытается преобразовать её в число, если преобразование проходит с ошибкой, то пользователя вновь просят ввести строку с числом.

Функция, преобразующая полученную от пользователя строку в целое число:

```
def get_int(msg):
  while True:
    try:
      i = int(input(msg))
```

```
        return i
    except ValueError as err:
        print(err)
```

Ниже показан пример использования этой функции.

Пример использования функции, которая преобразует введенное пользователем значение в число:

```
age = get_int("Введите свой возраст: ")
```
[5].

Контрольные вопросы по главе 1

1. Какие преимущества у клиент-серверной архитектуры приложений?
2. Что такое HTTP и для каких целей он в основном используется?
3. Что такое методы запросов? Приведите примеры двух методов запроса и укажите их различия.
4. Как в формате HTML указать, что текущий текст — это заголовок второго уровня?
5. Как создавать строки и списки в языке программирования Python?
6. Как в языке программирования Python определяется тело циклов, условий, функций?
7. Как на языке программирования Python создать список из натуральных чисел от 1 до 100?
8. Как на языке программирования Python узнать длину строки?

Глава 2

ОПИСАНИЕ DJANGO И ИСПОЛЬЗОВАНИЕ КОМАНД ИЗ DJANGO-ADMIN.PY

2.1 Основное описание веб-фреймворка Django

Django — свободный фреймворк для веб-приложений на языке Python, использующий шаблон проектирования MVC. Django является фреймворком, потому что именно с помощью него формируется каркас будущего приложения. Создатели Django уже продумали архитектуру будущего веб-приложения, которой нужно следовать, программируя с помощью фреймворка. Иными словами, Django — это довольно многофункциональный каркас, который предоставит почти все нужные возможности для всех самых распространенных в вебе задач.

Отличие фреймворков от библиотек состоит в следующем принципе: при работе с первым программист свое приложение встраивает в этот фреймворк, а при работе со второй, программист функции библиотеки встраивает в свое приложение.

Веб-фреймворк Django используется в таких крупных и известных сайтах, как Instagram, Disqus, Mozilla, The Washington Times, New York Times, National Geographic, Pinterest и др.

В архитектуре всех проектов на Django должно прослеживаться четкое деление проекта на несколько как можно более независимых *приложений*, каждое из которых выполняет только свою узконаправленную задачу. Такой подход позволяет значительно упростить структуру проекта и вследствие этого избежать множества архитектурных ошибок. Также довольно характерной чертой Django является то, что каждое составное приложение проекта создается с помощью шаблона проектирования MVC, которое было немного изменено в связи со спецификой веб-приложений.

В общих чертах для проекта, у которого три приложения (например, одно приложение для авторизации/регистрации пользователей, второе для создания/редактирования/удаления/чтения статей в блоге, третье для рекламы на сайте), схема работы представлена ниже на рисунке 2.1. Заметьте, что, в идеале, приложения не

должны пересекаться с собой слишком часто, а в каждом приложении реализована своя MVC-структура, где контроллер влияет и на модель, и на представление, и ещё вдобавок представление влияет тоже на модель (что видно из направления стрелок).

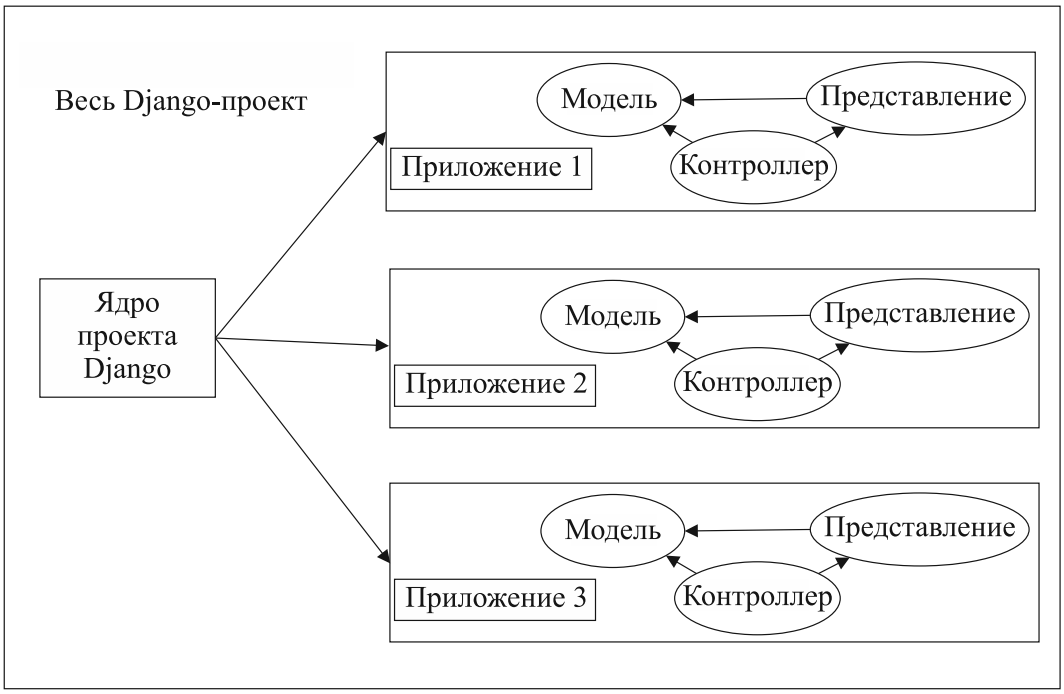
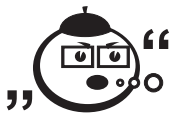


Рис. 2.1 – Структура базового Django-проекта с тремя приложениями

Говоря простыми словами, MVC — это такой способ разработки программного обеспечения, при котором код определения и доступа к *данным* (модель) отделен от *логики взаимодействия* с приложением (контроллер), которая, в свою очередь, отделена от *пользовательского интерфейса* (представление).

Главное достоинство такого подхода состоит в том, что компоненты слабо связаны. У каждого компонента веб-приложения, созданного на базе Django, имеется единственное назначение, поэтому его можно изменять независимо от остальных компонентов. Например, разработчик может изменить URL некоторой части приложения, и это никак не скажется на ее реализации. Дизайнер может изменить HTML страницы, не трогая генерирующий ее код на языке Python. Администратор базы данных может переименовать таблицу базы данных и описать это изменение в одном-единственном месте, а не заниматься контекстным поиском и заменой в десятках файлов.

Как пишут сами разработчики платформы, их приложение появилось примерно из-за следующей схемы работы в их компании:

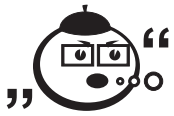


«Если у вас есть достаточно продолжительный опыт создания веб-приложений, то вы наверняка знакомы с проблемами, присущими рассмотренному выше примеру CGI-сценария. Классический веб-разработчик проходит такой путь:

1. *Пишет веб-приложение с нуля.*
2. *Пишет еще одно веб-приложение с нуля.*
3. *Осознает, что первое веб-приложение имеет много общего со вторым.*
4. *Перерабатывает код так, чтобы некоторые вещи из первого приложения можно было использовать повторно во втором.*
5. *Повторяет шаги 2–4 несколько раз.*
6. *Понимает, что он придумал фреймворк.*

Именно так и был создан Django!» [6].

В ходе дальнейшего развития платформы применение паттерна MVC было немного изменено, и эти изменения от стандартного подхода нужно тонко чувствовать. Суть модели остается той же самой: на уровне модели в Django реализуется только запись и получение данных из базы. Однако уровень «представления» в Django не является последним этапом в отображении данных — представления в платформе Django по своей сути ближе к «контроллерам» в архитектуре MVC. Они являются функциями на языке Python, которые связывают между собой уровень модели и уровень отображения (состоящий из разметки HTML и языка шаблонов платформы Django).



Процитируем высказывание членов команды разработчиков Django: «В нашей интерпретации MVC «представление» описывает данные, отображаемые перед пользователем. Важно не то, как данные должны выглядеть, а то, какие данные должны быть представлены. Другими словами, представление описывает, какие данные вы увидите, а не то, как они будут выглядеть. Это довольно тонкое отличие.

Проще говоря, платформа Django разбивает уровень представления на два — функцию представления, определяющую, какие данные из модели будут отображаться, и шаблон, определяющий окончательное представление информации. Что касается контроллера, то его функции выполняются самой платформой — она обладает механизмами, определяющими, какое представление и шаблон должны использоваться в ответ на конкретный запрос» [7].

В итоге полученная модель получила название *MTV* — что значит *Model-Template-View*.

Модели. Основой любого приложения, будь то веб-приложение или любое другое, является информация, которую это приложение собирает, модифицирует и отображает. С точки зрения многоуровневой архитектуры приложения модель является самым нижним уровнем, или фундаментом. Представления и шаблоны могут добавляться и удаляться, изменять порядок ввода/вывода данных и их представления, но модель остается практически неизменной.

С точки зрения проектирования многоуровневого веб-приложения модель является, пожалуй, самой простой для понимания и самой сложной в реализации. Моделирование задач реального мира в объектно-ориентированной системе часто является относительно простой задачей, но с точки зрения веб-сайтов, работающих с высокой нагрузкой, самая реалистичная модель не всегда является самой эффективной.

Модель включает широкий диапазон потенциальных ловушек, одна из которых связана с изменением программного кода модели уже после развертывания приложения. Несмотря на то, что «изменяется всего лишь программный код модели», тем не менее в действительности изменяется структура базы данных, а это часто отражается на данных, уже хранящихся в базе. В последующих главах мы рассмотрим множество подобных проблем, когда будем исследовать архитектуру некоторых примеров приложений.

Представления. Представления в значительной степени (иногда полностью) формируют логику приложений на платформе Django. Их определение выглядит обманчиво просто: функции на языке Python, связанные с одним или более адресами URL и возвращающие объекты ответов HTTP. Все, что будет происходить между этими двумя точками применения механизмов платформы Django, реализующих работу с протоколом HTTP, целиком и полностью зависит от вас. На практике на этом этапе обычно решается несколько похожих задач, таких как отображение объекта или списка объектов, полученных от модели, или добавление новых объектов, а также проверка аутентификации пользователя приложения и либо предоставление, либо отклонение попытки доступа к данным.

Платформа Django предоставляет множество вспомогательных функций для решения подобных задач, но вы можете реализовать всю логику работы самостоятельно, чтобы иметь полный контроль над процессом, широко использовать вспомогательные функции для быстрого создания прототипа и разработки самого приложения или комбинировать эти два подхода.

Шаблоны. Вы должны были заметить, что мы только что заявили, что представление отвечает за отображение объектов, полученных из модели. Это верно не на 100 процентов. Если подразумевать, что методы просто возвращают ответ HTTP, это достаточно верно — можно было бы реализовать на языке Python вывод строки и возвращать ее, и это было бы ничуть не хуже. Однако в подавляющем большинстве случаев такая реализация крайне неэффективна и, как уже упоминалось ранее, очень важно придерживаться деления на уровни.

Вместо этого для отображения результатов в разметку HTML, которая часто является конечным результатом работы веб-приложения, большинство разработчиков приложений на платформе Django используют язык шаблонов. Шаблоны,

по сути, являются текстовыми документами в формате HTML, со специальным форматированием там, куда выводятся значения, получаемые динамически, поддерживающие возможность использовать простые логические конструкции, такие как циклы и другие. Когда от представления требуется вернуть документ HTML, оно обычно указывает шаблон, передает в него информацию для отображения и использует отображенный шаблон в своем ответе.

Хотя HTML является наиболее типичным форматом, в действительности шаблоны не обязательно должны содержать какую либо разметку HTML — шаблоны могут использоваться для воспроизведения любого текстового формата, такого как CSV (comma separated values — значения, разделенные запятыми), или даже текста сообщения электронной почты. Самое важное состоит в том, что шаблоны позволяют отделить отображение данных от программного кода представления, которое определяет, какие данные следует представить.

К настоящему моменту мы рассмотрели некоторые из наиболее крупных архитектурных компонентов, составляющих саму систему Django, а также вспомогательные компоненты, не входящие в ее границы.

Давайте теперь сложим их вместе, чтобы получить общее представление. На рисунке 2.2 можно видеть, что ближе всего к пользователю располагается протокол HTTP. С помощью адресов URL пользователи могут отправлять запросы веб-приложениям на платформе Django и принимать ответы посредством своих веб-клиентов, которые могут также выполнять программный код JavaScript и использовать технологию Ajax для взаимодействия с сервером.

На другом конце спектра (внизу рисунка) можно видеть базу данных — хранилище информации, которое управляется с помощью моделей и механизма ORM платформы Django, взаимодействующих с базой данных посредством DB API языка Python и клиентских библиотек базы данных. Эти библиотеки используются в качестве промежуточного звена, они обычно написаны на языке C/C++ и предоставляют интерфейс для языка Python.

Наконец, в середине располагается основа приложения на платформе Django. Парадигма MVC на языке Django превращается в парадигму MTV (Model Template View — модель, шаблон, представление). Представления играют роль контроллера между моделью данных, создающей, изменяющей и удаляющей данные в базе посредством механизма ORM, и окончательным представлением данных пользователю с помощью шаблонов.

