Е. Г. Сысолетин, С. Д. Ростунцев

# РАЗРАБОТКА ИНТЕРНЕТ-ПРИЛОЖЕНИЙ

УЧЕБНОЕ ПОСОБИЕ ДЛЯ СПО

*Рекомендовано Учебно-методическим отделом среднего профессионального образования в качестве учебного пособия для студентов образовательных учреждений среднего профессионального образования*

**Книга доступна на образовательной платформе «Юрайт» urait.ru, а также в мобильном приложении «Юрайт.Библиотека»**

**Москва • Юрайт • 2021**

УДК 004.4(075.32)
ББК 32.973я723
С95

***Авторы:***

**Сысолетин Евгений Геннадьевич** — старший преподаватель кафедры информационных технологий Института радиоэлектроники и информационных технологий Уральского федерального университета имени первого Президента России Б. Н. Ельцина;

**Ростунцев Савва Дмитриевич** — ассистент кафедры информационных технологий Института радиоэлектроники и информационных технологий Уральского федерального университета имени первого Президента России Б. Н. Ельцина.

***Рецензенты:***

*Папуловская Н.В.* — кандидат педагогических наук, доцент кафедры электрооборудования и энергоснабжения Российского государственного профессионально-педагогического университета;

*Филимонов А. Ю.* — начальник информационно-аналитического отдела Екатеринбургской городской Думы.

**Сысолетин, Е. Г.**

С95 Разработка интернет-приложений : учебное пособие для среднего профессионального образования / Е. Г. Сысолетин, С. Д. Ростунцев. — Москва : Издательство Юрайт, 2021. — 90 с. — (Профессиональное образование). — Текст : непосредственный.

ISBN 978-5-534-10015-0

В учебном пособии раскрывается тема основ проектирования интернет-приложения. В первой части пособия содержатся теоретические положения: вводная информация об интернете и способах взаимодействия с ней; технологии построения интернет-приложений; особенности создания клиентской и серверной части приложения; особенности проектирования интернет-приложений.

Во второй части пособия приводятся методические указания выполнения практических заданий. Практическая часть состоит из пяти лабораторных работ, которые содержат задание и вспомогательную информацию для выполнения заданий.

Соответствует актуальным требованиям Федерального государственного образовательного стандарта среднего профессионального образования и профессиональным требованиям.

*Для студентов образовательных учреждений среднего профессионального образования, обучающихся по инженерно-техническим специальностям.*

УДК 004.4(075.32)
ББК 32.973я723



ISBN 978-5-534-10015-0

# Содержание

# Введение

Учебно-методическое пособие посвящено основам проектирования интернет-приложений. Разработано в рамках дисциплины «Проектирование интернет-приложений» для студентов специальности «Информатика и вычислительная техника».

Цель курса:

— ознакомление студентов с основными технологиями, необходимыми для создания интернет-приложений;
— получение практических навыков на основе выполнения лабораторных работ.

В настоящее время интернет является неотъемлемой частью жизни людей, без которой существование уже и не представляется возможным. Но для взаимодействия через интернет нужны приложения, позволяющие это осуществить, то есть интернет-приложения. Интернет-приложения можно встретить на любом современном устройстве: телефоне, планшете, компьютере. Чаще всего это приложения: браузеры, чаты, «облачные хранилища», игры — таких приложений безграничное множество. Но у всех этих приложений есть и другая сторона — серверная, которая и выполняет функцию хранения и обработки информации. Другими словами, интернет-приложения повсеместно применяются и для приложений уровня рабочей группы, уровня предприятия и т. д. Тому есть несколько причин.

— Отсутствие фазы развертывания приложения. Предположим, у нас есть 100 потенциальных участников некой информационной системы. В случае с *классическим* интерфейсом нам нужно было бы установить клиента на 100 рабочих мест. В случае с интернет-приложением достаточно развернуть его в одной точке — на сервере. Клиенты при этом получают возможность работы с приложением, просто набрав в браузере определенный адрес.

— Кросс-платформенность. Из указанных 100 человек некоторые вполне могли оказаться дизайнерами, для которых *компьютер* — это только название бренда, например, Apple, о существовании PC они могут и вообще не знать. В случае *классического* оформления приложения пришлось бы иметь две версии: одну для PC/Windows, другую — для Apple/Mac OS X.

— Отсутствие версионности. Web-организация взаимодействия с пользователями допускает просто замену приложения на сервере, после чего следующий пришедший запрос будет обработан уже новой версией приложения. Учитывая, что HTTP как протокол не поддерживает состояния сессии, описанное выше решение по замене версий вполне работоспособно и очень удобно в эксплуатации. Представим себе следующую ситуацию: на крупном предприятии установлена учетная система (ERP, Enterprise Resource Planning). Предприятие существует не само по себе, а в рамках государства и действующего законодательства, и обязано выполнять требование последнего. При достаточно кардинальном изменении — налогообложения, порядка расчета себестоимости или чего-нибудь еще — у разработчиков остается только один сценарий: выпуск новой версии программного обеспечения, и она должна быть установлена у всех работающих с данной системой сотрудников. Подобная ситуация случается на практике достаточно редко, тем не менее вызывает чуть ли не временный паралич работы предприятия: смена серверного программного обеспечения; необходимость настройки каждого клиентского приложения.

— Отсутствие хранения каких-либо данных на клиентской части. В ходе своей работы с системой пользователь может взаимодействовать с информацией разного рода: с документами, таблицами, графиками и т. д. Но вся информация обычно и хранится, и обрабатывается на серверной стороне. Из этого факта вытекают два следствия:

- пользователь не обязательно должен работать с одного и того же рабочего места. В *классическом* варианте, если пользователь пересаживается за соседний компьютер, то для него меняется все: становятся другими документы, таблицы и т. д. Более того, цвета и внешний вид окон так же могут оказаться разными. В случае с интернет-приложением пользователь получает доступ к тем же самым данным и с другого рабочего места, и с другого устройства. Например, с телефона или планшета;

- в случае поломки компьютера пользователя вопрос восстановления занимает минимум времени, никакие данные при этом не будут потеряны. Конечно же, сервер — это тоже компьютер и он тоже может ломаться. Но обеспечить (и аппаратно, и программно) резервирование на одном сервере значительно проще и дешевле, чем сделать то же самое на ста рабочих станциях. Стоимость рабочей станции при таком подходе уменьшается, на ней находится минимум необходимого программного обеспечения, необходимо только устройство с операционной системой и браузером, и он сможет продолжать свою работу. Другими словами, разработанное приложение становится значительно доступнее и надежнее.

— Современные средства позволяют разработать пользовательский интерфейс для интернет-приложений, не уступающий *классическому* подходу ни в красоте исполнения, ни в функциональности, ни в эргономике.

Можно также отметить и недостатки интернет-приложений. За указанные выше преимущества приходится платить спецификой разработки: как большим объемом, так и появлением на стадии разработки специфичных для web-фаз дизайна и верстки. При разработке *классического* приложения внутри операционной системы эти две фазы практически полностью отсутствуют: верстка не нужна совсем, а дизайн обычно используется тот, который предоставляется самой операционной системой. Кроме того, в классическом подходе достаточно владеть каким-то одним языком программирования. Например, C++. В случае web-подхода, как правило, одного языка для реализации серверной и клиентской части недостаточно. Более того, кроме языков программирования как таковых при разработке web-приложения используются и специфичные технологии: язык разметки HTML, каскадные стили CSS и т. д.

Есть еще один фактор, который нельзя обойти вниманием при обсуждении web-приложения, — это серверное окружение. Обычно для работы *web-приложения* нужно некое внешнее *обрамление*: есть один или несколько HTTP-серверов, сам контейнер, в котором выполняется приложение, сервер базы данных и т. д.

Таким образом, разработка web-ориентированного приложения выглядит более сложным процессом по сравнению с классическим вариантом, то есть приложением, выполняющимся под управлением операционной системы.

Выбор того или другого подхода зачастую рассматривается с экономической точки зрения. Например, если планируемое приложение имеет десяток возможных потребителей, то, скорее всего, будет принято решение разрабатывать его по *классическому* пути. При большом количестве возможных участников системы снижение эксплуатационных расходов будет настолько существенным, что есть смысл увеличить затрат этапа разработки, то есть разрабатывать приложе-ния с помощью web-технологий.

Таким образом, web-подход к разработке приложений приобретает в последнее время все большую популярность. Цель курса «Про-ектирование интернет-приложений», а также цикла лабораторных работ — познакомиться с общими принципами построения интернет-приложений, на практических примерах показать входящие в них компоненты и взаимодействие этих компонентов между собой.

Для изучения данного материала у читателя уже должны быть сформированы знания в области объектно-ориентированного программирования, без этих знаний понимание читателем курса может быть затруднительным и неэффективным (табл. 1).

Книга разделена на две основных части.

В первой части читатель сможет познакомиться с теоретической частью интернет-приложений, его аспектами и возможными вариантами работы приложений в интернете.

Во второй части предложено пять лабораторных работ с теоретической основой, позволяющей решить предложенные задачи. Все практические задания связаны между собой и являются логическим продолжением друг друга; их необходимо выполнять на объектно-ориентированном языке программирования Java. Также использованы некоторые библиотеки, позволяющие взаимодействовать с базой данных по технологиям ORM (object-relational mapping).

Изучив материалы пособия, студенты освоят: **необходимые знания** технологий создания интернет-проектов; ПО для создания интернет-приложений; **необходимые умения** разработать интернет-сайт или интернет-приложение по заданной тематике; **трудовые действия** владения навыком создания сайтов и веб-приложений по заданной тематике.

## Критерий оценок

*Таблица 1*

**Критерий оценок по дисциплине «Проектирование интернет-приложений»**

| Тип занятия | Контрольное мероприятие | Критерии оценки | Количество баллов |
|---|---|---|---|
| Лекционные занятия | Посещение лекций (18) | Присутствие на всем протяжении лекции | 1 посещение лекционного занятия = 1 балл |
| | Домашняя работа № 1 (Разработка технического задания для проектирования интернет-приложения) | Своевременная сдача домашней работы. | 4 |
| | | Логичный ход выполнения задания | 5 |
| | | Студент применил для выполнения задачи, нестандартный метод решения | 4 |
| | | Оформление домашней работы соответствует необходимым требованиям | 3 |
| | Домашняя работа № 2 (Проектирование структуры клиентской части приложения) | Своевременная сдача домашней работы | 4 |
| | | Логичный ход выполнения задания | 5 |
| | | Студент применил для выполнения задачи, нестандартный метод решения | 4 |
| | | Оформление домашней работы соответствует необходимым требованиям | 3 |
| | Домашняя работа № 3 (Проектирование структуры серверной части приложения) | Своевременная сдача домашней работы | 4 |
| | | Логичный ход выполнения задания | 5 |
| | | Студент применил для выполнения задачи, нестандартный метод решения | 4 |
| | | Оформление домашней работы соответствует необходимым требованиям | 3 |
| | Контрольная работа № 1 (Принципы построения клиентской части) | В контрольной работе, даны понятийные ответы на знания дисциплины или присутствует небольшой процент неверных ответов (не больше 60 % от всех вопросов в контрольной работы) | 1–5 |
| | | В контрольной работе, дан четко структурированный ответ без лишних объяснений или присутствуют незначительные неточности в ответах (не более 80 % от всех вопросов в контрольной работы) | 6–10 |

*Продолжение табл. 1*

| Тип занятия | Контрольное мероприятие | Критерии оценки | Количество баллов |
|---|---|---|---|
| Лекционные занятия | | В контрольной работе, дан четко структурированный ответ с дополнительной информацией демонстрирующие отличные знания темы дисциплины. Работа выполнена без ошибок | 11–16 |
| | Контрольная работа № 2 (Принципы построения серверной части) | В контрольной работе, даны понятийные ответы на знания дисциплины или присутствует небольшой процент неверных ответов (не больше 60 % от всех вопросов в контрольной работы) | 1–8 |
| | | В контрольной работе, дан четко структурированный ответ без лишних объяснений или присутствуют незначительные неточности в ответах (не более 80 % от всех вопросов в контрольной работы) | 9–13 |
| | | В контрольной работе, дан четко структурированный ответ с дополнительной информацией демонстрирующие отличные знания темы дисциплины. Работа выполнена без ошибок | 14–18 |
| Лабораторные работы | Посещение лабораторных занятий (18) | Присутствие на всем протяжении лабораторного занятия | 1 посещение лабораторной работы = 1 балл |
| | Выполнение лабораторной работы № 1–4 | Правильно выполненная работа в соответствии с требованиями в методическом материале | 3 |
| | | Самостоятельно выполненная работа (не присутствуют элементы чужой работы) | 3 |
| | | Имеет нестандартный подход к решению задачи, подход который не отражен в методическом материале | 2 |
| | | Правильные ответы на вопросы по выполненной работе | 3 |

*Окончание табл. 1*

| Тип занятия | Контрольное мероприятие | Критерии оценки | Количество баллов |
|---|---|---|---|
| Лабораторные работы | Выполнение лабораторной работы № 5 | Правильно выполненная работа в соответствии с требованиями в методическом материале | 3 |
| | | Самостоятельно выполненная работа (не присутствуют элементы чужой работы) | 4 |
| | | Имеет нестандартный подход к решению задачи, подход который не отражен в методическом материале | 3 |
| | | Правильные ответы на вопросы по выполненной работе | 3 |
| | Вовремя сданная лабораторная работа (5) | Работа сдана в указанный срок. | 1 отчет = 3 балла |
| | Написание отчета по лабораторной работе (5) | Оформление отчета по лабораторным работам в соответствии с методическими указаниями | 1 отчет = 2 балл |

# Часть 1.
# Основы проектирования интернет-приложений

## 1.1. Основные понятия интернет-приложений

В данной главе рассмотрены основные определения, связанные с интернет-приложениями.

### 1.1.1. ИНТЕРНЕТ И ЕГО ОСОБЕННОСТИ

Интернет — Interconnected Networks — объединенные сети, глобальная телекоммуникационная сеть информационных и вычислительных ресурсов. В разгар холодной войны, 4 октября 1957 года, СССР запустил первый искусственный спутник Земли, тем самым получив преимущество в космосе. В США решили, что деньги, отпущенные Пентагоном на научные исследования, тратятся впустую, поэтому было принято решение создать единую научную организацию под покровительством Министерства обороны — ARPA (Advanced Research Projects Agency). Один из разрабатываемых передовых проектов — это распределенная децентрализованная вычислительная сеть, способная пережить даже ядерную войну. Существующие в то время телефонные сети не обеспечивали достаточной надежности: выход из строя одного крупного узла мог разделить сеть на изолированные участки. В декабре 1969 года была создана экспериментальная сеть, построенная на принципах цифровой коммутации пакетов и соединяющая 4 узла:

— калифорнийский университет в Лос-Анджелесе;
— калифорнийский университет в Санта-Барбаре;
— исследовательский университет Стенфорда;
— университет штата Юта.

Это событие считается рождением современного интернета.
Особенности интернета:

— интернет не имеет собственника, является достоянием всего человечества;
— интернет нельзя выключить целиком;
— интернет может связать каждый компьютер с любым другим;
— чтобы из множества подключенных к интернету устройств выбрать нужный, существуют две составляющих части: это адреса и имена участников сети.

#### 1.1.1.1. Адрес в интернете

Для идентификации компьютера в интернете используются адреса. В настоящее время работают одновременно две версии адресации:

— IP v4, представляющая собой четырехбайтное число (32 бита). Для удобства работы цифры адреса разделяют между собой точкой. Например, адрес локальной машины — это 127.0.0.1.
— IP v6, 128 бит. Восемь групп по 4 шестнадцатеричных цифры, разделенных двоеточием. Пример адреса:
2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d

Нули могут заменяться двумя двоеточиями. Тот же адрес локальной машины можно написать как::1. Используется одновременно с IP v4 за счет так называемого «отображения адресов»:: FFFF: xx.xx.xx.xx дает IP v4 адрес, то есть младшие 32 бита при этом равны ip v4 адресу. Такой подход дает возможность с сетей IP v6 обращаться к тем хостам, которые поддерживают только IP v4.

В обоих стандартах существуют особые группы адресов. Сюда относятся так называемые *не маршрутизируемые* адреса, предназначенные для организации локальных сетей и служебного взаимодействия самих маршрутизаторов между собой внутри сетей провайдера — это мультикастовые адреса, предназначенные для широковещательных сообщений, когда одно сообщение передается одновременно нескольким получателям (но не одному и не всем участникам сети); адреса, уникальные только в пределах одного компьютера (локальные адреса).

#### 1.1.1.2. Имя в интернете

Другой важной составляющей интернета является служба имен (DNS, Domain Name System), которая предназначена для представления адресов компьютеров в более *человеческой* универсальной фор-

ме. Служба DNS преобразует полученное имя хоста (строку) в IP-адрес устройства. Ключевым элементом отсчета является точка («.» так и называется — *домен точка*, или *корневой домен*, или *домен нулевого уровня*). Домен точка разделен на зоны (.com.,.org.,.net.,.ru. и так далее). В связи с тем, что *домен точка* присутствует в адресе всегда, он опускается в записи адреса (последняя точка не указывается, вместо urfu.ru. пишут urfu.ru).