Соединив все вместе, получаем следующую картину: поступающие запросы HTTP передаются веб-сервером платформе Django, которая принимает их на промежуточном уровне обработки запросов. После этого, исходя из шаблонов адресов URL запросы передаются соответствующему представлению, которое выполняет основную часть работы, привлекая к работе при этом модель и/или шаблоны, необходимые для создания ответа. Затем ответ проходит один или более промежуточных уровней, где выполняется окончательная обработка перед передачей ответа HTTP обратно веб-серверу, который в свою очередь отправляет ответ пользователю [7].

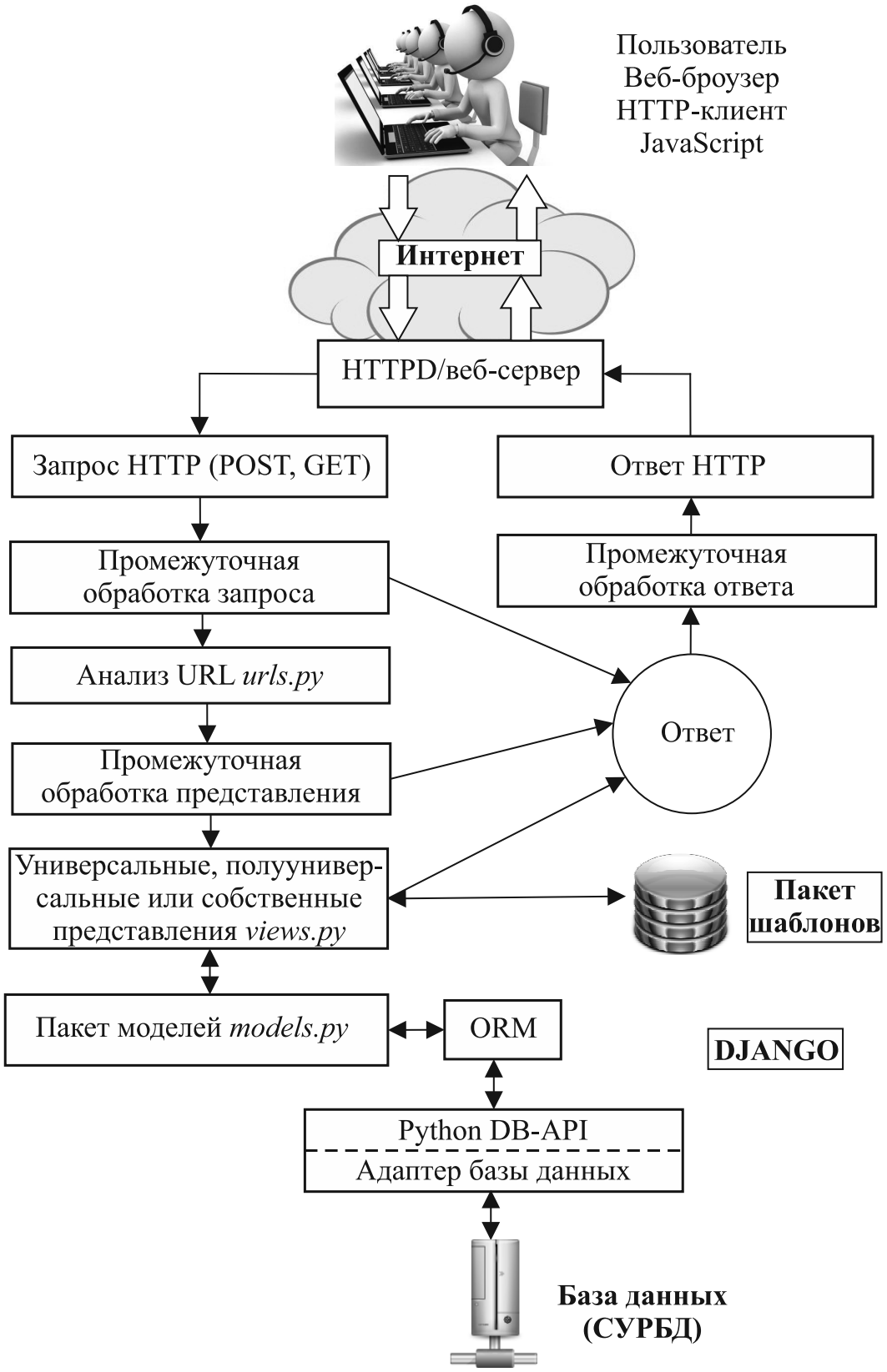


Рис. 2.2 – Общая схема работы приложения на платформе Django

2.2 Описание команд в django-admin.py

При установке *Django* в папку *C:/python27/Scripts* установится дополнительный файл *django-admin.py*. С помощью этого файла вы можете через командную строку запускать различные команды по созданию и конфигурированию Django-проектов. Конечно же, из-за того, что скрипт написан на языке Python, вам придется запускать команды через оболочку Python.

Пример команды, создающей новый проект:

```
django-admin.py startproject new_poject
```

Команда создаст в директории, где выполняется команда, папку *new_poject*, а в ней некоторые базовые файлы для работы с проектом, имена которых приведены ниже.

Структура новосозданного проекта:

```
new_poject/
    manage.py
    new_poject/
       __init__.py
      settings.py
      urls.py
      wsgi.py
```

Эти файлы (*manage.py, settings.py, urls.py и wsgi.py*) и представляют из себя *ядро проекта*, которое обозначено левым прямоугольником в базовой схеме выше.

Однако большая часть задач, которую вам нужно будет решать с помощью команды *django-admin.py*, будет связана с конкретным проектом (предыдущий же пример, в отличие от этого, как раз не имел никакого отношения к уже существующим проектам на Django, в том примере проект только создавался). Например, с помощью этого скрипта можно создать базу данных со структурой, описанной вами в моделях на языке Python. Или можно начать обработку всех статичных файлов проекта, собрав их в одну папку. Задач, которые нужно решать для какого-то конкретного проекта, может быть великое множество, и чтобы указать скрипту *django-admin.py*, что введенную вами команду нужно применить только для данного проекта, создается файл *manage.py*, который есть в описании структуры нового проекта абзацем выше. Чтобы выполнить специфичную для проекта команду, надо в командной строке перейти в папку самого проекта (в ту же папку, где хранится файл *manage.py*) и набрать примерно следующее:

```
python manage.py syncdb
```

Это значит, что с помощью интерпретатора *Python* выполнить скрипт *manage.py*, передав в качестве параметра строку *syncdb*. Естественно, вместо *syncdb* можно передавать и многие другие аргументы, узнать о всех вариантах вы можете, выполнив команду, которая выводит на экран список всех доступных для проекта команд:

```
python manage.py help
```

Как уже говорилось выше, каждый проект в Django — это набор нескольких приложений, каждое из которых выполняет свою узконаправленную задачу. Чтобы добавить в Django-проект новое приложение, достаточно выполнить команду добавления нового приложения в проект:

```
python manage.py startapp app_name,
```

где вместо `app_name` нужно указать имя будущего приложения. Данная команда создаст внутри папки проекта новую директорию `app_name` и несколько файлов внутри неё.

Файловая структура нового приложения:

```
new_app/
 __init__.py
models.py
tests.py
views.py
```

Как вы уже знаете, весь код, который определяет данные и обработку данных, должен описываться в файле *models.py*. Например, для приложения авторизации/регистрации пользователей в файле *models.py* должна описываться структура баз данных для пользователей, методы создания нового пользователя, изменения состояния уже зарегистрированных и так далее. Файл *view.py* служит для того, чтобы при запросе на определенный адрес вызывать нужные методы *models.py* и полученные из моделей данные передавать в шаблоны, чтобы затем вернуть сформированный из шаблонов HTML-код пользователю.

Контрольные вопросы по главе 2

1. Чем проект отличается от приложения в терминологии фреймворка Django?
2. Какая файловая структура у ядра проекта, созданного на Django?
3. Что такое MVC и в чем его основное отличие от MTV?
4. Что отвечает за работу с данными в паттерне MVC?
5. Как Django выбирает, какое представление должно создавать ответ на пользовательский запрос?
6. Чем скрипт *manage.py* проекта отличается от скрипта *django-admin.py* платформы Django? Что между ними общего?
7. Как добавить новое приложение в существующий проект на Django?

Глава 3

ШАБЛОНЫ

3.1 Работа с шаблонами в Django

Как уже упоминалось ранее, шаблон в Django представляет собой строку текста, предназначенную для отделения представления документа от его данных. В шаблоне могут встречаться маркеры и простые логические конструкции (шаблонные теги), управляющие отображением документа. Обычно шаблоны применяются для создания HTML-разметки, но в Django они позволяют генерировать документы в любом текстовом формате [6].

В каком-то смысле шаблоны в Django — это некий аналог бланков из реального мира. Приходя в любое государственное учреждение, вы можете получить полупустую заготовку какого-либо документа, к примеру заявления, которое вы заполняете лишь отчасти, потому что общие части уже пропечатаны за вас. То есть шаблон — это основа будущей HTML-разметки, которая должна быть заполнена теми данными, что будут переданы в шаблон.

Эти данные, которые необходимо внести в шаблон, в сумме называются *контекстом*, а процесс, когда данные вносятся в шаблон, называется *рендерингом*.

Пример шаблона для *Django*:

```
from django.template import Template
template = Template(\\<h1>Привет, моё имя {{name}}!</h1>//)
```

и если вы пожелаете на место переменной *name* поставить, например, имя «Василий», то контекстом будет обычный словарь из языка Python, оформленный в специальный класс *Context* примерно такого содержания:

```
from django.template import Context
context = Context({\\ name// : u\\ Василий//})
```

Теперь осталось только совместить шаблон и контекст с помощью рендеринга. Рендеринг шаблонов в Django:

```
html_page = template.render(context)
# html_page == \\ <h1>Привет, моё имя Василий!</h1>//
```

Помимо прямого вывода переменной в будущий HTML-документ в шаблонах можно пользоваться некими аналогами конструкций из языка Python. Например, в шаблонах можно использовать циклы *for* и условия *if*.

Пример использования алгоритмических конструкций в шаблонах:

```
from django.template import Context, Template
template = Template(\\////
  <h1>Привет, моё имя {{ my_name }}!</h1>
  {% for friend in friend_list %}
    {% if friend.is_groupmate %}
      <p>Мой одногруппник: {{ friend.name }}</p>
    {% endif %}
  {% endfor %}
\\////)
context = Context({\\my_name//: u\\Василий//,
  \\friend_list//: [
    {\\name//: u\\Георгий//, \\is_groupmate//: False},
    {\\name//: u\\Леонид//, \\is_groupmate//: True},
    {\\name//: u\\Константин//, \\is_groupmate//: True},
    {\\name//: u\\Гавриил//, \\is_groupmate//: False}, ]  })
html_page = template.render(context)
# в итоге получится следующий html-код:
# <h1>Привет, моё имя Василий!</h1>
# <p>Мой одногруппник: Леонид</p>
# <p>Мой одногруппник: Константин</p>
```

В данном примере прямо внутри шаблона был произведен обход по всем элементам массива *friend_list* и для каждого из друзей была произведена проверка, является ли друг ещё и одногруппником, и если вся проверка успешно прошла, имя друга выводится в списке ниже.

Естественно, зашивать HTML-код прямо в представления на языке Python не очень хорошая штука (как и любое другое смешивание языков), поэтому хорошей, точнее сказать, обязательной, практикой является вынос шаблона в отдельный файл с форматом *.html*, и файл этот должен храниться в папке *templates* любого вашего приложения. Так что файловая структура ваших приложений в Django-проектах станет ещё чуточку сложнее. Теперь пример базового приложения стал шире.

Файловая структура приложения с шаблонами:

```
your_app/
     templates/
       # здесь будут храниться ваши шаблоны
      __init__.py
     models.py
     tests.py
     views.py
```

Также вам обязательно нужно знать, что создавать вручную объекты *Template*, затем объекты *Context*, а потом вызывать метод *render* первого, передавая туда контекст, необязательно. Естественно, как и все уважающие себя программисты, создатели Django создали автоматизированный вариант этого кода, который умещается всего... в одну строку кода, куда уж короче! Для этого была создана функция *render*, которая находится в модуле *django.shortcuts*. Пользоваться этой функцией очень просто.

Пример использования функции *render()*:

```
from django.shortcuts import render
def some_view(request):
   return render(request,
     \some_template.html/,
     {
       \context_key1/:\context_value1/,
       \context_key2/:\context_value2/,
     }
   )
```

Здесь функции *render* в качестве первого аргумента передается *объект запроса*, который должен быть в каждом представлении, вторым аргументом идет *имя шаблона*, который сохранен в одной из папок *templates* ваших приложений, а третий аргумент — не что иное, как словарь, содержащий *контекст вашего шаблона*.

Контрольные вопросы по главе 3

1. Зачем нужны шаблоны в фреймворке Django? Какие проблемы они решают?
2. Что такое контекст шаблона?
3. Как создать новый контекст?
4. Как контекст применить к уже существующему шаблону?
5. Где должны храниться файлы шаблонов?
6. Что такое рендеринг шаблонов?
7. С помощью какой функции можно запустить процесс рендеринга файла шаблона?

Глава 4

СТАТИЧНЫЕ ФАЙЛЫ В DJANGO-ПРОЕКТАХ И РАБОТА С CSS

4.1 Работа со статичными файлами в Django-проектах

Django-разработчики в основном работают с динамической частью приложения — представлениями и шаблонами, которые чаще всего изменяют свое содержимое при каждом запросе (например, страница профиля */profile/* будет у каждого пользователя разная, хотя каркас для всех будет общим). Но веб-приложения содержат и другую часть: статические файлы (изображения, CSS, Javascript и др.), которые не требуют никакой программной обработки. Для них нет потребности в рендеринге, они не зависят от содержимого базы данных. При каждом запросе к такому файлу веб-серверу достаточно просто вернуть их прямо такими, какими их сохранили в последний раз.

В больших проектах — особенно состоящих из десятков, а то и сотен приложений — работа с большим количеством файлов становится нелегким делом, потому что статические файлы расположены в разных папках. При базовой настройке ваши статичные файлы должны храниться в папке *static* каждого вашего приложения, и получается, что сколько приложений в вашем проекте, столько папок со статичными файлами нужно обрабатывать.

Для этого было создано приложение *django.contrib.staticfiles*: оно собирает статические файлы из всех ваших приложений (и остальных мест, которые вы укажите) в одном месте, что позволяет легко настроить выдачу статических файлов на реальном сервере.

Как уже было указано, в число статичных файлов обычно входят картинки всевозможных форматов (только если при каждом запросе вам нет нужды в дополнительной обработке изображения, однако такая потребность из разряда экзотических), сценарии на языке *JavaScript* и каскадные таблицы стилей (*CSS, Cascading Style Sheets*). О последних ниже будет краткое описание.

4.2 Основы CSS

CSS создавался с аналогичными целями, что и HTML, только направлен он не на описание содержания документа, а на *описание внешнего вида* всех элементов. Попробуйте пофантазировать несколько минут, как вы подошли бы к проблеме текстового описания документа в формате HTML. Первая трудность, с которой вы столкнулись бы, как определить внутри CSS-описания, каким элементам задавать стиль? Как отделить первый тег *h1* от второго, чтобы дать им разные стили? Или, например, как сделать так, чтобы каждая вторая строка в таблице имела легкий фон (кстати, это довольно распространенный подход для повышения удобства при чтении)?

Решением этой проблемы стали так называемые *селекторы*. Как известно, HTML-документы строятся на основании иерархии элементов, которая может быть наглядно представлена в древовидной форме. Элементы HTML друг для друга могут быть родительскими, дочерними, элементами-предками, элементами-потомками, сестринскими.

Элемент является родителем другого элемента, если в иерархической структуре документа он находится сразу, непосредственно над этим элементом. Элемент является предком другого элемента, если в иерархической структуре документа он находится где-то выше этого элемента.

Пускай, например, в документе присутствуют два абзаца *p*, включающие в себя шрифт с полужирным начертанием *b*. Тогда элементы *b* будут дочерними элементами своих родительских элементов *p* и потомками своих предков *body*. В свою очередь, для элементов *p* элемент *body* будет являться только родителем. И кроме того, эти два элемента *p* будут являться сестринскими элементами, как имеющими одного и того же родителя — *body*.

В CSS могут задаваться при помощи селекторов не только одиночные элементы, но и элементы, являющиеся потомками, дочерними или сестринскими элементами других элементов.

Теперь осталось только разобраться, как же эти селекторы работают. Каждое правило CSS из таблицы стилей имеет две основные части — *селектор* и *блок объявлений*. *Селектор*, расположенный в левой части правила, определяет, на какие части документа распространяется правило. *Блок объявлений* располагается в правой части правила, помещается в фигурные скобки и, в свою очередь, состоит из одного или более объявлений, разделённые знаком «;». Каждое объявление представляет собой сочетание свойства CSS и значения, разделённые знаком двоеточия «:». Селекторы могут группироваться в одной строке через запятую. В таком случае свойство применяется к каждому из них.

Синтаксис стилей на языке CSS:

```
селектор1, селектор2 {
   свойство1: значение1;
   свойство2: значение2;
   свойство3: значение3; }
```

Данный блок отыщет в разметке все элементы, которым соответствует селектор 1, затем отыщет все элементы, которым соответствует селектор 2 и для всей этой кучи применит стили, описанные в блоке объявлений.

Селекторы имеют несколько правил, с помощью которых можно найти почти что любой элемент в сколь угодной сложной иерархии HTML-разметки:

1) универсальный селектор, подходит для всех элементов;
   пример универсального селектора:

```
* {margin: 0; padding: 0;}
```

2) селектор по тегу;
   пример селектора по тегу:

```
p {font-family: Garamond, serif;}
```

3) селектор классов;
   пример селектора классов:

```
.note {color: red; background: yellow; font-weight: bold;}
```

4) селектор идентификаторов;
   пример селектора идентификатора:

```
#paragraph1 {margin: 0;}
```

5) селектор атрибутов;
   пример селектора атрибутов:

```
a[href=''http://www.somesite.com''] {font-weight: bold;}
```

6) селектор потомков (контекстными селекторами);
   пример селектора потомков:

```
div .note {color: red;}
/*
   в данном случае стиль применится к элементам
   с классом note, которые находятся внутри элементов div
*/
```

7) селектор дочерних элементов;
   пример селектора дочерних элементов:

```
p.note > b {color: green;}
/*
   в данном случае стиль применится к элементам
   тега b, которые находятся непосредственно внутри
   элементов p с классом note
   к самому элементу p стиль НЕ применится
*/
```

8) селектор сестринских элементов;
   пример селектора сестринских элементов:

```
h1 + p {font-size: 24px;}
/*
   в данном случае стиль применится к элементам
   тега p, которые находятся непосредственно после
   элементов h1
   к элементу h1 стиль НЕ применится
*/
```

9) селектор псевдоклассов;
   пример селектора псевдоклассов:

```
a:hover {color: yellow;}
/*
   в данном случае стиль применится к элементам-ссылкам
   тега a, которые находятся в наведенном состоянии;
   как только пользователь уберет курсор с элемента,
   эти стили отменятся
*/
```
[8].

Чтобы задать класс или идентификатор элементу, достаточно воспользоваться соответствующими атрибутами тегов.

Пример добавления класса к HTML-тегу:

```
<a href=///// class=//cool-link//>Я хорошая ссылка,
                                  кликни по мне</a>
```

таким образом задаются классы.

Пример добавления идентификатора к HTML-тегу:

```
<a href=///// id=//cool-link//>Я хорошая ссылка,
                               кликни по мне</a>
```

а таким образом идентификаторы. Первые от вторых отличаются тем, что идентификатор может быть только у одного элемента, и у каждого элемента не может быть больше одного идентификатора, а на классы такое ограничение не наложено. Соответственно, когда вы ищете элемент по идентификатору, можете не беспокоиться, что отыщется два или более элементов (если, конечно, верстку грамотный специалист делал и ошибок не натворил), что особенно часто требуется при манипулировании элементами с помощью JavaScript. Также следует упомянуть важную особенность: чтобы одному элементу задать несколько классов, нужно перечислить их через пробел.

Пример добавления нескольких классов к HTML-тегу:

```
<a href=///// class=//cool-link click-me one-more-class//>
 У меня целых три класса!</a>
```

Контрольные вопросы по главе 4

1. Почему статические файлы так называются?
2. В чем сложность обработки статических файлов в проектах на Django?
3. Что такое селекторы?
4. Как найти все элементы класса «*some-class*» внутри DOM?
5. Как найти все теги *p*, которые находятся внутри тегов *div*?
6. Как задать цвет шрифта всем элементам с идентификатором «*some-id*»?

Глава 5

МОДЕЛИ, ПРЕДСТАВЛЕНИЯ И КОНФИГУРАЦИЯ URL В DJANGO

5.1 Модели в Django

Логика современных веб-приложений часто требует обращения к базе данных. Такой управляемый данными сайт подключается к серверу базы данных, получает от него данные и отображает их на странице.

Некоторые сайты позволяют посетителям пополнять базу данных. Многие развитые сайты сочетают обе возможности. Например, Amazon.com — прекрасный пример сайта, управляемого данными. Страница каждого продукта представляет собой результат запроса к базе данных Amazon, отформатированный в виде HTML, а отправленный вами отзыв сохраняется в базе данных.

Django отлично подходит для создания управляемых данными сайтов, поскольку включает простые и вместе с тем мощные средства для выполнения запросов к базе данных из программы на языке Python.

Чтобы лучше понимать, что из себя представляют модели, сначала попробуйте решить следующую задачу: у вас есть некое хранилище данных в виде таблицы (пусть это будет всем знакомый файл для программы Microsoft Excel) и вам нужно сначала на языке, подобном Pythony, как-то определить структуру вашего файла, в котором есть несколько таблиц или листов, а затем программно как-то читать данные с минимальным количеством кода. Здесь немаловажной деталью является тот факт, что структура таблиц должна быть описана именно на языке программирования и внедрена самим скриптом в файл, а не наоборот, когда пользователь самостоятельно задает содержимое, а программа читает содержимое файла и распознаёт, какие строки есть в таблице. У последнего замечания (которое, естественно, имеет прямой аналог в реальной системе моделей Django) есть несколько оснований для существования:

- При работе с реальными таблицами баз данных скорость прочтения таблиц и разбор структуры всей БД будет занимать довольно много времени.

- Если сначала описывать структуру в таблице, а потом читать её из скрипта, программисту придется работать в двух реальностях, которые очень сильно отличаются друг от друга (как справедливо замечают создатели Django, «*писать на Python вообще приятно, и если представлять все на этом языке, то сокращается количество мысленных «переключений контекста». Чем дольше разработчику удается оставаться в рамках одного программного окружения и менталитета, тем выше его продуктивность. Когда приходится писать код на SQL, потом на Python, а потом снова на SQL, продуктивность падает*»).
- Высокоуровневые типы данных повышают продуктивность и степень повторной используемости кода. Так, в большинстве СУБД нет специального типа данных для представления адресов электронной почты или URL. А в моделях Django это возможно, потому что стоит лишь добавить пару строчек кода на Python, и обычное текстовое поле превращается в узконаправленное поле, в которое можно поместить только адрес URL, а если попытаться что-то некорректное, то в процессе сохранения просто напросто возникнет исключение [6].

Итак, решением, возможно, стало бы создать словарь, где в качестве ключей шли бы названия столбцов, а по каждому ключу хранился бы массив данных в этих данных. Однако такой вариант не очень удобен, потому что постоянно бы пришлось синхронизировать количество элементов в массивах, которые расположены по разным ключам.

Чтобы создать по такому принципу таблицу 5.1 вам пришлось бы создать примерно такой словарь:

```
{
    \\name//: [\\George//, \\Mike//, \\Steve//],
    \\age//: [32, 25, 43],
    \\occupation//: [\\doctor//, \\lawyer//, \\engineer//]
}
```

Таблица 5.1 – Список пользователей

Имя	Возраст	Профессия
George	32	Doctor
Mike	25	Lawyer
Steve	43	Engineer

Можно было бы применить другой подход: сделать словарь, где каждый ключ — опять же название столбца, и по этому ключу хранится ещё один, вложенный, словарь, в котором каждый ключ — это уже название строки. Однако и такой вариант не очень удобен, ведь пришлось бы для каждого столбца заново прописывать название строки.

Другой пример словаря с данными:

```
{
    \\age//: {\\George//: 32, \\Mike//: 25, \\Steve//:43},
```

```
        \\occupation//: {\\George//: \\doctor//, \\Mike//: \\lawyer//,
\\Steve//:\\engineer//},
    }
```

Тут из-за того, что имена и так хранятся во всех словарях, можно не описывать одно из полей.

И чтобы добавить нового пользователя, в любом случае пришлось бы по каждому из столбцов добавлять новое значение, не слабая такая задачка!

Создатели Django пошли по немного другому пути. Они предложили каждую таблицу в БД описывать через класс данных. Вы уже знаете, что в языке Python существуют такие классы данных (класс и тип данных по сущности одно и то же), как строка, число, список и другие. Однако в языке также существует возможность создавать свои собственные типы данных, которые могут быть как очень похожими, так и принципиально отличными от какого-либо встроенного типа. Для создания класса достаточно воспользоваться ключевым словом *class*, после которого нужно указать имя класса и в скобках отметить, от какого типа данных текущий класс должен наследоваться.

Пример создания модели для *Django*:

```
from django.db import models
class Article (models.Model):
    title = models.CharField(max_length=200)
    text = models.TextField()
```

С помощью таких трех строк кода была описана целая таблица в базе данных, которая имеет два столбца — *title* и *text*. Так как здесь производится наследование от уже написанного создателями Django класса *models.Model*, у вашего типа данных будет очень богатый функциональный набор по работе с базой данных. Например, чтобы создать новую запись вам будет достаточно выполнить следующую строку кода (пример создания нового экземпляра модели):

```
new_article = Article.objects.create(title=//Новая статья//,
text=//С интересным текстом//)
```

Теперь в вашей таблице появится новая строка, которая будет содержать данные о статье «*Новая статья*». Также для того, чтобы вы могли далее оперировать этой строкой, вы можете сохранить данные этой строки в переменной, которая в данном случае называется *new_article*. Например, можно изменить значение столбца *text* таким образом:

Пример изменения существующего экземпляра модели:

```
new_article.text = \\Теперь у статьи будет новый текст//
new_article.save()
```

Чтобы данные изменились не только в переменной Python, но и в базе данных, нужно вызвать метод *save()*.

Чтобы получить данные о всех записях в таблице, достаточно выполнить следующую строку кода (пример получения всех экземпляров модели *Article*):

```
all_articles = Article.objects.all()
```

А если помимо получения всех записей нужно ещё и отсортировать данные (пример получения всех экземпляров модели *Article* и их сортировки по названию):

```
all_articles_by_title = Article.objects.order_by('title')
```

Также от *models.Model* ваш класс унаследует метод *filter()*, который позволяет выбрать только те данные, что вам нужны.