Существует несколько утилит, позволяющих *вручную* провести распознавание адреса. Направлены эти утилиты прежде всего на то, чтобы диагностировать работоспособность службы DNS на данном участке сети. В частности, можно отметить команды host и nslookup:

```
sally ~ # host urfu.ru
urfu.ru has address 93.88.190.5
urfu.ru mail is handled by 100 relay1.urfu.ru.
urfu.ru mail is handled by 200 relay2.urfu.ru.
```

У DNS существует как прямое направление распознавания (по имени найти IP-адрес), так и обратное (по адресу найти имя). В общем случае полученные значения могут не совпадать между собой, потому что один физический IP-адрес может иметь несколько имен. В приведенном выше примере адрес urfu.ru и его IP-адрес не совпадают между собой в прямом и обратном направлении:

```
sally ~ # host 93.88.190.5
5.190.88.93.in-addr.arpa domain name pointer ustu.ru.
```

Для идентификации ресурса в интернете существуют определенные стандарты. Чтобы получить конкретный ресурс (документ, изображение, почтовое сообщение, вызываемую процедуру и так далее), кроме собственно адреса хоста, на котором этот ресурс расположен, нужно указать еще ряд параметров. В общем случае строка идентификации ресурса согласно спецификации URL (Uniform Resource Locator) выглядит следующим образом:

```
имя_службы://имя_хоста.имя_домена.зона: порт/имя_ресурса
```

Кроме того, указанная строка при необходимости может быть дополнена именем входа/паролем, а также дополнительными произвольными параметрами, передаваемыми в запросе ресурса (например, параметрами для вызова RPC-процедуры).

Вот как выглядит адрес ресурса, предоставляющего сведения о погоде в Екатеринбурге:

```
http://www.gismeteo.ru/city/hourly/4517/#wdaily1
```

Забегая вперед, отметим, что довольно часто при разработке интернет-приложений появляется необходимость обмана службы DNS с путем присвоения некоторым соседним компьютерам фиктивных имен, на самом деле DNS не распознаваемых. Либо второй вариант — присвоить одному компьютеру несколько имен. Делается это при помощи файла hosts. В MS Windows файл hosts расположен в каталоге c:\Windows\System32\drivers\etc\, в *nix системах — в каталоге/etc.

Например, мы хотим одновременно разрабатывать на одном компьютере два web-приложения. Одно из них представляет собой библиотеку книг с указанием автора и краткой аннотацией и называется library. Другое из них посвящено учету персональных финансов и называется purse. Когда приложения будут готовы, в файлы DNS будут внесены соответствующие изменения, и имена станут доступны всем пользователям интернета, но на этапе разработки это не обязательно. Достаточно внести следующую строчку в упомянутый файл hosts:

127.0.0.1 localhost purse library

Компьютер, за клавиатурой которого непосредственно производится разработка обоих указанных приложений (то есть «локальный» компьютер) будет откликаться на два этих имени.

Такой подход (обман) не позволит подключиться любому пользователю интернета к вашим приложениям. Но он дает возможность организовать на одном компьютере два независимых виртуальных хоста и обращаться к ним из строки браузера, набирая в ней (на локальном компьютере): http://purse или http://library и получая ответ от разрабатываемых приложений.

#### 1.1.1.3. Службы (сервисы)

Службы (сервисы) — это информационные системы, разработанные для предоставления конкретных информационных услуг в сети интернет. Они включают в себя наборы программ и протоколов прикладного уровня, обеспечивающих пользователей сети возможностью выполнять работу с распределенными информационными ресурсами. Каждый сервис базируется на своем протоколе (или семействе протоколов), который позволяет клиентам и сервисам общаться между собой. Можно привести несколько примеров сервисов, базирующихся на соответствующих протоколах:

— служба DNS (BIND, 53 порт). Базис интернета. Неработоспособность службы DNS приведет к неработоспособности всех

остальных служб, поэтому существует ряд технических и организационных решений, направленных на обеспечение круглосуточной и бесперебойной работы этой службы в сети;
- служба электронной почты (SMTP, POP-3, IMAP);
- обмен моментальными сообщениями (ICQ, IRC, Skype);
- передача файлов между компьютерами (FTP, SFTP);
- управление удаленными серверами, выполнение на них команд удаленным способом, копирование файлов (SSH, RSH, FISH);
- управление сетью (SNMP);
- служба доступа к каталогам справочной информации (LDAP).

Наиболее известным сервисом в интернете является web-сервис (WWW, World Wide Web, «Всемирная паутина»), предоставляющий пользователям доступ к документам с использованием протокола HTTP. Очень часто между понятиями *интернет* и *web* ошибочно ставят знак равенства. Между тем, протокол HTTP, лежащий в основе службы web, безусловно, самый распространенный, но все же один из многих протоколов передачи данных, объединенных между собой названием *интернет*.

#### 1.1.1.4. Сокета

Сокета представляет собой абстракцию, программный объект внутри операционной системы, с помощью которого происходит соединение клиентской и серверной части. При этом клиент и сервер могут находиться как внутри одного и того же компьютера, так и быть на разных компьютерах.

Представим себе такую ситуацию: вам звонят по телефону. При этом и вы, и звонящий вам абонент берете в руки телефонные трубки. Вас не интересуют подробности работы телефонной сети. Вы не знаете, какими базовыми станциями обслуживается ваш разговор, по каким каналам связи идет сигнал, где и какое сработало коммутационное оборудование и так далее. Зато вы знаете, что если набрать номер, через какое-то время установится соединение и можно будет говорить. Причем делать это нужно по очереди. То есть для общения вы должны соблюдать некоторый протокол, последовательно обращаясь к некоторым функциям трубки (набрать номер, дождаться ответа абонента, говорить). Можно сказать, что телефонная сеть для вас представлена трубкой. То же самое справедливо и для сокеты: сразу за ней начина-

ется сеть, но подробности работы самой этой сети приложению знать не обязательно, достаточно вызывать некоторые функции объекта *сокеты* в определенном порядке (то есть с соблюдением протокола).

Сокета — объект универсальный, единый по своей сути для клиентской и серверной стороны. После ее создания при помощи вызова соответствующих методов сокета может стать как клиентской, так и серверной. Если воспользоваться аллегорией, то серверная сокета — это электрическая *розетка* в стене. У этой розетки есть замечательное свойство: она всегда является свободной. Как только к ней подключается клиент (*вилка*), сервер сразу же порождает новый процесс, обслуживающий данное соединение. Розетка с включенной в нее вилкой *отходит* в сторону, где происходит непосредственно обмен данным между конкретным клиентом и сервером. А исходная *розетка* по-прежнему остается свободной, доступной для следующих соединений. Каждый подключившийся клиент занимает определенные ресурсы операционной системы, поэтому при создании серверной сокеты указывается максимально допустимое число клиентских соединений.

Для работы с созданной сокетой точно так же, как и с любым другим устройством операционная система использует дескриптор — беззнаковое целое число, идентифицирующее ресурс внутри операционной системы. После получения дескриптора в сокету можно писать данные, читать из нее, назначать на нее некие обработчики событий (сигналов) и так далее. Существует великое множество библиотек для работы с сокетами, которые предоставляют реализацию функций либо вообще готовых классов клиентской и серверной сокеты. Эти библиотеки представляют собой переход на более высокий уровень работы, предназначены для облегчения труда программистов. Действительно, особого смысла каждый раз реализовывать одни и те же низкоуровневые процедуры нет. Однако на нижнем уровне обращение с сокетой в любой из библиотек все равно происходит с использованием именно дескриптора.

Сокета как программный объект характеризуется тремя параметрами: доменом (*областью*, семейством протоколов, которые могут быть использованы для данной сокеты), типом и портом.

Наиболее распространенными являются следующие домены:

— AF_UNIX (AF_LOCAL) — для организации обмена в рамках одного и того же компьютера;

— AF_INET, AF_INET6 — по протоколу IP v4/IP v6 соответственно;
— AF_NETLINK — для взаимодействия между пространством пользовательских программ и ядром операционной системы.

Среди типов сокет наиболее распространены три:

— SOCK_STREAM — создаваемая сокета, предназначена для организации последовательного потока. При этом между клиентом и сервером устанавливается виртуальное соединение, ошибки в работе которого также обрабатываются сокетой. Соответственно, если мы создадим сокету с доменом AF_INET и типом SOCK_STREAM, такая сокета будет обрабатывать TCP/IP соединение.
— SOCK_DGRAM — сокета, предназначена для передачи дейтаграмм. Дейтаграммы (датаграммы, datagram) — это пакеты информации, которые передаются по сети без установления соединения. Они имеют адрес отправителя и получателя, но факт их доставки не гарантируется. Точно так же не гарантируется и порядок прихода дейтаграмм: далеко не факт, что получатель примет сообщения в том же самом порядке, в каком они были переданы. Основной особенностью работы с дейтаграммами является отсутствие временных задержек, вызванных скоростью работы сети. Являются основой протокола UDP (User Datagram Protocol).
— SOCK_RAW — так называемая *сырая* сокета. Данный тип сокеты используется для низкоуровневого программирования, а также при необходимости дополнения существующих протоколов (например, добавления в сообщение собственных заголовков).

Порт сокеты — это беззнаковое целое двухбайтное число, которое определяет службу сервера. Сочетание порт — протокол является уникальным на данном компьютере в том смысле, что если данный порт и данный протокол уже прослушиваются каким-либо процессом, другой процесс не сможет повторно создать такую же сокету. Тот же порт, но с другим протоколом при этом захватить можно. Порты диапазона 0–1023 являются *привилегированными*, то есть для создания сокеты, прослушивающей данный порт, необходимы особые привилегии. Теоретически любой порт свыше 1024 может быть захвачен первым запросившим его процессом. На самом деле число офици-

ально зарегистрированных портов гораздо больше, чем 1024. В качестве примера можно назвать сочетание 3306/tcp, которое используется сервером MySQL по умолчанию. Общепринятые присвоения портов не являются официальным стандартом, и любой созданный пользователем процесс имеет полное право захватить порт 3306/tcp. И иногда программисты намеренно этим пользуются с целью ввести в заблуждение возможных злоумышленников. Однако, если у вас нет на то каких-то особых причин, не стоит использовать в своих программах зарегистрированные пары порт — служба.

После создания сокеты (иными словами, получения от операционной системы дескриптора сокеты) на нее могут быть установлены модификаторы — такие же, как и для любого символьного устройства в операционной системе. Например, сокету можно сделать неблокирующей. При этом, если из сокеты на данный момент нечего читать, то оператор чтения сокеты вернет нулевое количество прочитанных байт (а не приостановит работу программы, как это произойдет в случае с блокирующей сокетой).

#### 1.1.1.5. Протокол HTTP. Виды запросов

HTTP — Hyper Text Transfer Protocol — протокол передачи гипертекста, то есть такого текста, когда вместе с содержимым передается информация по его отображению. Изначально HTTP предназначался для передачи HTML-содержимого. Однако в настоящее время протокол HTTP используется для передачи любой информации произвольного содержания, оставаясь при этом текстовым. Термин текстовый означает, что в ходе работы могут передаваться только отображаемые на экране символы. Например, в принимаемой и передаваемой информации не может быть символа «0». Разумеется, что понятие любая информация подразумевает, что каждый передаваемый байт может принимать все допустимые значения, то есть 0–255 включительно. Такая информация еще называется *бинарной*. Передача бинарной информации с помощью текстового протокола осуществляется за счет специального кодирования, когда исходная информация на момент передачи видоизменяется таким образом, что содержит только текстовые символы. На клиенте после получения сообщения осуществляется обратное преобразование. С помощью такого механизма посредством HTTP могут передаваться, например, двумерные или трехмерные изображения.

Кроме того, HTTP широко используется как транспортный протокол для передачи между разными программами объектов (экземпляров классов). Информация о структуре объекта (именах полей, их модификаторах, значениях и т. д.) преобразуется в текстовый вид (например, в XML или JSON). В текстовом виде передается по сети и на *клиенте* осуществляется обратное преобразование из текста в программный объект. Одним из существенных преимуществ подобного метода передачи является то обстоятельство, что клиент и сервер могут быть написаны на разных языках программирования, выполняться на различных архитектурах и т. д. Таким образом, с помощью протокола HTTP связываются между собой гетерогенные приложения в логически единое целое (рис. 1.1).

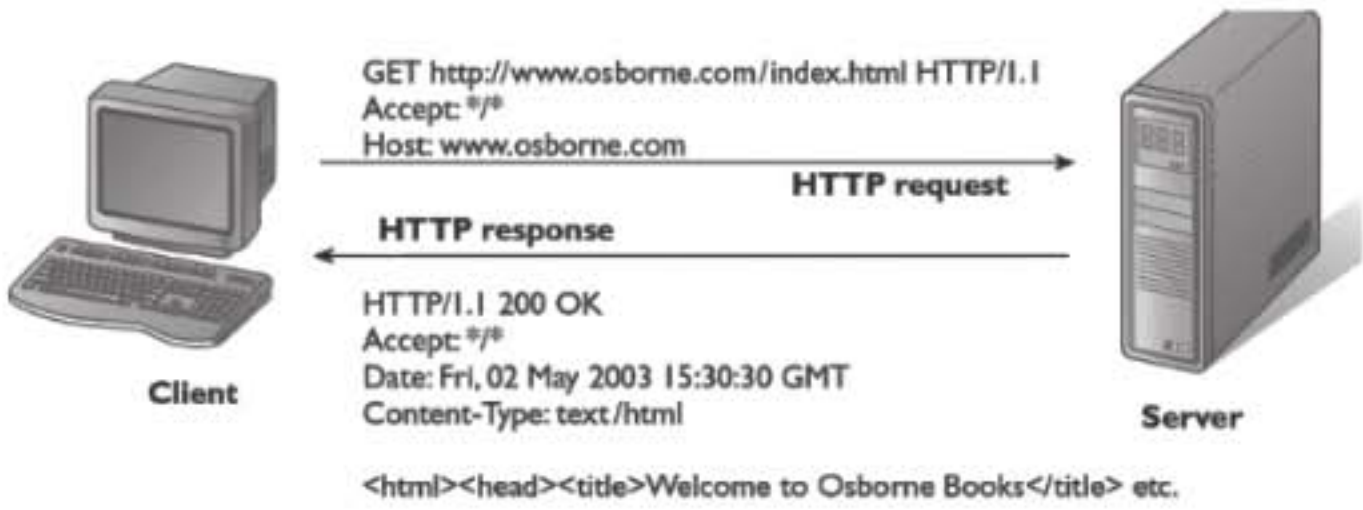


Рис. 1.1. Пример общения клиента и сервера по HTTP-протоколу

Основой HTTP является технология *клиент-сервер*, то есть всегда выделяются два участника обмена: клиент, который делает запрос (HTTP Request), и сервер, который отвечает клиенту на данный запрос (передавая по сети пакет HTTP Response). На этом один *шаг* протокола заканчивается, и данный цикл может повторяться сколь угодно большое число раз. Особенностью HTTP является тот факт, что сам по себе протокол не поддерживает информацию о сессии, не обязан что-либо знать об истории. Иными словами, протокол не предусматривает сохранения состояния. Есть запрос, и есть ответ на него. Причем, как правило, сразу после ответа сервер разорвет TCP-соединение с клиентом с целью более рачительного использования собственных ресурсов. При необходимости клиенты, использующие HTTP, могут самостоятельно сохранять информацию об истории данного сеанса. Для этих целей существуют два механизма:

— *куки* — cookies — небольшие пакеты данных, хранящиеся на стороне клиента и передающиеся вместе с запросом. Как правило, cookie имеет два атрибута, определяющие его сохранение клиентом, то есть возможность сохранения информации на диске. В случае, если cookie разрешается хранить информацию у клиента на диске, указывается *время жизни* данного cookie на клиентской стороне, после чего он будет считаться устаревшим, не актуальным. В частности, в качестве cookie может выступать уникальный идентификатор сессии на стороне сервера;
— *сессии* — HttpSessions — на стороне сервера. Сессия представляет собой программно-доступный объект, который формируется сервером и содержит в себе информацию о работе с данным клиентом. Объект сессии уникален для сеанса работы с каждым клиентом. Обычно сессия имеет идентификатор, который сохраняется в cookies на клиенте. Механизм сессий скрывает подробности реализации, разработчику нет необходимости отслеживать, что из информации запоминается на клиентской части, а что — на серверной. Достаточно просто получить объект *сессии* и думать о нем, что он однозначно определяет сеанс данного клиента.

Реализация сервера может хранить информацию о заголовках последних запросов, поступивших с данного IP-адреса. Сам протокол не осведомлен о предыдущих запросах и ответах, внутренняя поддержка состояния в нем не предусмотрена. Текущей версией протокола HTTP/1.1 предусмотрен режим постоянного соединения, то есть установленное tcp/ip соединение *может* оставаться открытым и после отправки ответа на поступивший запрос. Ключевое слово здесь — может, при разработке собственных интернет-приложений правильнее будет полагать, что HTTP работает как одна-единственная изолированная от контекста пара *запрос-ответ*.

Вне зависимости от вида запрос это или ответ пакеты в HTTP имеют одинаковую структуру и состоят из следующих параметров:

— стартовой строки или строки запроса/ответа. Содержание стартовой строки однозначно идентифицирует тип сообщения (тип пакета). Содержание стартовой строки отличается для запросов и ответов;
— заголовков сообщения. Разделенные двоеточием пары имя: значение. Характеризуют тело сообщения (какой сервер использу-

ется; когда последний раз менялся объект; что конкретно будет передаваться; в какой форме кодироваться и так далее). Существование заголовков в том числе позволяет передавать с помощью HTTP бинарную (двоичную, то есть *любую*, не обязательно текстовую) информацию, хотя сам протокол является текстовым. На использовании заголовков базируются и расширения HTTP: разработчик серверного программного обеспечения вправе добавить свои собственные заголовки. При этом сохраняется совместимость с существующими клиентами, которые будут просто игнорировать незнакомые заголовки.

— тела сообщения. Тела сообщения может и не быть в запросе в том случае, если стартовая строка и заголовки однозначно идентифицируют запрашиваемый ресурс.

Заголовки от тела отделяются пустой строкой. То есть такой строкой, которая не содержит других символов, кроме возврата каретки и перевода строки, <CR><LF>, «\r\n», коды символов — 13 и 10 в десятичной системе. Довольно часто встречаются решения, ограничивающее данное правило только до одного символа — возврата каретки («\r»).

#### 1.1.1.5.1. Структура запроса (Request)

Стартовая строка запроса выглядит следующим образом:

Метод URI HTTP/Версия

*Метод* — это название запроса, одно слово заглавными буквами. Наибольшее распространение имеют следующие методы:

*GET* — запросить содержимое указанного ресурса. Клиент может передавать параметры указанному ресурсу, перечисляя их после символа «?». Пример стартовой строки метода GET с передачей параметров на сервер:

GET/some_resource?param1=value&param2=value2 HTTP/1.1

Метод GET является идемпотентным, то есть многократный запрос GET с одними и теми же параметрами должен приводить к одним и тем же результатам.