Пример получения экземпляров модели *Article*, у которых название «Новая статья»:

```
some_articles = Article.objects.filter(title="Новая статья")
```

Если по заданному критерию в вашей базе данных есть несколько статей, то вернутся все они в формате *QuerySet*, который по поведению очень напоминает список, только он наделен многими дополнительными возможностям. Например, можно и сам объект *QuerySet* отфильтровать и отсортировать, причем таким же образом, как было показано ранее:

```
some_articles = Article.objects.filter(title="Новая статья")
# теперь some_articles — объект QuerySet
some_articles = some_articles.order_by('text')
# и у этого объекта есть такие же методы
same_articles = some_articles.filter(text="Какой-то текст
                                          статьи")
```

5.2 Представления и конфигурация URL в Django

Обычно представления создаются в файле *views.py*, однако это не задано жесткими рамками, название может быть абсолютно другим, однако принято называть его именно *views.py*, чтобы другие разработчики, читающие ваш код, сразу понимали, что в нем находится.

В качестве начального примера будет рассмотрено представление «Hello world». Ниже приведен код функции вместе с командами импорта, который нужно поместить в файл *views.py*.

Простейшая функция-представление *hello*:

```
from django.http import HttpResponse
def hello(request):
    return HttpResponse("Hello world")
```

- Сначала импортируется класс *HttpResponse*, который находится в модуле *django.http*. Импортировать его необходимо, потому что он используется в коде функции ниже.
- Далее определяется функция представления *hello*.
- Любая функция представления принимает по меньшей мере один параметр, который принято называть *request*. Это объект, содержащий информацию о текущем веб-запросе, в ответ на который была вызвана функция; он является экземпляром класса *django.http.HttpRequest*. В данном примере

мы не используем параметр *request*, тем не менее он должен быть первым параметром представления.

- Отметим, что имя функции представления не имеет значения, фреймворк Django не предъявляет каких-либо специфических требований к именам. Мы назвали ее *hello* просто потому, что это имя ясно показывает назначение представления, но могли бы назвать *hello_wonderful_beautiful_world* или еще как-то. В следующем разделе будет показано, каким образом Django находит эту функцию.
- Сама функция состоит всего из одной строки: она просто возвращает объект *HttpResponse*, инициализированный строкой «*Hello world*».

Главный урок состоит в том, что представление — обычная функция на языке Python, которая *принимает* экземпляр класса *HttpRequest* в качестве первого параметра и *возвращает* экземпляр класса *HttpResponse*. Чтобы функция на Python могла считаться функцией представления, она должна обладать этими двумя свойствами.

Если сейчас для нового проекта выполнить команду, запускающую сервер Django:

```
python manage.py runserver
```

то появится сообщение «Welcome to Django» без каких бы то ни было следов представления «Hello world». Объясняется это тем, что проект, который вы только что создали, еще ничего не знает о представлении *hello*; необходимо явно сообщить Django, что при обращении к некоторому URL должно активироваться это представление. (Если продолжить аналогию с публикацией статических HTML-страниц, то сейчас файл был только создан, но еще не загружен в каталог на сервере.) Чтобы связать функцию представления с URL, в Django используется механизм конфигурации URL.

Можно сказать, что конфигурация URL — это оглавление веб-сайта, созданного с помощью Django. По сути дела, речь идет об установлении соответствия между URL и функцией представления, которая должна вызываться при обращении к этому URL. Django получает указания: «Для этого адреса URL следует вызвать эту функцию, а для этого — эту». Например, «При обращении к URL */foo/* следует вызвать функцию представления *foo_view()*, которая находится в Python-модуле *views.py*».

Во время выполнения команды, создающей новый Django проект:

```
django-admin.py startproject project_name_here
```

сценарий автоматически создал конфигурацию URL: файл *urls.py*. По умолчанию она выглядит следующим образом (файл конфигурации URL, созданный автоматически):

```
from django.conf.urls.defaults import *

# Раскомментировать следующие две строки для активации
# административного интерфейса:
# from django.contrib import admin
# admin.autodiscover()
```

```
urlpatterns = patterns('',
# Пример:
# (r'^mysite/', include('mysite.foo.urls')),

# Раскомментировать строку admin/doc ниже и добавить
# 'django.contrib.admindocs' в INSTALLED_APPS для активации
# документации по административному интерфейсу:
# (r'^admin/doc/', include('django.contrib.admindocs.urls')),

# Раскомментировать следующую строку для активации
# административного интерфейса:
# (r'^admin/', include(admin.site.urls)),
)
```

В этой конфигурации URL по умолчанию некоторые часто используемые функции Django закомментированы, для их активации достаточно раскомментировать соответствующие строки. Если не обращать внимания на закомментированный код, то конфигурация URL сведется к следующему коду. Пример файла конфигурации URL без комментариев:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
)
```

- В первой строке импортируются все объекты из модуля *django.conf.urls.defaults* поддержки механизма конфигурации URL. В частности, импортируется функция *patterns*.
- Во второй строке производится вызов функции *patterns*, а возвращенный ею результат сохраняется в переменной *urlpatterns*. Функции *patterns* передается всего один аргумент — пустая строка. (С ее помощью можно задать общий префикс для функций представления.)

Главное здесь — переменная *urlpatterns*, которую Django ожидает найти в конфигурации URL. Она определяет соответствие между URL-адресами и обрабатывающим их кодом. По умолчанию конфигурация URL пуста, то есть приложение Django — чистая доска.

Чтобы добавить URL и представление в конфигурацию URL, достаточно включить кортеж, отображающий шаблон URL-адреса на функцию представления. Вот как подключается представление *hello* (пример добавления нового URL):

```
from django.conf.urls.defaults import *
from mysite.views import hello

urlpatterns = patterns('',
('^hello/$', hello),
)
```

Было внесено два изменения:

1. Во-первых, импортировали функцию представления *hello* из модуля, где она находится, — *mysite/views.py*, полное имя которого, согласно синтаксису импорта, принятому в Python, транслируется в *mysite.views*. (Здесь пред-

полагается, что *mysite/views.py* включен в путь, где интерпретатор Python пытается искать файлы.)

2. Далее в список шаблонов *urlpatterns* мы добавили строку (\^*hello/$*', *hello*). Такая строка называется шаблоном URL. Это кортеж Python, в котором первый элемент — строка с шаблоном (регулярное выражение), а второй — функция представления, соответствующая этому шаблону.

Тем самым мы сказали Django, что любой запрос к URL */hello/* должен обрабатываться функцией представления *hello*.

Имеет смысл подробнее остановиться на синтаксисе определения шаблона URL, так как он может показаться неочевидным. Вам требуется обеспечить совпадение с URL */hello/*, а шаблон выглядит несколько иначе. И вот почему:

- Django удаляет символ слеша в начале любого поступающего URL и только потом приступает к сопоставлению с шаблонами URL. Поэтому начальный символ слеша не включен в образец. (На первый взгляд, это требование противоречит здравому смыслу, зато позволяет многое упростить, например включение одних шабонов URL в другие.)
- Шаблон включает знаки вставки (^) и доллара ($). В регулярных выражениях эти символы имеют специальное значение: знак вставки означает, что совпадение с шаблоном должно начинаться в начале строки, а знак доллара — что совпадение с шаблоном должно заканчиваться в конце строки.
- Этот синтаксис проще объяснить на примере. Если бы мы задали шаблон \^*hello/*' (без знака доллара в конце), то ему соответствовал бы любой URL, начинающийся с */hello/* (например */hello/foo* и */hello/bar*, а не только*/hello/*). Аналогично, если бы мы опустили знак вставки в начале (например, *hello/$*'), то ему соответствовал бы любой URL, заканчивающийся строкой *hello/*, например */foo/bar/hello/*. Если написать просто *hello/* без знаков вставки и доллара, то подойдет вообще любой URL, содержащий строку *hello/*, например */foo/hello/bar*). Поэтому мы включаем оба знака — вставки и доллара, чтобы образцу соответствовал только URL */hello/* и ничего больше.
- Как правило, шаблоны URL начинаются знаком вставки и заканчиваются знаком доллара, но полезно все же иметь дополнительную гибкость на случай, если потребуется более хитрое сопоставление.
- Но что произойдет, если пользователь обратится к URL */hello* (без завершающего символа слеша)? Так как в образце URL завершающий символ слеша присутствует, то такой URL с ним не совпадет. Однако по умолчанию запрос к любому URL, который не соответствует ни одному шаблону URL и не заканчивается символом слеша, переадресуется на URL, отличающийся от исходного только добавленным в конце символом слеша. (Этот режим управляется параметром Django *APPEND_SLASH*.)
- Если вы предпочитаете завершать все URL-адреса символом слеша (как большинство разработчиков Django), то просто включайте завершающий символ слеша в конец каждого шаблона URL и не изменяйте принятое по умолчанию значение True параметра *APPEND_SLASH*. Если же вам боль-

ше нравятся URL-адреса, не завершающиеся символом слеша, или если вы решаете этот вопрос для каждого URL в отдельности, то задайте для параметра *APPEND_SLASH* значение *False* и разбирайтесь с символом слеша, как считаете нужным.

Теперь немного теории о том, как в целом работает вся система конфигурирования URL и представлений. Все начинается с файла параметров. При выполнении команды, запускающей сервер Django:

```
python manage.py runserver
```

сценарий ищет файл *settings.py* в том же каталоге, в котором находится файл *manage.py*. В этом файле хранятся всевозможные параметры настройки данного проекта Django, записанные заглавными буквами: *TEMPLATE_DIRS*, *DATABASE_NAME* и т. д. Самый важный параметр называется *ROOT_URLCONF*. Он говорит Django, какой Python-модуль следует использовать в качестве конфигурации URL для данного веб-сайта.

Вспомните, что команда, создающая новый Django-проект:

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

создала два файла: *settings.py* и *urls.py*. В автоматически сгенерированном файле *settings.py* параметр *ROOT_URLCONF* указывает на автоматически сгенерированный файл *urls.py*. Откройте *settings.py* и убедитесь сами; он должен выглядеть примерно так (пример переменной ROOT_URLCONF в файле settings.py):

```
ROOT_URLCONF = 'mysite.urls'
```

Это соответствует файлу *mysite/urls.py*. Когда поступает запрос к некоторому URL — например */hello/*, — фреймворк Django загружает конфигурацию URL из файла, на который указывает параметр *ROOT_URLCONF*. Далее он поочередно сравнивает каждый образец URL, представленный в конфигурации, с запрошенным URL, пока не найдет соответствие. Обнаружив соответствие, Django вызывает функцию представления, ассоциированную с найденным образцом, передавая ей в качестве первого параметра объект *HttpRequest*.

В примере первого представления выше вы видели, что такая функция должна возвращать объект *HttpResponse*. А Django сделает все остальное: превратит объект Python в веб-ответ с нужными HTTP-заголовками и телом (содержимым веб-страницы).

Вот перечень выполняемых шагов:

1. Поступает запрос к URL */hello/*.
2. Django находит корневую конфигурацию URL, сверяясь с параметром *ROOT_URLCONF*.
3. Django просматривает все образцы URL в конфигурации URL, пока не найдет первый, соответствующий URL */hello/*.
4. Если соответствие найдено, вызывается ассоциированная с ним функция представления.
5. Функция представления возвращает объект *HttpResponse*.

6. Django преобразует *HttpResponse* в соответствующий HTTP-ответ, который визуализируется в виде веб-страницы.

Вот вы и познакомились с основами создания страниц с помощью Django. На самом деле все просто: нужно лишь написать функции представления и отобразить их на URL-адреса с помощью конфигурации URL [6].

Контрольные вопросы по главе 5

1. По какому принципу работает Django: сначала программист должен сам создать базу данных и определить её структуры с помощью языка SQL или программа, созданная с помощью Django, сама способна определять таблицы в базах?
2. Каковы преимущества принципа, который использует Django при работе с базами данных?
3. Как создать новую запись в уже существующей таблице базы данных?
4. Как отсортировать список QuerySet по одному из полей?
5. Может ли файл views.py, где содержатся функции-представления, называться как-то иначе?
6. Чем функция-представление отличается от простых функций на языке Python?
7. Как указать Django, что какую-либо конкретную функцию-представление нужно вызывать по запросу на определенный адрес?
8. Зачем добавляются символы «^» и «&» в начале и конце регулярного выражения?
9. Что задает параметр *ROOT_URLCONF* внутри файла *settings.py* Django-проекта?

Глава 6

ФОРМЫ В DJANGO И СИСТЕМА АВТОРИЗАЦИИ И РЕГИСТРАЦИИ

6.1 Работа с формами в HTML и обработка данных из форм в представлениях Django

6.1.1 Работа с формами в представлениях Django

HTML-формы — становой хребет интерактивных веб-сайтов. Это может быть единственное поле для ввода поискового запроса, как на сайте Google, вездесущая форма для добавления комментария в блог или сложный специализированный интерфейс ввода данных. В этой главе будет рассказываться, как Django позволяет обратиться к данным, которые отправил пользователь, проверить их и что-то с ними сделать. Попутно пойдет речь об объектах *HttpRequest*.

Как вы помните, любая функция представления принимает объект *HttpRequest* в качестве первого параметра.

Пример функции-представления *hello*:

```
from django.http import HttpResponse
def hello(request):
    return HttpResponse(\\Hello world//)
```

У объекта с типом данных *HttpRequest*, каковым является переменная *request*, есть целый ряд интересных атрибутов и методов, с которыми необходимо познакомиться, чтобы знать, какие существуют возможности. С их помощью можно получить информацию о текущем запросе (например, имя пользователя или версию браузера, который загружает страницу вашего сайта) в момент выполнения функции представления.

Например, у объекта *request* есть атрибут *META* (доступ к нему можно получить так: *request.META*) — это словарь Python, содержащий все HTTP-заголовки данного запроса, включая IP-адрес пользователя и информацию об агенте поль-

зователя (обычно название и номер версии веб-браузера). В список входят как заголовки, отправленные пользователем, так и те, что были установлены вашим веб-сервером. Ниже перечислены некоторые часто встречающиеся ключи словаря:

- *HTTP_REFERER*: ссылающийся URL, если указан. (Обратите внимание на ошибку в написании слова REFERER.)
- *HTTP_USER_AGENT*: строка с описанием агента пользователя (если указана). Выглядит примерно так: «Mozilla 5.0 (X11; U; Linux i686) Gecko/20080829 Firefox/2.0.0.17».
- *REMOTE_ADDR*: IP-адрес клиента, например «12.345.67.89». (Если запрос проходил через прокси-серверы, то это может быть список IP-адресов, разделенных запятыми, например «12.345.67.89,23.456.78.90».)

Отметим, что поскольку *request.META* — обычный словарь Python, то при попытке обратиться к несуществующему ключу будет возбуждено исключение *KeyError*. (Ведь HTTP-заголовки — это внешние данные, то есть отправлены пользовательским браузером, доверять им нельзя, поэтому вы должны проектировать свое приложение так, чтобы оно корректно обрабатывало ситуацию, когда некоторый заголовок пуст или отсутствует.) Нужно либо использовать *try*/*except*-блок, либо осуществлять доступ по ключу методом *get()*.

Пример работы со словарем *request.META*:

```
# ПЛОХО!
def ua_display_bad(request):
    ua = request.META[\HTTP_USER_AGENT/]
    # Может возникнуть KeyError!
    return HttpResponse(\\Ваш браузер %s// % ua)
# ХОРОШО (ВАРИАНТ 1)
def ua_display_good1(request):
    try:
      ua = request.META[\HTTP_USER_AGENT/]
    except KeyError:
      ua = \unknown/
    return HttpResponse(\\Ваш браузер %s// % ua)
# ХОРОШО (ВАРИАНТ 2)
def ua_display_good2(request):
    ua = request.META.get(\HTTP_USER_AGENT/, \unknown/)
    return HttpResponse(\\Ваш браузер %s// % ua)
```

Помимо основных метаданных о запросе объект HttpRequest имеет два атрибута, в которых хранится отправленная пользователем информация: *request.GET* и *request.POST*. Тот и другой объекты похожи на словарь и дают доступ к данным, отправленным соответственно методом *GET* или *POST*. POST-данные обычно поступают из HTML-формы (тег <*form*>), а GET-данные — из формы или строки запроса, указанной в гиперссылке на странице.

В общем случае, с точки зрения разработки, у формы есть две стороны: пользовательский HTML-интерфейс и код для обработки отправленных данных на стороне сервера. С первой частью все просто, вот представление для отображения формы поиска.

Пример представления, отображающего форму:

```
from django.shortcuts import render

def search(request):
    return render(request, 'search_form.html', {})
```

Представление можно поместить в файл *application_name/views.py*.

Соответствующий шаблон *search_form.html* мог бы выглядеть так (пример простой формы вопроса):

```
<html>
  <head>
    <title> Поиск </title>
  </head>
  <body>
    <form method="POST">
      <input type="text" name="question">
      <input type="submit" value="Найти">
    </form>
  </body>
</html>
```

А вот образец URL в файле urls.py (пример конфигурации URL для страницы с формой):

```
from mysite.books.views import search
urlpatterns = patterns('',
# ...
(r'^search-form/$', search),
# ...
)
```

Если теперь запустить сервер разработки и зайти на страницу по адресу *http://127.0.0.1:8000/search-form/*, то появится интерфейс поиска. Все просто [6].

Однако, заполнив поле ввода в форме и нажав кнопку «*Найти*», единственное, что вы увидите..., та же самая форма, только уже опустевшая. Почему такое произошло? Потому что в данном случае, когда вы нажали кнопку «Найти», был отправлен запрос на тот же адрес URL (обратите внимание, что в адресной строке браузера ничего не поменялось), по этому адресу у вас в конфигурации URL поставлена та же функция представления, которая просто занимается рендерингом шаблона. Но раз при отправлении данных в форму происходит запуск представления *search*, то, значит, в нем и надо как-то обработать поступающие данные. Осталось только разобраться, а что пользователю сейчас от сервера надо: просто получить страницу с формой (когда пользователь первый раз зашел на сайт) или обработать данные с уже заполненной формы (то есть пользователь минимум второй раз на странице). Чтобы отличить одно от другого, можно проверить метод, которым отправлялись данные. Метод «*GET*» по стандарту используется только для получения какой-либо информации с сервера (в данном случае информацией является html-код возвращаемой страницы), методом же «*POST*» отправляются запросы, которые должны сохранить на сервере какую-либо информацию (например,

когда пользователь отправляет свой логин-пароль для регистрации). Соответственно, здесь в случае метода «*GET*» нужно просто отрендерить страницу с формой и вернуть её, а если использовался метод «*POST*», то нужно с помощью переменной *request* прочитать введенные данные и исходя из них вернуть какой-то определенный ответ или просто перенаправить на другую страницу. Для проверки можно просто сравнить содержимое атрибута *request.method*, в котором содержится название используемого метода (пример проверки метода запроса):

```
if request.method == "POST":
    # обработать данные формы, если метод POST
else:
    # просто вернуть страницу с формой, если метод GET
return render(request, 'create_post.html', {})
```

Чтобы получить доступ к самим данным, следует использовать объект, напоминающий словарь, *request.POST*. В нем содержатся все данные, которые были переданы в форме.

Пример получения доступа к данным, переданным через форму:

```
if request.method == "POST":
    # обработать данные формы, если метод POST
    question = request.POST[\question/]
else:
    # просто вернуть страницу с формой, если метод GET
    return render(request, 'create_post.html', {})
```

Обратите внимание, что у словаря *request.POST* имеется ключ *question*. Название ключа берется из имени инпута, который находится в отправленной форме. Помните шаблон формы, там как раз поле ввода имело следующий код:

```
<input type=//text// name=//question//>
```

Если бы вместо *name=//question//* было указано что-то другое, например *name=//interest//*, то и в функции-представлении пришлось бы обращаться по ключу *interest*.

Попробуйте запустить тестовый сервер и создать функцию-представление, которая будет спрашивать ваше имя и, если оно введено, будет возвращать приветствие по этому имени. Шаблон с именем *\\greetings.html//* для этой цели можете использовать следующий:

```
<html>
  <head>
    <title> Приветствие </title>
  </head>
  <body>
    <h1> Введите ваше имя </h1>
    <form method=//POST//>
      <input type=//text// name=//first_name//>
      <input type=//submit// value=//Найти//>
    </form>
    {% if first_name %}
```

```
      Привет, {{first_name}}!
    {% endif %}
  </body>
    </html>
```

Тогда файл *views.py* должен содержать примерно такую функцию (пример представления, возвращающего шаблон с приветствием):

```
from django.shortcuts import render
def search(request):
  if request.method == "POST":
    # обработать данные формы, если метод POST
    first_name = request.POST[\first_name/]
    # вернуть страницу с приветствием, если метод GET
    return render(
      request,
      'greetings.html',
      {\\first_name//: first_name}
    )
  else:
    # просто вернуть страницу с формой, если метод GET
      return render(request, 'greetings.html', {})
```

Однако у такого подхода есть существенный минус, который упоминался ранее: если пользователь воспользуется методом POST, но не отправит поле имени или это поле переименует, будет возбуждена ошибка *KeyError*, потому что в строке (пример неправильной работы со словарем request.POST):

```
first_name = request.POST[\first_name/]
```

будет обращение к ключу словаря, которого не существует. Хорошей практикой, как вы, конечно, помните, является использование метода *get()* у словарей, который позволяет безопасно извлекать значения по ключам и указывать значения по умолчанию вторым параметром (пример правильной работы со словарем *request.POST*):

```
first_name = request.POST.get(\first_name/, \Anonymous/)
```

В данном случае, если ключа «*first_name*» в словаре не окажется, вместо реального имени будет использоваться строка «*Anonymous*».

6.2 Система авторизации и регистрации в Django

Протокол HTTP спроектирован так, что не сохраняет информацию о состоянии соединения, то есть все запросы независимы друг от друга. Между предыдущим и следующим запросом нет никакой связи и не существует такого свойства запроса (IP-адрес, агент пользователя и т. п.), которое позволило бы надежно идентифицировать цепочку последовательных запросов от одного и того же лица.

Разработчики браузеров уже давно поняли, что отсутствие информации о состоянии в протоколе HTTP ставит серьезную проблему перед веб-программистами.

Поэтому на свет появились cookies. Cookie — это небольшой блок информации, который отправляется веб-сервером и сохраняется браузером. Запрашивая любую страницу с некоторого сервера, браузер посылает ему блок информации, который получил от него ранее.

Посмотрите, как действует этот механизм. Когда вы открываете браузер и вводите в адресной строке google.com, браузер посылает серверу Google HTTP-запрос, который начинается так (пример запроса на страницу google.com/):

```
GET / HTTP/1.1
Host: google.com
...
```

Полученный от Google ответ выглядит приблизительно так (пример ответа на запрос страницы по адресу google.com/):

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie:
   PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
expires=Sun, 17-Jan-2038 19:14:07 GMT;
path=/; domain=.google.com
Server: GWS/2.1
...
```

Обратите внимание на заголовок Set-Cookie. Браузер сохранит значение cookie (*PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671*) и будет отправлять его Google при каждом обращении к этому сайту. Поэтому при следующем посещении сайта Google запрос, отправленный браузером, будет иметь такой вид (пример следующего запроса на страницу google.com/):

```
GET / HTTP/1.1
Host: google.com
Cookie:
   PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671
...
```

Обнаружив заголовок Cookie, Google понимает, что запрос пришел от человека, уже посещавшего сайт. Значением cookie может быть, например, ключ в таблице базы данных, где хранятся сведения о пользователе. И Google мог бы (да, собственно, так и делает) отобразить на странице имя вашей учетной записи [6].

Соответственно, если вы пожелаете создать систему авторизации и регистрации пользователей на своем сайте, вам придется с помощью cookie в браузере клиента хранить данные о том, что текущий пользователь — это тот человек, который регистрировался на сайте полгода назад под ником «Vasya». Так что первое пришедшее решение «в лоб»: когда юзер входит в аккаунт (введя логин и пароль), можно просто ему ставить cookie «*username=vasya;*» и cookie будет прилеплена ко всем запросом. От вас потребуется лишь каждый раз читать значение cookie и радоваться, что такая система авторизации работает.

Однако тут есть маленькая проблема безопасности: любой злоумышленник сможет вручную установить у себя в браузере cookie «*username=vasya;*» и притво-

риться Васей! Он сможет читать его сообщения и рассылать спам от его имени! Естественно, это довольно легко решается, чуть-чуть усложнив систему на стороне сервера. Надо лишь устанавливать в куках не значение никнейма пользователя, а лишь какой-то бессмысленный для посторонних идентификатор, символов этак в 20. Тогда вам при каждом запросе достаточно всего лишь на сервере этот уникальный идентификатор проверять, в базе данных отыскивать, какому пользователю соответствует переданный id. В случае такой системы авторизации, чтобы взломать Васю, придется угадать именно тот айди, который сейчас установлен у аккаунта Васи (к тому же чаще всего id будет меняться время от времени). Такой подход к авторизации называется сессиями, а самому идентификатору внутри cookies присваивается имя наподобие session_id.

В принципе, чтобы это всю подобную систему проверить в действии, достаточно сначала зайти в административной панели Django в свой аккаунт, затем на вкладке Resourses панели разработчиков Chrome найти установленные cookies. Среди них как раз должна находиться cookies с именем sessionid. Она-то при каждом, даже пустом, запросе передается серверу, считывается и определяет пользователя, который этому идентификатор соответствует. Ниже на рисунке 6.1 показан скриншот, который иллюстрирует, как в браузере найти установленные cookies:

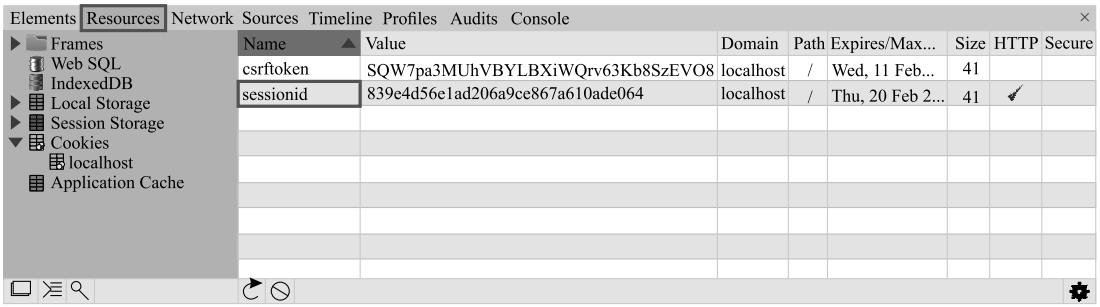


Рис. 6.1 – Отображение cookies в панели разработчика Google Chrome

Для непосредственной работы с системой авторизации/регистрации пользователей в Django существует написанное создателями фреймворка приложение *django.contrib.auth*. В нем хранятся все необходимые функции и модели для создания, редактирования пользователей, а также функции для реализации входа в аккаунт и выхода из аккаунта.