*HEAD* — то же самое, что и GET, но само содержимое ресурса при этом сервером не передается, передаются только заголовки. Метод позволяет узнать, существует ли запрашиваемый ресурс на сервере. Если имеется — не менялось ли его содержимое со времени последнего запроса.

*POST* — применяется для передачи пользовательских данных (параметров) указанному ресурсу. При этом сами передаваемые параметры включаются в тело запроса. При помощи метода POST можно создать ресурс (например, загрузить файл на сервер). В этом случае сервер выдаст ответ 210 (Created) и в заголовке Location будет указан URI созданного ресурса. Метод POST идемпотентным не является, то есть многократное повторение POST с теми же параметрами может приводить к разным результатам.

*PUT* — загрузка содержимого запроса на указанный ресурс.

*DELETE* — удалить указанный ресурс.

*URI* (Uniform Resource Identifier) — путь к запрашиваемому ресурсу (документ, изображение, файл, службу, ящик электронной почты и т. д.).

*Версия* — пара разделенной точкой цифр.

#### 1.1.1.5.2. Структура ответа (Response)

Стартовая строка ответа сервера имеет следующий формат:

HTTP/*Версия КодСостояния Пояснение*

*Версия* — две цифры, разделенные точкой.

*КодСостояния* — три цифры, значение которых определяет результат выполнения запроса и дальнейшее поведение клиента. Первая цифра кода состояния определяет его класс, группу, к которой принадлежит данный код состояния. В настоящее время выделяют следующие группы:

- 1xx — информационное сообщение;
- 2xx — успешно выполненный сервером запрос;
- 3xx — перенаправление. Сообщает клиенту, что нужно сделать еще один запрос, как правило, по другому URI. Адрес, по которому клиент должен сделать запрос, как правило, указывается в заголовке Location;
- 4xx — были допущены ошибки со стороны клиента;
- 5xx — возникли ошибки на сервере.

При возникновении ошибок, как правило, тело сообщения содержит гипертекстовую информацию, поясняющую возникшую ошибку.

Пояснение — текстовое короткое пояснение к коду ответа. Ни на что не влияет и не является обязательным. Большая часть библиотек работы с HTTP имеет собственные средства определения «пояснения» по полученному коду, в том числе и локализованного пояснения.

Пример HTTPresponse:
HTTP/1.1 200 OK
Date: Wed, 11 Feb 2013 11:20:59 GMT
Server: Apache
X-Powered-By: PHP/5.2.4–2ubuntu5wm1
Last-Modified: Wed, 11 Feb 2013 11:20:59 GMT
Content-Language: ru
Content-Type: text/html; charset=utf-8
Content-Length: 1234
Connection: close
*(пустаястрока)*
*(далее следует запрошенная страница в HTML)*

### 1.1.2. ИНТЕРНЕТ-ПРИЛОЖЕНИЯ

Стоит отметить, что термины *интернет-приложения* и *web-приложения* не являются синонимами. Как уже говорилось выше, интернет-приложение не обязано базироваться именно на Web-службе и использовать для межкомпонентного обмена протокол HTTP. Однако, поскольку большую часть интернет-приложений составляют именно такие приложения, разница между терминами практически сходит на нет.

#### 1.1.2.1. Web-приложения

Web-приложение построено, как минимум, по двухуровневой архитектуре (то есть по архитектуре клиент-сервер). При этом в качестве клиентской программы используется web-браузер, а обмен с серверной частью происходит с использованием протоколов HTTP/HTTPS.

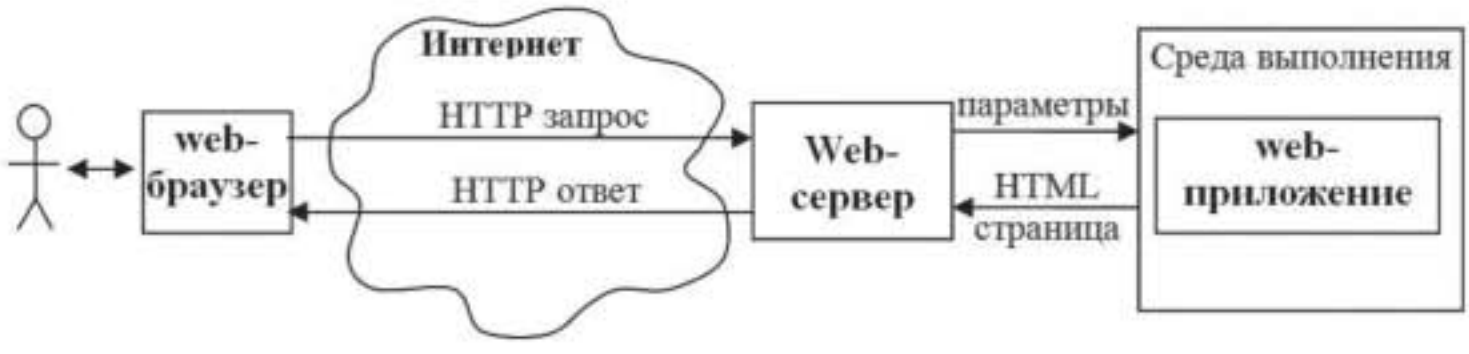


Рис. 1.2. Общая схема взаимодействия пользователя с Web-приложением

Как видно из рис. 1.2, web-сервер (блок, реализующий обмен с клиентом по протоколу HTTP) не является единственной составляющей приложения. Он транслирует методы и их параметры в некую среду, которая программным путем формирует HTML-страницу. Такие страницы называются динамическими, потому что их содержание меняется во времени, может зависеть от параметров, от предыдущих шагов клиента в рамках данной сессии. Среда выполнения может быть различной, более подробно вопрос о способах формирования динамических страниц будет рассмотрен ниже.

Основные причины широкого распространения именно web-приложений обусловлены их достоинствами, а именно:

— *доступностью*, пользователю не нужно что-либо ставить на компьютер в качестве клиентского программного обеспечения, достаточно просто набрать нужный адрес в браузере;
— *отсутствием версионности*, если разработчик изменил код приложения, клиентам не нужно сообщать, доступна ли новая версия программы. Следующий после изменения кода запрос даст пользователю уже новое содержание.

#### 1.1.2.2. Web-сервисы

Web-приложение направлено на работу с пользователем и имеет пользовательский интерфейс. В противоположность этому, web-сервис работает либо с другими web-сервисами, либо с web-приложениями. Обмен при этом происходит точно так же, как и в случае с приложениями, то есть по схеме *запрос-ответ*. В качестве *клиента* может выступать любая программа, которая правильно сформирует HTTP-запрос и расшифрует полученный HTTP-ответ. Однако для общения стандартной версии HTML может оказаться недостаточно, поэтому используются его расширения: JSON, XML-RPC, SOAP, REST и так далее. Общая схема работы web-сервиса представлена на рис. 1.3.

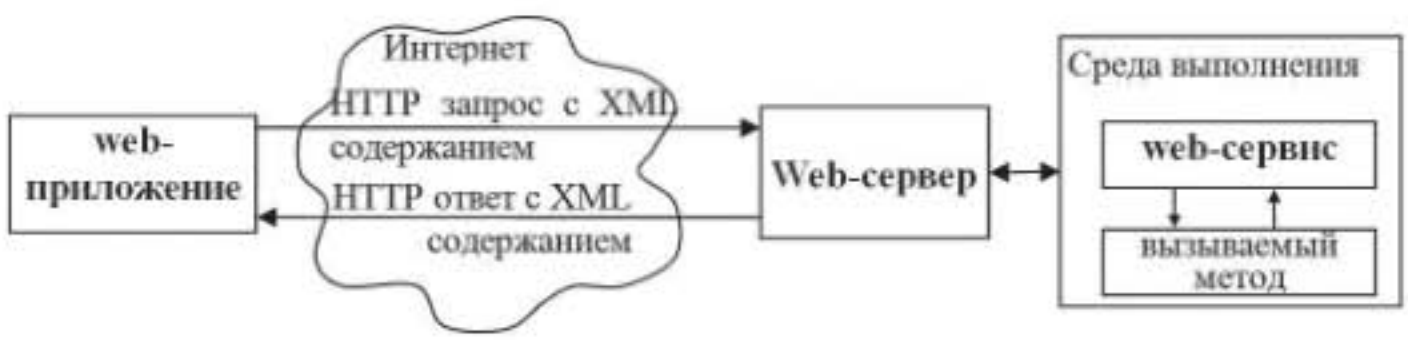


Рис. 1.3 Общая схема взаимодействия пользователя с web-сервисом

Для описания сервиса существует специальный язык, называемый WSDL (Web Service Definition Language). При помощи WSDL можно запросить у web-сервиса сведения о существующих методах и необходимых параметрах, то есть получить полную описательную информацию о предоставляемом сервисе. Кроме собственно информативной составляющей, WSDL несет и другую нагрузку: на его основе строятся различные средства автоматизированного проектирования web-сервисов — программные средства, позволяющие из WSDL файла создавать скелеты классов и наоборот.