Контрольные вопросы по главе 6

1. Что такое формы в HTML и зачем они нужны?
2. Как в функции-представлении определить, была ли заполнена форма или пользователь впервые на странице? Какие должны быть выполнены требования, чтобы предложенный вами метод работал?

3. Как получить доступ к отправленным пользователем данным из функции-представления?
4. Что будет, если в теге <*input*> изменить атрибут *name*?
5. Что такое cookies и почему в них нельзя хранить важные данные (например, под каким аккаунтом авторизован текущий пользователь)?
6. Как вы можете посмотреть, какие cookies сейчас установлены для данной страницы вашего браузера?

Глава 7

ОСНОВЫ ЯЗЫКА СЦЕНАРИЕВ JAVASCRIPT

7.1 Базовые операторы, типы данных, функции и глобальные переменные JavaScript

7.1.1 Операторы, типы данных, функции и глобальные переменные JavaScript

Язык JavaScript — это бесплатный язык сценариев, исполняемых на стороне клиента, который позволяет создавать интерактивные HTML-страницы. «На стороне клиента» (client-side) означает, что JavaScript запускается в Web-браузере и не используется на стороне сервера. Сценарии на стороне клиента позволяют пользователю интерактивно взаимодействовать с Web-страницей после того, как она была обработана сервером и загружена Web-браузером. Например, в GoogleMaps применение языка JavaScript дает пользователям возможность взаимодействовать с картой, перемещать её, приближать и удалять и т. д. Без JavaScript Web-страницу приходилось бы обновлять при каждом взаимодействии с пользователем, если, конечно, не использовать такие плагины, как Adobe Flash или Microsoft Silverlight. Язык JavaScript не требует плагинов.

Так как JavaScript обеспечивает взаимодействие пользователя с Web-страницей после её загрузки, разработчики обычно используют его для решения следующих задач:

- динамическое добавление, редактирование и удаление HTML-элементов и их значений;
- проверка содержимого web-форм перед отправкой на сервер;
- создание на компьютере клиента cookie-файлов для сохранения и получения данных при последующих визитах.

Перед началом изучения языка следует познакомиться с его основными принципами:

- чтобы добавить JavaScript-код в HTML-файл, его необходимо поместить внутрь тегов script, добавить атрибут text/javascript и указать в *src* путь к файлу;
- все выражения в JavaScript оканчиваются точкой с запятой;
- язык чувствителен к регистру символов;
- имена всех переменных должны начинаться с буквы или символа подчеркивания;
- можно использовать комментарии, чтобы выделить определенные строки в сценарии; комментарии должны начинаться с двойного прямого слеша (//), за которым следует текст комментария;
- комментарии также можно использовать для отключения фрагментов сценария; для отключения нескольких строк можно использовать конструкцию */* фрагмент кода */*; любой код внутри */**/* не будет запускаться во время выполнения.

Проще всего добавить JavaScript-код на Web-страницу, если загрузить его из внешнего JS-файла с помощью атрибута *src* в теге *script* (пример добавления сценария JavaScript к веб-странице):

```
<script type=''text/javascript''src=''path/to/javascript-fi-
                                        le.js''></script>
```

Существуют переменные двух типов: локальные и глобальные. Локальные переменные объявляются с помощью ключевого слова *var*, а глобальные — без него. При использовании *var* переменная считается локальной, так как она доступна только в той области, где была объявлена. Например, если объявить локальную переменную внутри функции (этот случай будет рассмотрен в следующих разделах), то к ней нельзя будет получить доступ за пределами функции и она станет локальной переменной данной функции. Если же объявить эту же переменную без *var*, то она будет доступна по всему сценарию, а не только в этой функции.

Ниже приведен пример создания локальной переменной *num*, которой присвоено значение 10:

```
var num = 10;
```

Чтобы сохранить арифметическое выражение в переменной, достаточно присвоить переменной вычисленное значение, как показано в примере ниже. В переменной будет храниться вычисленный результат, а не само выражение.

Пример создания переменной num, как суммы двух чисел:

```
var num = (5 + 5);
```

В JavaScript операторы требуются для выполнения любого действия — сложения, вычитания, сравнения и т. д. В языке существует четыре типа операторов:

- арифметические:
  - сложение: «+»;
  - вычитание: «–»;

  - умножение: «*»;
  - деление: «/»;
  - вычисление остатка от деления: «%»;
  - инкремент (увеличение на единицу): «++»;
  - декремент (уменьшение на единицу): «--»;
- присваивания:
  - простое присваивание: «=»;
  - присвоить переменной результат сложения: «+ =»;
  - присвоить переменной результат вычитания: «- =»;
  - присвоить переменной результат умножения: «* =»;
  - присвоить переменной результат деления: «/ =»;
  - присвоить переменной результат вычисления остатка от деления: «% =»;
- сравнения:
  - равенство: «==»;
  - равенство по значению и типу объекта: «===»;
  - неравенство: «! =»;
  - больше чем: «>»;
  - меньше чем: «<»;
  - больше или равно: «>=»;
  - меньше или равно: «<=»;
- логические:
  - И: «&&»;
  - ИЛИ: «||»;
  - НЕ: «!».

Массивы. Массивы похожи на переменные, но отличаются от них тем, что могут хранить несколько значений и выражений под одним именем. Возможность хранения нескольких значений в одной переменой — это главное преимущество массива. В JavaScript для массивов не существует ограничений на количество или тип данных, которые будут в нем храниться, пока эти данные находятся в области видимости массива. Доступ к значению любого элемента массива можно получить в любой момент времени после объявления массива в сценарии. Хотя в JS-массиве можно хранить данные любого типа, включая другие массивы, обычно в массиве хранятся однородные данные и его название также каким-то образом связано с хранящимися данными. Ниже представлены примеры использования массивов для хранения однородных данных.

Пример создания массива:

```
var colors = new Array("orange", "blue", "red", "brown");
// можно создавать массивы с помощью функции Array
var shapes = [" circle", " square", " triangle", " pentagon"];
```

```
// а можно сразу задавать данные для массива, два способа по
// большей части одинаковы
```

Хотя доступ к значениям в массиве осуществляется легко, но есть одна тонкость. Массив всегда начинается с 0-го, а не первого элемента, что поначалу может смущать. Нумерация элементов начинается с 0, 1, 2, 3 и т. д. Для доступа к элементу массива необходимо использовать его идентификатор, соответствующий позиции элемента в массиве.

Пример обращения к элементам массива по индексу:

```
console.log("Orange: "+ colors[0]);
console.log("Blue: "+ colors[1]);
console.log("Red: "+ colors[2]);
console.log("Brown: "+ colors[3])
```

Тут для вывода переменных использовалась функция *console.log()*, которая позволяет узнать значение переменной в панели разработки таких браузеров, как Google Chrome, Mozzila Firefox и других. Чтобы посмотреть на вывод, достаточно включить панель (обычно это клавиша f12) и перейти на вкладку «Console».

Тем же способом можно присвоить значение элементу, находящемуся на определенной позиции в массиве, или обновить значение элемента в массиве. Пример обновления значения массива по индексу:

```
var colors = new Array();
colors[0] = "orange";
colors[1] = "blue";
colors[2] = "red";
colors[3] = "brown";
console.log(colors)
```

Ассоциативные массивы, они же объекты. Список пар *ключ:значение*. Отличаются от массивов в первую очередь тем, что доступ к определенному элементу производится не за счет позиции в массиве, а за счет имени этого значения. Для создания объектов используются фигурные скобки. Пример создания объектов:

```
emptyObject = {};
newObject = {\name/:/John/, \age/:30};
```

Тип данных для ключа должен быть обязательно строкой, а вот значение может принадлежать к любому классу переменных. Даже функцию можно поместить в качестве значения для одного из элементов объекта. Пример добавления функции в один из элементов объекта:

```
someMan = {\name/:\John/, 'age':30, \sayHello/: function(){
     console.log(\Hello!/);
}};
someMan.sayHello();
// выведет: Hello!
```

Такие функции, которые не существуют сами по себе, а являются элементом (иначе атрибутом) некоего объекта, называются методы. Так что функция *sayHello* — это метод объекта *someMan*.

Чтобы прочитать свойство объекта, можно имя свойства указать через точку либо внутри квадратных скобок. Пример обращения к свойству через точку и через квадратные скобки:

```
someMan.name // \John/
someMan[\age/] // 30
```

Чтобы изменить атрибут объекта, достаточно использовать оператор присваивания для этого атрибута (кстати, если такого атрибута не существует у объекта, то он будет создан, и это главный способ создания новых ключей и их значений).

Пример изменения и создания атрибутов объекта:

```
someMan[\age/] = 31; // теперь Джону стукнул 31
someMan[\occupation/] = \doctor/;
// и у него появилась профессия — он стал доктором
```

Условия. Во всех языках программирования основой для создания бизнес-логики являются условные выражения, и JavaScript — не исключение. Условные выражения определяют, какое действие должно быть выполнено в зависимости от условий, установленных в сценарии.

Пример оператора условия:

```
var num = 10;
if(num == 5)
{
   console.log("num равно 5");
}
else
{
     console.log("num НЕ равно 5, num равно: "+ num);
}
```

Функции. Функции обладают целым рядом преимуществ. Во-первых, они служат контейнерами для сценариев, которые будут выполняться только при наступлении события или вызове функции. Также функции не выполняются при первой загрузке Web-страницы, когда Web-браузер загружает и выполняет сценарии, находящиеся на этой странице. Предназначение функции — хранить код для выполнения определенной задачи, чтобы его можно было вызвать в любое время.

Функция оформляется в коде очень просто: она начинается с ключевого слова *function*, за которым следует пробел и название функции. Название функции может быть любым, но желательно, чтобы оно имело отношение к задаче, которую выполняет функция. Ниже приведен пример функции, изменяющей значение существующей переменной.

Пример создания функции:

```
var num = 10;
function changeVariableValue() {
   num = 11;
}
changeVariableValue();
console.log("num равно: "+ num);
```

Здесь демонстрируются не только структура функции, но и пример её вызова для изменения значения переменной. В представленном фрагменте можно изменить значение переменной, так как она объявлена в области видимости основного сценария, как и сама функция, поэтому функция знает о существовании переменной. Но если объявить переменную внутри функции, то к ней нельзя будет получить доступ за пределами функции.

Функции также могут принимать данные через входные параметры. Функция может иметь один или несколько параметров, и в вызове функции будет передаваться столько параметров, сколько объявлено в сигнатуре функции. Важно не путать понятие «параметр» с «аргументом». Параметр — это часть определения (или сигнатуры) функции, а аргумент — это выражение, используемое при вызове функции. Ниже приведен пример вызова функции, имеющей параметры, и вызова функции с использованием аргументов.

Пример создания функции с параметрами:

```
var num = 10;
function increase(_num)
{
    _num++;
}
increase(num);
console.log("num равно: "+ num);
// выведет: num равно: 11
```

В функциях также используются выражения *return*. Эти выражения возвращают значение, полученное после выполнения сценария в функции. Например, можно присвоить переменной значение, возвращенное функцией. Ниже показано, как вернуть значение из функции после выполнения сценария.

Функция, возвращающая сумму переданных ей аргументов:

```
function add(_num1, _num2)
{
   return _num1 + _num2;
}
var num = add(10, 30);
console.log("num равно: "+ num);
```

После запуска на странице будет напечатано «*num равно 40*». Удобство этой функции в том, что можно передать в функцию два параметра, она выполнит их сложение и вернет результат, который и будет присвоен переменной, вместо прямого присваивания.

Циклы. Как было продемонстрировано выше, массивы — это удобный способ хранить большое количество многократно используемых данных, но это ещё не всё. Циклы *for* и *while* предоставляют средства для итерирования по этим массивам, доступа к их значениям и использования их в сценариях.

Чаще всего в JavaScript используется цикл *for*. Этот цикл обычно состоит из переменой, которой присваивается числовое значение, затем эта переменная используется с оператором сравнения для проверки условия, а в конце числовое значение

переменной увеличивается или уменьшается. В ходе сравнения в цикле *for* обычно определяется, что числовое значение исходной переменной больше или меньше определенного числового значения. Если это условие выполняется (равно *true*), то цикл выполняется и значение переменной увеличивается или уменьшается, пока условие не станет равным *false*. Ниже приведен пример цикла *for*, работающего, пока числовое значение переменной меньше длины массива.

Пример цикла *for*:

```
var colors = new Array("orange", "blue", "red", "brown");
for(var i=0; i<colors.length; i++) {
   console.log("цвет: "+ colors[i] + "<br/>");
}
```

Свойство *length* (длина) у массива содержит числовое значение, равное количеству элементов в массиве. Не забывайте, что индексация массива начинается с 0. Поэтому если в массиве 4 элемента, то значение *length* равно 4, а доступные индексы — 0, 1, 2, 3.

Другой тип цикла — цикл *while*. Считается, что этот цикл работает «быстрее» цикла *for*, но он не очень подходит для итерирования по массивам, и его лучше применять в случаях, когда необходимо выполнять сценарий до тех пор, пока условие истинно. Ниже приведен исходный код цикла *while*, в котором сценарий выполняется до тех пор, пока значение численной переменной меньше 10.

Пример цикла *while*:

```
var i = 0;
while(i < 10)
{
   console.log(i + "\n"); //\n—символ переноса строки
   i++;
} [9].
```

Также в JavaScript довольно большую роль играют глобальные объекты *window* и *document*.

Объект *window* сочетает два в одном: глобальный объект javascript и окно браузера.

Для обращения к функциям и методам *window* не нужно указывать объект.

Обращение к функции decodeURI через объект window и без него:

```
window.decodeURI(. . .)
// то же что и
decodeURI(. . .)
```

Вообще, любая переменная в конечном счете (если не найдена локально) ищется в глобальном объекте.

Ко всем глобальным объектам можно обращаться как к свойстам window:

```
// можно указать window явно—будет работать:
str = new window.String("test")
a = new window.Array(1,2,3)
```

Кроме роли «глобального объекта», *window* также предоставляет интерфейс для работы с окном браузера.

Функция focus():

```
window.focus()
```

обычно, если окно не минимизировано, оно делается его текущим и выводится на передний план. Если же окно минимизировано, то его обозначение в списке минимизированных окон начинает мигать.

Функция open():

```
var newWin = window.open(strUrl, winName [, winParams])
```

Метод *open* создает новое окно браузера, аналогично команде «Новое окно» в меню браузера. Обычно это не вкладка, а именно новое окно, но в некоторых браузерах можно настроить то или иное поведение явным образом. Если параметр *strUrl* — пустая строка, то в окно будет загружен пустой ресурс *about:blank*.

В любом случае, загрузка осуществляется асинхронно. Создается пустое окно, загрузка ресурса в которое начнется уже после завершения исполнения текущего блока кода. Аргументы:

- *strUrl* — адрес для загрузки в новое окно, любая адресная строка, которую поддерживает браузер.
- *winName* — имя нового окна.
- *winParams* — необязательный список настроек, с которыми открывать новое окно.

Функция close():

```
window.close()
```

закрывает текущее окно. Если закрываемое окно не было открыто при помощи *window.open()*, то при его закрытии выводится предупреждение. Посетитель может отклонить закрытие [10].

В свою очередь объект *document* представляет собой «окно» для работы с DOM вашей HTML-разметки. Если вам будет нужно найти какой-либо элемент в DOM-е, изменить у какого-либо элемента CSS-свойство или удалить целую ветку элементов, вам придется иметь дело с *document*, даже когда вы этого явно не делаете (например, далее будет описываться библиотека *jQuery*, которая позволяет не писать много шаблонного кода и сама за программиста работает напрямую с объектом *document*).

Контрольные вопросы по главе 7

1. Как добавить JavaScript сценарий к веб-странице?
2. Чем характерно объявление переменной с использованием *var*?
3. Какие есть два способа создания массивов в JavaScript?

4. Как получить доступ к 5-ому элементу массива?
5. Как изменить значение ассоциативного массива по заданному атрибуту?
6. Видны ли внешние переменные из тела какой-либо функции?
7. Что выполняет выражение *return* внутри тела функции?
8. Для чего используется объект *document*?

Глава 8

ПРИНЦИПЫ РАБОТЫ С DOM ПРИ ПОМОЩИ JAVASCRIPT

8.1 Манипуляции элементами DOM, добавление обработчиков на пользовательские события

Основным инструментом работы и динамических изменений на странице является DOM (Document Object Model) — объектная модель, используемая для HTML-документов.

Согласно DOM-модели документ является иерархией. Каждый HTML-тег образует отдельный элемент-узел, каждый фрагмент текста — текстовый элемент и т. п.

Проще говоря, DOM — это представление документа в виде дерева тегов. Это дерево образуется за счет вложенной структуры тегов плюс текстовые фрагменты страницы, каждый из которых образует отдельный узел.

Построим, для начала, дерево DOM для следующего документа.

HTML-разметка для документа о лосе:

```
<html>
  <head>
    <title>
      О лосях
    </title>
  </head>
  <body>
    Правда о лосях
    <ol>
      <li>
        Лось — животное хитрое
      </li>
      <li>
        ... И коварное
```

```
        </li>
      </ol>
    </body>
</html>
```

Внутри <*html*> находятся два узла: <*head*> и <*body*> — они становятся дочерними узлами для <*html*>. Теги образуют узлы-элементы (*element node*). Текст представлен текстовыми узлами (*text node*). И то, и другое — равноправные узлы дерева DOM.

На рисунке 8.1 схематически представлена структура созданной разметки, если внутри квадрата указано название тега, то это элемент-узел, а если просто написан текст, то, соответственно, это текстовый элемент.

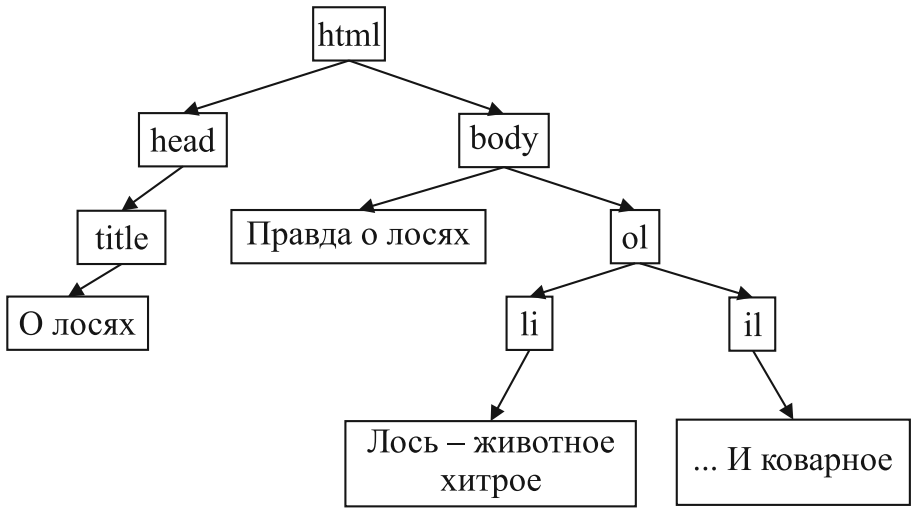


Рис. 8.1 – DOM-структура HTML-документа

Для манипуляций с DOM используется объект *document*.

Используя *document*, можно получать нужный элемент дерева и менять его содержание.

Поиск элемента в DOM, удаление и добавление потомков:

```
var ol = document.getElementsByTagName('ol')[0]
// найти все элементы, имя тега которых—\\ol// так как
// таких элементов может быть много (а может быть
// и всего один), то данный метод всегда возвращает массив
// но в данном примере массив не нужен, нужен сам
// единственный на странице список
var firstChild = ol.removeChild(ol.firstChild)
// удалить первый элемент списка
var secondChild = ol.removeChild(ol.firstChild)
// удалить второй элемент списка, который после
// предыдущего оператора стал первым в списке
ol.appendChild(secondChild)
// вновь добавить удаленный элемент, но теперь он уже станет
// первым, потому что из списка до этого было все удалено
```

```
ol.appendChild(firstChild)
// элемент ранее бывший первым теперь стал вторым, потому что
// когда его добавили, в списке уже была занята одна позиция.
```

У DOM-элементов есть масса свойств. Обычно используется максимум треть из них. Некоторые из них можно читать и устанавливать, другие — только читать.

tagName — Атрибут есть у элементов-тегов и содержит имя тега в верхнем регистре, только для чтения.

Использование атрибута tagName:

```
console.log(document.body.tagName) // => BODY
```

style — Это свойство управляет стилем. Оно аналогично установке стиля в CSS. Есть общее правило замены — если CSS-атрибут имеет дефисы, то для установки *style* нужно заменить их на верхний регистр букв. Например, для установки свойства *z-index* в 1000, нужно поставить:

Использование атрибута zIndex:

```
element.style.zIndex = 1000
```

innerHTML — оно содержит весь HTML-код внутри узла, и его можно менять. Свойство *innerHTML* применяется, в основном, для динамического изменения содержания страницы, например (использование атрибута innerHTML):

```
document.getElementById('footer').innerHTML = '<h1>Bye!</h1>
<p>See you later!</p>'
```

Пожалуй, *innerHTML* — одно из наиболее часто используемых свойств DOM-элемента.

onclick, *onkeypress*, *onfocus*... — и другие свойства, начинающиеся на «*on*...», хранят функции-обработчики соответствующих событий. Например, можно присвоить обработчик события «click» [11].

Теперь нужно узнать поподробней о том, как работает поиск элементов по DOM. Вы уже видели минимум две реализации поиска: первый с помощью метода.

Пример метода *getElementsByTagName*:

```
document.getElementsByTagName()
```

этот метод возвращает множество всех элементов, которые имеют заданный в аргументах тег. Например, поиск всех элементов div в DOM:

```
document.getElementsByTagName(\div/)
```

вернет все элементы <*div*> на странице. Естественно, работать напрямую с каждым элементом можно, только если обращаться к каждому элементу по индексу.

Добавление всем элементам div в DOM-е текста с номером текущего элемента:

```
allDivs = document.getElementsByTagName(\div/)
for (var i=0; i < colors.allDivs; i++){
allDivs[i].innerTHML = \\<p>Это \\ + i + \\-й блок</p>\\
}
```

allDivs[i] — именно с помощью такого обращения можно найти один конкретный элемент.

Если же вас во всех иерархиях DOM интересует один-единственный вполне определенный элемент, то его можно найти по заданному в атрибуте идентификатору. Это и есть второй способ, который уже встречался вам в примерах.

Пример использования метода *getElementById* для изменения CSS-стиля:

```
document.getElementById('some-id').style.color = \\red//;
```

Данный способ принципиально отличается тем, что возвращает не список узлов, как предыдущий пример, а конкретный узел, у которого можно менять состояние, не обращаясь к индексам. Однако такой подход требует, чтобы у нужного вам элемента был установлен идентификатор, что не всегда удобно (ведь придется все время придумывать для каждого элемента в DOM уникальное имя).

И третий способ, который во многом напоминает первый, это поиск элементов по классу.

Поиск всех элементов с классом buttom с помощью *getElementsByClassName*:

```
allButtons = document.getElementsByClassName(''button'');
```

Данная строка кода сохранит в переменную *allButtons* все элементы, которые имеют класс «*button*». Это становится довольно удобным, когда у вас есть список однородных элементов и вы в цикле пожелаете изменить им какое-либо свойство.

События. Очень важной частью работы с DOM является обработка различных событий, например клик мышкой по какому-либо элементу. Для этого в JavaScript предусмотрена следующая функция (пример добавления функции-обработчика к элементу):

```
document.getElementById('some-id')
     .addEventListener(\click/, function(){
       console.log(\you clicked on #some-id/)
});
```

С помощью данного кода к элементу с идентификатором «*some-id*» будет добавлен обработчик клика в виде функции, которая выводит простой текст в консоль. Данная функция может принимать один аргумент — объект события, где будет храниться основная информация о том моменте, когда пользователь запустил функцию.

Пример использования объекта события:

```
document.getElementById('some-id')
     .addEventListener(\click/, function(eventObject){
     console.log(\Вы кликнули по элементу /,
                                    eventObject.target);
     if (eventObject.shiftKey){
       console.log(\Была зажата клавиша Shift/);
     }
     console.log(\Координата X курсора /,
                                    eventObject.clientX);
     console.log(\Координата Y курсора /,
                                    eventObject.clientY);
});
```

С помощью описанных выше атрибутов вы сможете определить:

- У какого именно DOM-элемента было вызвано событие. Это свойство становится особенно важным, когда один обработчик создается для нескольких элементов, тогда остро встает вопрос, а как же найти именно тот элемент, что интересует пользователя.
- Была ли зажата клавиша Shift во время выполнения события. Естественно, в данном случае Shift всего лишь пример, узнать можно почти что любую подобную информацию, главное знать название атрибута для объекта *eventObject*.
- Какие были координаты у курсора в момент нажатия.

Контрольные вопросы по главе 8

1. Как найти в DOM все элементы *div*?
2. Как добавить один элемент внутрь другого?
3. Как добавить обработчик события ко всем элементам класса «*some-class*»?
4. Что в себе содержит первый параметр, передаваемый в функцию-обработчик?
5. Как внутри функции-обработчика обратиться к элементу, который вызвало событие (например, изменить CSS-свойство того элемента, по которому кликнул пользователь)?

Глава 9

ПРЕИМУЩЕСТВА БИБЛИОТЕКИ JQUERY

9.1 Основные принципы работы с библиотекой jQuery для языка JavaScript

jQuery — библиотека, которая позволяет значительно сократить количество монотонного кода, сделать ваш скрипт более читаемым и менее склонным к ошибкам. jQuery в первую очередь направлена на манипуляцию элементами DOM, так что вряд ли с помощью неё вам удастся выполнить какие-то сложные научные расчеты или какую-то подобную сложную вычислительную работу. Так что обязательно запомните — jQuery лишь помогает работать с DOM, почти все остальные задачи, возложенные на JavaScript, вам придется решать другими путями, хотя тут есть редкие исключения, например создание запросов на сервер. В jQuery есть метод *ajax()*, который позволяет отправлять и получать данные с сервера без перезагрузки страницы.