В современном интернете существует множество web-сервисов. В качестве примера можно указать Яндекс.XML. Существует всем известная поисковая система Яндекс. Однако для ее использования не обязательно заходить на http://yandex.ru. Вы можете использовать ее в любом созданном приложении (и не обязательно web-ориентированном). Сервис Яндекс.XML позволяет обратиться с запросом к самой поисковой системе, получить результат выполнения этого запроса в виде XML и использовать полученный результат в своем приложении. Разумеется, при этом существует вопрос лицензионности: придется пройти ряд предписанных Яндексом шагов и обязательно указать в своем приложении ссылку, на основе чего сформирован данный ответ. Но суть остается прежней: предлагается некий сервис, данные из которого могут использоваться по своему усмотрению.

#### 1.1.2.3. Особенности проектирования

Интернет-приложения, как правило, доступны всем участникам сети интернет, даже если целью является разработка простой узкоспециализированной информационной системы. Предположим, разрабатывается информационная система предприятия. Одной из поставленных задач будет задача разделения прав пользователей: кто и что сможет видеть внутри этой системы. Независимо от того, что будет происходить в клиентской части, серверная часть, точка входа в разрабатываемую систему, будет расположена в интернете, соответственно, доступна всей многомиллиардной аудитории. В отличие от классических пользовательских программ большую часть интернет-приложений не нужно устанавливать, никто не спросит разрешения на попытку *поработать* с системой.

Соответственно, при проектировании интернет-приложений приходится уделять повышенное внимание вопросам безопасности ра-

боты. При разработке системы безопасности основным принципом является разумная достаточность предпринимаемых мер. Нет никакого смысла проводить сканирование сетчатки глаза, снимать отпечатки пальцев, делать голосовой анализ, вытаскивать из процессора его серийный номер и из сетевой карты ее mac-адрес — и все это только для того, чтобы отобразить пользователю на экране сегодняшние новости.

Другой не менее важной особенностью является непрогнозируемая нагрузка. Сегодня приложение имеет 10 одновременно работающих клиентов в максимуме, а завтра их станет миллион. Соответственно, при разработке приложения ориентируетесь на 10 клиентов, это расчетная нагрузка. Нет никакого смысла сразу же делать приложение из расчета на миллион, потому что нагрузочная способность — это, в конечном счете, деньги и время, что и будет успешно потрачено, а вот миллион клиентов у приложения может и не появиться в силу целого ряда причин. Однако необходимо предусмотреть возможность масштабирования системы таким образом, чтобы удержать лавинообразно нарастающую нагрузку.

Третья рассматриваемая особенность — это круглосуточная работоспособность приложения. Если мы говорим о локальной сети предприятия, то существует нерабочее время для сотрудников этого предприятия, выходные, праздники. Иными словами, у администраторов всегда есть возможность что-то менять, не нарушая работоспособности системы: переставить программное обеспечение, провести сервисные процедуры с базой данных и даже целиком поменять сервер на более мощный. При должной подготовке даже серьезные изменения в системе не скажутся на пользователях, поскольку к началу следующего рабочего дня их будет ждать полностью способная выполнять свои функции система. Совсем другая картина получается с интернет-приложениями, пользователи которых находятся в разных часовых поясах и имеют полное право затребовать функции разработанного сервиса в любое время любого дня. А между тем задачи администрирования и сервисного обслуживания никуда не исчезли, их по-прежнему необходимо выполнять. И одно из возможных решений — это правильная архитектура разрабатываемого приложения, допускающая временную неработоспособность одной или нескольких компонент без нарушения функционала системы в целом. Это возможно.

### 1.1.2.4. Особенности пользовательского интерфейса

Пользовательский интерфейс web-приложений тоже имеет свои особенности, обусловленные большой (возможной) аудиторией. Web-приложения должны быть простыми, насколько это вообще возможно, то есть интуитивно понятными для большинства пользователей. Предположим, разрабатывается интерфейс приложения, предназначенного для работы в рамках некоторого предприятия. При этом всегда есть возможность устроить для будущих пользователей системы обучающие курсы, рассказать им, что есть на экране и что разработчики хотели этим сказать. В случае с web-приложением такой возможности не будет. Пользователь либо догадается сам, что разработчики имели в виду, либо просто уйдет.

Поскольку у пользователя есть варианты его поведения внутри созданного web-приложения (варианты пути пользователя), нужно выделять желаемый вариант поведения пользователя: цветом, размером, шрифтом. Если довести ситуацию до шутливого абсурда, то получим «идеальный web-интерфейс» — это белый экран и на нем огромная красная кнопка «Оплатить».

Обычно разработчики приложений придерживаются неких правил хорошего тона. Эти правила не являются уникальными именно для интернета, они обусловлены общечеловеческой этикой. На сайт в первый раз попал человек. Возможно, случайно, возможно, по рекомендации своих знакомых. Ситуация точно такая же по сути, но без привязки к сетям и сайтам: вы устроили грандиозный прием (или небольшую вечеринку) и зашел случайный гость. Как вы себя поведете в таком случае, учитывая, что в госте вы — заинтересованы? Это же ваш посетитель и вам крайне важно, чтобы у него осталось хорошее впечатление, и он пришел сюда еще раз. А лучше бы — еще и друзей с собой привел. Если посмотреть на ведущие web-приложения мира, то все они ведут себя в данной ситуации одинаково. Наверное, так же, как вы бы себя повели в жизненной ситуации с вечеринкой.

— Человека нужно встретить. Причем встретить прямо на пороге, чтобы он не смог уйти из-за того, что просто постеснялся войти внутрь. Объяснить ему, что вы ему очень рады, что он не будет здесь лишним. Попросить его *чувствовать себя как дома* и так далее.
— Его нужно познакомить с обстановкой и остальными гостями. Вкратце обрисовать ему, куда он попал, что здесь проис-

ходит. То есть, возвращаясь к ситуации с интернетом, коротенько пояснить суть приложения.

— Человеку нужно создать *благоприятную обстановку* — следует сделать так, чтобы в рамках приложения пользователю стало хорошо и он не пожалел, что сюда зашел, почувствовав свою прямую выгоду.
— Заинтересовать человека *бонусами*. Пришедшему пользователю нужно дать понять, что если он приведет сюда друзей, то ему станет еще лучше, выгода будет еще больше.

Другой особенностью является отсутствие стандартов на оформление визуальных элементов интерфейса. Если мы возьмем элемент *кнопка*, или *поле ввода*, или *меню*, или любой другой визуальный элемент, то в рамках операционной системы этот элемент выглядит одинаково и ведет себя одинаково в любой программе. Соответственно, пользователь привыкает к расположению, внешнему виду и поведению визуальных элементов данной операционной системы. Меню в любой программе расположены вверху окна (Windows) или в специальной строке меню (MacOsX). Кнопка всегда имеет цвет, определенный текущей палитрой пользователя. Внешний вид кнопки одинаков в разных программах. И так далее.

Описанное выше не является справедливым для web-приложений в полной мере. Стандартами языка HTML предусмотрены визуальные элементы *кнопка*, *поле ввода* и так далее. Однако как в силу сравнительной бедности этих элементов, так и в силу имеющихся развитых программных средств для их модификации каждый разработчик *раскрашивает* элементы и изменяет их поведение по своему усмотрению.

Можно попытаться сформулировать некоторое *правило хорошего тона* для разработки интерфейса web-приложения: не стоит чрезмерно модифицировать внешний вид и поведение визуальных элементов по сравнению с наиболее распространенным, имеющимся на уже существующих широко известных сайтах. Есть смысл сначала посмотреть, как та же самая функция реализована наиболее значимыми, распространенными web-приложениями. В противном случае имеется существенный риск дезориентировать пользователя. Если нарисовать круглую кнопку, пользователи могут просто не найти ее на экране, потому что ранее сформировавшаяся привычка заставляет искать глазами кнопку в форме квадрата. Конечно, вышесказан-

ное — это только правило, а вовсе не закон, ведь вполне возможно, что нестандартный интерфейс как раз и является *фишкой* разрабатываемого сайта.

Приведенный перечень особенностей не является исчерпывающим, но является достаточным для демонстрации специфичности интернет-приложений, их непохожести на задачи классического прикладного программирования.

## 1.2. Технологии построения интернет-приложений

Приложения вообще и интернет-приложения в частности имеют особенность развиваться с течением времени. Изначально задумывалась только одна небольшая функция. Она была реализована, появились первые пользователи. Позже к приложению добавилась еще одна функция, следом по желанию пользователей была добавлена четвертая, пятая и так далее. И может получиться ситуация, когда плохо структурированный, плохо читаемый, плохо поддерживаемый программный код выступит в качестве снежной лавины и *засыплет* под собой разработчиков. Такие ситуации происходили и с крупными проектами, пример тому — один из первых появившихся браузеров, — NetscapeNavigator. Переписать уже имеющееся приложение — процесс трудоемкий и длительный, за это время конкурирующие продукты неизбежно займут ваше место на рынке. Так, за время переделывания браузер Mozilla Firefox потерял лидирующие позиции.