Банальное описание принципов работы jQuery выглядит примерно так:

1. Подключение файла библиотеки, например через тег *<script>*, после которого в вашей программе появляется глобальная переменная с коротким названием «$» — да, название этого объекта — это всего лишь знак доллара. Вот и первое сокращение ненужного кода, не нужно писать длинного имени переменной *document* для манипуляций элементами.
2. Затем в любом месте программы вы можете вызвать объект «$» как самую обычную функцию, передав ей в качестве параметра CSS-селектор. Например, $(*div.post*\) вернет список всех div-элементов, у которых есть класс «post». Вызов такой функции всегда возвращает специальный объект JavaScript, содержащий массив элементов DOM, соответствующий селектору. У этого специального объекта есть много методов, которые собственно и позволяют различным образом манипулировать элементами.
3. У объекта, который вернул вызов функции $*()* вызвать любую jQuery функцию. Например, функция *addClass(\\new-class\\)* добавит всем элементам в списке новый класс «*new-class*».

4. Любой выполненный метод ещё раз возвращает специальный объект jQuery, так что можно продолжить работу с тем же списком элементов и таким образом создать цепочку манипуляций над элементами: $(*div.post*/). *addClass*(*new-class*//).*hide().remove()* — сначала добавить класс элементам, потом их скрыть, а затем и вовсе удалить.

Из-за того что только объекты jQuery обладают всеми этими широкими возможностями, обычные элементы DOM нужно сначала «обрамить» вызовом функции jQuery, которая вернет тот же список тех же элементов, только уже с нужными возможностями.

Исчезновение элемента с последующим добавлением потомка с помощью jQuery:

```
$ (e.target).hide().append(\<p>hey!</p>/)
```

В данном случае, внутри переменной *e.target* хранится уже знакомый вам обычный узел DOM-а, однако, «обрамив» этот узел вызовом jQuery, вы сможете к данному элементу применить все возможности библиотеки.

Также jQuery позволяет гораздо более удобным способом добавлять обработчики на различные события. Это становится еще более актуально, если один и тот же обработчик нужно добавить сразу к нескольким элементам. С помощью данной библиотеки можно всего за одну строку добавить обработчик какого-либо события сразу всем элементам, которые подходят на заданные селекторы, и совсем отпадает нужда в создании циклов.

Добавление функции-обработчика на событие click сразу для нескольких элементов с помощью jQuery:

```
$ (\.one-post/).click(function(e){
     console.log(\you clicked me/);
})
```

Помимо приведенных выше методов, jQuery предоставляет богатейший набор возможностей, список которых вы можете увидеть на официальном сайте проекта http://jquery.com/.

Контрольные вопросы по главе 9

1. Имеют ли элементы, найденные с помощью метода document.getElement ById(’some-id’), доступ к jQuery методам (например, к методу *addClass*)? Если нет, то как это исправить?
2. Как добавить функцию-обработчик на событие *click* у всех элементов с классом «*some-class*»?
3. Как с помощью jQuery удалить элемент из DOM?
4. Как с помощью jQuery скрыть элемент?
5. Что упрощает библиотека jQuery при добавлении функции-обработчика на какое-либо событие?

ЗАКЛЮЧЕНИЕ

В учебном пособии были освещены многие аспекты веб-разработки на языках Python и JavaScript. Однако, помимо вышеописанного, в создании сервисов существует масса других очень важных сторон, которые не были должным образом освещены. Каждому, кто желает создавать относительно сложные современные, динамичные и красивые веб-страницы с помощью Django и jQuery, также следует ознакомиться с такими технологиями, как:

- AJAX, который позволяет отправлять запросы на сервер и получать оттуда данные без перезагрузки страницы;
- формы в Django, которые позволяют выносить логику обработки и сохранения данных из форм в другие модули, способствуя систематизации вашего приложения;
- AMD, которая позволяет грамотно реализовывать модульный подход в программировании на JavaScript;
- CSS3-свойства, которые позволяют без вмешательства JavaScript-сценариев добавлять различные эффекты на страницы;
- многое другое, включая даже иные языки программирования и библиотеки, ведь для разных задач могут быть более подходящие инструменты, чем те, которыми вы уже владеете.

ЛИТЕРАТУРА

[1] Клиент-сервер [Электронный ресурс] // Wikimedia Foundation, Inc. — URL : http://ru.wikipedia.org/wiki/Клиент-сервер (дата обращения: 21.05.2014).

[2] Учим Python качественно [Электронный ресурс] // Thematic Media. — URL : http://habrahabr.ru/post/150302/ (дата обращения: 22.05.2014).

[3] Основы Python в кратком изложении [Электронный ресурс] // Thematic Media. — URL : http://habrahabr.ru/post/29778/ (дата обращения: 22.05.2014).

[4] Основы Python — кратко. Часть 3. Списки, кортежи, файлы. [Электронный ресурс] // Thematic Media. — URL : http://habrahabr.ru/post/30092/ (дата обращения: 22.05.2014).

[5] Саммерфилд М. Программирование на Python 3 : пер. с англ. / М. Саммерфилд. — Спб. : Символ Плюс, 2009. — 608 с.

[6] Головатый А. Django. Подробное руководство : пер. с англ. / А. Головатый, Джейкоб Каплан-Мосс. — 2-е изд. — Спб. : Символ Плюс, 2010. — 560 с.

[7] Форсье Джефф. Django. Разработка веб приложений на Python : пер. с англ. / Джефф Форсье, Пол Биссекс, Уэсли Чан. — Спб. : Символ Плюс, 2009. — 456 с.

[8] CSS [Электронный ресурс] // Wikimedia Foundation, Inc. — URL : http://ru.wikipedia.org/wiki/Каскадные таблицы стилей (дата обращения: 23.05.2014).

[9] Знакомство с языком JavaScript : Часть 1. Основы [Электронный ресурс] // IBM. — URL : http://www.ibm.com/developerworks/ru/library/wa-javascriptstart/ (дата обращения: 24.05.2014).

[10] Кантор Илья. Справочник javascript: window [Электронный ресурс] // Илья Кантор. — URL : http://javascript.ru/window (дата обращения: 24.05.2014).

[11] Кантор Илья. Введение. DOM в примерах [Электронный ресурс] // Илья Кантор. — URL : http://javascript.ru/tutorial/dom/intro (дата обращения: 24.05.2014).

ГЛОССАРИЙ

Ajax — технология, позволяющая JavaScript без перезагрузки страницы отправлять данные серверу, получать и обрабатывать их.

API — совокупность всех возможных действий, которые может выполнить программист с какой-либо внешней программой. Например, jQuery предлагает программисту набор методов, вызов которых будет соответствующим образом влиять на DOM. Все эти методы в сумме и являются API.

CSS — каскадные таблицы стилей, которые определяют для HTML-документов визуальное отображение.

Django — фреймворк, созданный на языке Python, который предназначен для создания веб-приложения. Имеет собственную систему ORM для работы с базами данных и собственные встроенные систему шаблонов и веб-сервер.

DOM — API, позволяющее программам работать с элементами HTML-подобных разметок. Представляет из себя иерархическое дерево узлов, каждый из которых представляет элемент, атрибут, текстовый, графический или иной объект.

GET-метод запроса — специальный формат передачи данных по протоколу HTTP, который чаще всего используется только, когда пользователю надо получить информацию от сервера.

HTML — стандартизированный язык разметки, который позволяет в едином формате хранить, передавать и изменять информацию в текстовом виде.

HTTP — протокол передачи данных, который стандартизирует информацию через специальный формат.

Javascript — прототипно-ориентированный язык с нестрогой динамической типизацией, чаще всего используемый для создания клиентских сценариев, которые запускаются браузерами пользователей.

jQuery — библиотека, изначально предназначенная для облегчения манипулированием элементами DOM, однако позже в jQuery было включено довольно много сторонних вспомогательных функций. Jquery позволяет создавать простейшие эффекты для элементов DOM, создавать, редактировать и удалять элементы разметки, а также многое другое.

MVC — шаблон программирования, который призывает разделять логику любой программы на три как можно более независимых компонента, первый из который называется «модель» и отвечает за работу с данными, второй называется «представление» и отвечает за формирование внешнего вида программы, а третий, именуемый «контроллер», реагирует на действия пользователя, вызывая нужные методы моделей и представлений.

POST-метод запроса — специальный формат передачи данных по протоколу HTTP, который чаще всего используется, когда пользователю надо не только получить информацию от сервера, но ещё и обработать там какие-либо пользовательские данные.

Python — объектно-ориентированный язык со строгой динамической типизацией, в веб-программировании используется на серверном оборудовании для обработки поступающих от клиентов запросов.

URI — унифицированный идентификатор ресурса, который позволяет в едином формате определять уникальное имя для какого-либо файлового или абстрактного ресурса.

URL — подмножество URI, который не только описывает идентификатор ресурса, но и указывает путь (в реально существующей файловой системе или абстрактный) до этого ресурса. Начинается с символа слеша «/», который означает корневую директорию для хранилища, где расположен ресурс.

Библиотека — сборник подпрограмм, которые являются типичными для какой-либо области разработки. Например, может существовать библиотека, определяющая функции для работы с файлами в определенном формате, чтобы программисту самому не было нужды изучать тонкости работы данного типа файлов.

Клиентская программа — часть клиент-серверной программы, которая устанавливается на машине пользователя, выводит на экран внешний вид сервиса, обрабатывает события пользователя, отсылает запросы серверу. В веб-приложениях роль клиентской программы выполняет браузер.

Контекст шаблона — совокупность всех данных, которые в текущей ситуации должны быть отображены в шаблоне. Например, именно контекст должен определять, какое именно имя должно отображаться в конечном HTML-документе, сформированном из шаблона.

Контроллер — часть программы, реагирующая на пожелания пользователя, будь то нажатие какой-либо кнопки, ввод текста или просто любой запрос от клиентской программы. Именно контроллер должен определять, какое представление нужно возвращать пользователю и, нередко, какие методы модели для получения данных нужно вызвать.

Модель — часть программы, которая ответственна за работу с данными: хранение данных, их создание, проверка на корректность, модификация, удаление и так далее. Нередко также слово «модель» применяется к какой-либо определенной переменной или классу переменных, определяющих интерфейс для работы с данными (например, класс User — это модель пользователя, а класс Book — модель, соответственно, книги).

Представление — часть программы, ответственная за отображение пользовательского интерфейса. В представлениях определяется, какая информация должна отображаться при определенном запросе пользователя и как эта информация должна внешне выглядеть.

Рендеринг — процесс замены всех специально созданных «пустот» в шаблонах теми значениями, что переданы контекстом. Чаще всего — это просто поиск специально отмеченных имен переменных в шаблонах и замена имен переменных на их значения из контекста.

Селектор — правило, которое определяет, какие элементы следует найти в DOM. С помощью селекторов можно фильтровать элементы не только по собственным атрибутам, но и по свойствам родственных элементов (например, найти только те элементы, у которых один из родителей обладает атрибутом class, равным «*someclass*»).

Серверная программа — часть клиент-серверной программы, которая установлена на удаленном от пользователя компьютере, обрабатывает, чаще всего, запросы сразу от многих клиентских программ, выполняет вычисления, хранит информацию в базах данных, формирует и отсылает ответы.

Статичные файлы — файлы, которые не подлежат программной обработке и какой-либо модификации перед отправкой их непосредственно пользователю. Если какой-либо HTML-файл не нужно «рендерить» перед отправкой, то это статичный файл, а если перед отправкой происходит-таки процесс рендеринга, что означает как раз модификацию файла, значит, это уже не статичный файл.

Форма в HTML — группа полей, которые позволяют пользователю вводить информацию внутри HTML-элемента для последующей программной обработки этой информации. Именно в форме должны задаваться общие для всех полей параметры, например по какому адресу и каким методом отправлять данные на сервер.

Фреймворк — программный продукт, предоставляющий каркас для каких-либо типичных задач, решаемых на определенном языке программирования. В отличие от библиотек фреймворк задает файловую структуру и определяет основные черты архитектуры будущего проекта.

Функция-представление Django — функция, которая вызывается на запросы пользователей, принимает в качестве аргумента объект запроса и возвращает стандартизированный к HTTP-объект ответа, отправляющийся клиентскому приложению пользователя.

Шаблоны — текст, в котором, помимо какой-либо заранее определенной информации, расположены специальные места для внесения динамической информации, которая зависит от контекста ситуации (например, имя пользователя в конечном HTML-документе зависит от того, под каким аккаунтом авторизован текущий пользователь).

Учебное издание

Титков Антон Вячеславович
Черепанов Сергей Андреевич

СОЗДАНИЕ ВЕБ-ПРИЛОЖЕНИЙ

Учебное пособие

Корректор Осипова Е. А.
Компьютерная верстка Насынова Н. Е.

Подписано в печать 18.08.14. Формат 60x84/8.
Усл. печ. л. 8,4. Тираж 100 экз. Заказ

Издано в ООО «Эль Контент»
634029, г. Томск, ул. Кузнецова д. 11 оф. 17
Отпечатано в Томском государственном университете
систем управления и радиоэлектроники.
634050, г. Томск, пр. Ленина, 40
Тел. (3822) 533018.