Чтобы на определенном этапе развития не пришлось полностью переписывать все приложение, необходимо с самого начала его разработки следовать определенным парадигмам программирования. Это подходы, методологии, которые не зависят от конкретного языка программирования и позволяют организовать внутреннюю структуру создаваемого программного продукта определенным образом, способным строить приложения из логически законченных *кирпичиков*, взаимодействующих друг с другом. Избежать ситуации с необходимостью переписывания определенных частей кода, скорее всего, не удастся. Но вот разграничить между собой различные *сущности* создаваемого приложения, четко оговорить времена и способы взаимодействия этих сущностей между собой — задача вполне реаль-

ная. И затем, при необходимости доработки, изменяется только один такой кирпичик, все остальное приложение остается в его первоначальном виде.

### 1.2.1. ТЕХНОЛОГИЯ СОЗДАНИЯ КЛИЕНТСКОЙ ЧАСТИ

Код разрабатываемого приложения может выполняться как на серверной стороне, так и на стороне клиента. Более того, прямо в тексте HTML друг за другом могут идти куски кода, часть из которых будет выполняться на стороне сервера, а часть — на стороне клиента. Применительно к web-приложениям наличие какого-то кода с выполнением на стороне клиента не является обязательным. Вполне возможно обеспечить достаточную функциональность, пользуясь в качестве клиента только браузером и ограничиваясь HTML. Все программирование при этом целиком будет возложено на серверную часть. Однако такое приложение будет выглядеть несколько архаично и вряд ли понравится пользователям.

#### 1.2.1.1. HyperText Markup Language

HTML, HyperText Markup Language — это язык разметки документов. По сути, это обычный текст. Кроме собственно содержимого, он одновременно несет в себе информацию о том, как это содержимое следует отображать или обрабатывать на стороне клиента.

Любой документ в HTML состоит из элементов. Начало и конец каждого элемента обозначаются специальными пометками — тэгами, еще иногда их называют дескрипторами, в том числе таким элементом является и собственно документ, который заключается в тэги <HTML></HTML>. Как правило, тэги парные. То есть на каждый открывающий тэг в документе содержится точно такой же, но закрывающий, обозначающий конец элемента. Элементы могут быть пустыми, например, перевод строки, <BR/>. Тэг в этом случае является одновременно и открывающим, и закрывающим. Регистр букв значения не имеет, большие и маленькие литералы обрабатываются одинаково. Поскольку окончательное форматирование отображаемого документа производится браузером, игнорируются также и символы возврата каретки — перевода строки, также достаточно вольно интерпретируются символы пробела и табуляции. При необ-

ходимости браузер можно *настоятельно попросить* сделать именно так, как задумывалось. Либо при помощи специальных тэгов, либо при помощи так называемых *сущностей* (entities). Например, появление в документе сочетания   приведет к тому, что браузер в этом месте отобразит пробел даже в том случае, если в этом, на взгляд браузера, и не было необходимости. Сущности позволяют отобразить на экране специальные символы. Например, © приведет к формированию на экране знака авторского права ©, < — символа *меньше* (<), > — символа *больше* (>). Появление в документе двух последних символов в их обычном виде (<, >) является некорректным в связи с тем, что они используются для формирования тэгов. Вообще говоря, с помощью *сущностей* можно отобразить на экране любой символ, если указать его в виде &#NNNN; где NNNN — код символа в Unicode.

Элементы могут иметь свои собственные атрибуты, которые указываются в открывающем тэге.

```
<A HREF=”http://www.mysite.ru/hello.html”>Ссылка</A>
```

Появление такого тэга приведет к формированию на экране ссылки — выделенной надписи, при нажатии на которую произойдет переход по указанному в атрибуте href-адресу. Существует ряд атрибутов, которые могут быть применены к любому элементу внутри документа. Это такие атрибуты, как ID, NAME, CLASS, STYLE и так далее. Другие атрибуты могут быть применены только к элементам определенного вида. Например, указанный выше HREF применяется к атрибуту A (ссылка).

Тэги могут быть вложенными. В этом случае вложенный элемент наследует атрибуты его *родителя*. Пример вложенности тэгов:

```
<b> Этот текст будет выделен жирным шрифтом.
    <i> А этот текст, кроме жирного, будет еще и наклонным
        шрифтом.
    </i>
</b>
```

Поскольку спецификаций HTML, включая его расширения XHTML (Extendedhypertextmarkuplanguage), существует несколько, то правилом хорошего тона считается первым тэгом документа указывать <! DOCTYPE точное_указание_используемой_спецификации >, однозначно сообщающее браузеру, какого типа спецификация используется в документе. Обязательным это правило не является, просто

в противном случае корректность отображения всех элементов документа не гарантируется.

Язык HTML дает возможность добавления комментария, для чего предусмотрен специальный тэг. Символы между <!-- и --> игнорируются браузером, не отображаются на экране. Комментарий может включать в себя другие тэги, которые в этом случае также не отображаются на экране. Вложенные комментарии недопустимы.

В общем случае документ делится на заголовок и тело. Заголовок на экране не отображается, но влияет на поведение браузера. В заголовке можно указать, разрешается ли кэшировать данную страницу, в какой кодировке ее желательно отображать, что написать в качестве заголовка окна браузера и так далее.

Ниже приведен пример HTML-документа.

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Пример страницы</title>
<meta http-equiv="Content-Type" content=»text/html; charset=utf-8»>
</head>
<body>
<h1 align="center">Пример страницы</h1>
<!— Меню —>
<div>Меню</div>
</body>
</html>
```

#### 1.2.1.2. Cascading Style Sheets

CSS — Cascading Style Sheets — это язык стилей, определяющий отображение HTML-документов. При помощи CSS нельзя изменить содержимое страницы, но можно полностью изменить ее внешний вид: шрифты, цвета, расположение на странице, реакцию на различные события. В принципе, применение CSS для web-страниц не является обязательным. Тех же результатов можно достичь и средствами одного HTML. Однако CSS более точен, проработан, предоставляет большие возможности при ясном синтаксисе. Кроме того, применение каскадных таблиц стилей делает процесс разработки HTML-страницы более понятным, более гибким, более управляемым. *Каскадными* они

называются потому, что действуют на все дерево вложенных объектов, то есть *каскадом*: если тело документа содержит заголовок и абзац и к телу документа был применен стиль *сделать фон красным*, то и заголовок, и абзац (нижестоящие элементы) тоже будут иметь красный фон. В общем виде CSS выглядит следующим образом:

selector {property: value;}

selector — к какому конкретно HTML-тэгу (или тэгам) применяется новое свойство. Например, *body* применит ко всему *телу документа*.

property — какое свойство будем менять. Например, это будет цвет фона, *background-color*.

value — новое значение изменяемого нами свойства.

Таким образом, в HTML изменение фона страницы можно добиться, указав на странице:

<body bgcolor="#FF0000">

средствами CSS то же самое выглядит следующим образом:

body {background-color: #FF0000;}

Существует три способа применить CSS к документу.

— Непосредственно написав его в HTML, применив к данному элементу атрибут style.

<body style="background-color: #FF0000; "></body>

— Описав стиль внутри HTML документа.

```
<html>
    <head>
    <title>Example</title>
    <style type="text/css">
    body {background-color: #FF0000;}
    </style>
    </head>
    <body>
    <p>This is a red page</p>
    </body>
</html>
```

— Выделив стили в отдельный файл и подключив данные стили к документу при помощи тэга link.

<linkrel="stylesheet" type="text/css" href="style/style.css"/>

Наибольшее распространение имеет третий способ, потому что он позволяет отделить содержание документа от его оформления.

Содержание может меняться, в том числе и динамически подгружаться на страницу. А стиль при этом остается неизменным. Кроме того, внешние стили позволяют написать один стиль и применить его ко всему сайту сразу (то есть к нескольким HTML-документам).

Широкое применение в CSS имеет понятие *класс*. *Класс* — с точки зрения CSS — это именованный набор атрибутов, который целиком может быть применен к любому объекту на странице. Например:

```
.site{
color: navy;/* Синий цвет текста */
font-style: italic;/* Курсивное начертание */
}
```

После такого объявления все элементы, имеющие установленный класс site, будут иметь синий цвет и курсивный шрифт:

```
<p class="site">Текст, который будет выведен синим курсивом</p>
<b class="site">Объявление</b>
```

Последняя строка предписывает браузеру вывести слово «Объявление» жирным шрифтом. Однако, поскольку к ней дополнительно применен класс site, наше «Объявление» будет выведено синим, наклонным и жирным шрифтом одновременно.

#### 1.2.1.3. DOM (Document Object Model)

Полученная в HTML/XHTML/XML страница отображается браузером. При этом с программной точки зрения каждый входящий в документ элемент (в том числе и сам *документ*) являются объектом. DOM — это не зависящий ни от платформы, ни от языка программирования программный интерфейс, позволяющий программам и скриптам получить доступ к содержимому документов HTML, XHTML, XML; а также менять содержимое, структуру, оформление таких документов. Несмотря на все усилия специалистов W3C по стандартизации, модель DOM и поведение входящих в нее объектов до сих пор отличаются друг от друга в разных браузерах. Отличия несущественны, но они — есть. Поэтому для корректной работы клиентской части web-приложения лучше перед применением какой-либо особенности проверить, поддерживается ли данная особенность данным браузером. Для отображения страницы DOM не нужен, браузер при форматировании страницы и выводе ее на экран не обязан знать о существовании внутри него каких-то объектов. Но факт

существования DOM дает возможность *оживить* клиентскую часть web-приложения, динамически изменяя содержимое и/или оформление страницы.

Согласно DOM-модели, документ является иерархией объекта. Каждый HTML-тэг образует отдельный элемент-узел, каждый фрагмент текста — текстовый элемент и так далее. Проще говоря, DOM — это представление документа в виде дерева объектов. Это дерево формируется за счет вложенной структуры тэгов и текстовых фрагментов страницы, каждый из которых образует отдельный узел.

Пусть есть следующая HTML-страница:

```
<html>
<head>
<title>Заголовок</title>
</head>
<body>
        Прекрасный документ
</body>
</html>
```

В результате отображения указанной страницы в браузере будет создано следующее дерево объектов (рис. 1.4):

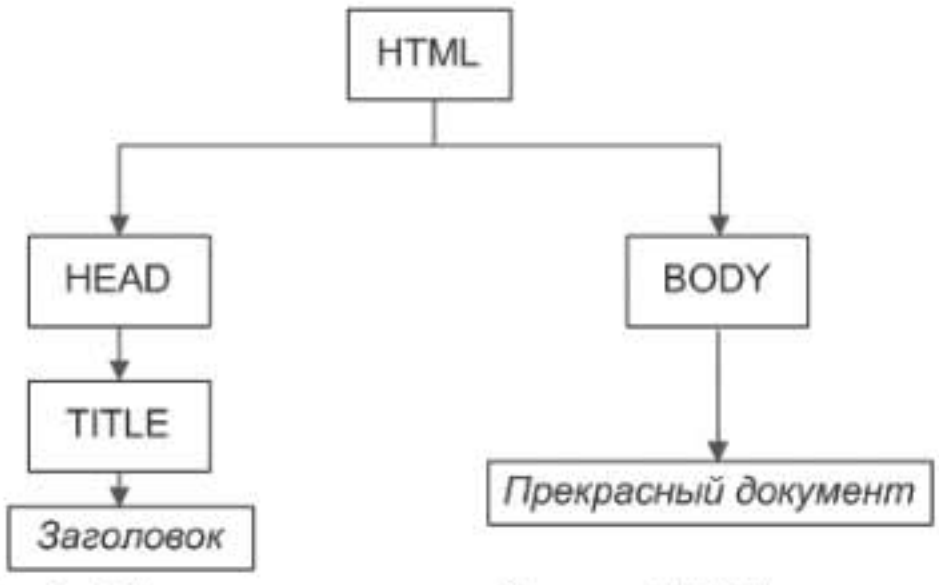


Рис. 1.4. Пример структуры объектов HTML-документа

Каждый DOM-элемент является объектом, то есть предоставляет методы и свойства для манипуляции своим содержимым, для доступа к родителям и потомкам.

Существует несколько способов получить для последующих манипуляций данный конкретный элемент. Можно, например, взять роди-

тельский элемент, получить список всех его потомков и искать в списке. Можно получить по имени тэга массив всех элементов данного типа. Самый простой из всех возможных способов — это получение элемента по его идентификатору. Для этой цели в тэг добавляется атрибут id (уникальный в пределах данной страницы идентификатор):

```
<div id="mydiv>
<table id="table1">
<tr id="oneRow">
<td id="firstCell">Первыйстолбец</td>
<td id="secondCell">Второйстолбец</td>
</tr>
</table>
</div>
```

При таком подходе нужный нам элемент находится однозначно и очень просто.

Пример на языке JavaScript:

```
var myDiv = document.getElementById ("mydiv");
```

Кроме элементов, их свойств и набора функций для манипуляции с ними, DOM предоставляет еще и события. Например, событие click происходит при клике мышкой на элементе, событие mouseover — когда указатель мыши перемещается над указанным элементом. На любое из предоставляемых событий можно посадить свой обработчик. Обычно обработчики называют по схеме «on+имя_события». Указанное выше событие click можно перехватить, назначив на данный элемент функцию onclick:

```
<tdid="firstCell"
onclick="alert ('Вы нажали на первый столбец');">
Первый столбец
</td>
```

Реакции на событие click можно достичь и иным способом, просто присвоив свойство onclick нужного нам элемента:

```
document.getElementById ("firstCell").onclick = function () {
alert ("Спасибо, что нажали на первый столбец");
};
```

С программной точки зрения между двумя приведенными способами не существует различий, в том и в другом случае код Вашего обработчика будет выполнен.

Несмотря на все усилия международного web-консорциума, направленные на стандартизацию, единой DOM-модели в разных браузерах не существует до сих пор. Что-то работает только и исключительно в Microsoft Internet Explorer, что-то — только в Mozilla Firefox. Обойти данное обстоятельство несложно, но это требует определенных трудозатрат и внимания.

#### 1.2.1.4. JavaScript

Есть несколько способов доступа к DOM-объектам. Наиболее распространенные из них на сегодняшний день базируются на языке JavaScript и его расширениях. Причины такого распространения в том, что JavaScript работает практически во всех существующих браузерах и прекрасно интегрирован с HTML/CSS.

JavaScript — это объектно-ориентированный язык. Иногда его называют *прототипно-ориентированным*: особенность языка заключается в том, что в нем нет классов. Классов нет, но есть объекты. Более того, любая *сущность* в языке есть объект, который может быть перекрыт и изменен. Можно изменить, например, поведение целого числа (int), добавив ему какие-то специфические, необходимые методы. Достигается это за счет использования прототипов. JavaScript — слабо-типизованный язык, контроль типов существует, но не является строгим. Поскольку все в языке есть *объект* и контроль типов слабый, то одной и той же переменной можно присваивать все, что угодно: целые числа, вещественные числа, функции, другие объекты и так далее.

JavaScript — это интерпретируемый язык, так называемый *скрипт*. Между текстовым представлением программы и ее выполнением нет промежуточных стадий, нет компиляции и компоновки. Написанный Вами текст будет выполняться построчно при помощи специальной встроенной в браузер программы-интерптератора.

JavaScript автоматически работает с памятью и производит сборку мусора. Специально освобождать выделенное под объекты пространство в памяти не нужно, интерпретатор сделает это самостоятельно.

Существует три способа вставить JavaScript в HTML-страницу:

— внутри страницы:

```
<scripttype="text/javascript">
alert ('Hello, World!');
</script>
```

— внутри тэга:
<td id="firstCell"
onclick="alert ('Вы нажали на первый столбец');">Первый столбец</td>
— вынесение скрипта в отдельный файл. Файл со скриптом подключается при помощи следующей конструкции:
<scripttype="text/javascript"src="someFile.js"></script>

JavaScript применяется не только в клиентской части вэб-приложений. На его основе построен ряд серверных средств, например NodeJS. Кроме того, JavaScript широко используется в браузерных операционных системах (IndraDesktop WebOS, IntOS).

#### 1.2.1.5. JQuery

JQuery представляет собой популярную библиотеку JavaScript, которая позволяет писать более высокоуровневые, более управляемые программы на языке JavaScript. Использование JQuery на начальном этапе может вызывать некоторые трудности из-за несколько необычного синтаксиса. А между тем суть довольно проста. Вся работа с JQuery ведется с помощью функции $. Условно можно разделить два способа применения функции $:

$("селектор") возвращает объект JQuery. При этом *селектор* выражение, согласно которому отбираются DOM-объекты. Это выражение по своему синтаксису полностью соответствует принятым в CSS соглашениям. Например:

$("#mydiv") — возвращает объект с id="mydiv".

$(".greenPanel") — возвращает все объекты данного HTML-документа, для которых установлен class="greenPanel".

$("a") — возвращает все ссылки, существующие в данном документе.

Каждый из операторов JQuery возвращает объект, отобранный или измененный в результате проведенных манипуляций. Эта особенность позволяет проводить каскадирование операций.

$("div.test").add ("p.quote").addClass ("blue").slideDown ("slow");

В результате одной строки сначала находим на странице все элементы , которые имеют установленный класс test. В полученное множество добавляем элементы <p>, имеющие класс quote. Далее всему полученному набору элементов устанавливаем class="blue" и визуально плавно спускаем их вниз по экрану. Приведенные приме-

ры демонстрируют емкость JQuery, возможность выполнения значительного числа операций небольшим количеством строк кода.

Библиотека JQuery состоит из ядра и набора подключаемых (при необходимости) частей — плагинов, выполняющих специализированные функции. Например, в виде отдельного плагина оформлены процедуры работы с пользовательским интерфейсом (JQuery UI).

#### 1.2.1.6. AJAX

AJAX — Asynchronous Javascript and XML — набор технологий, предназначенных для динамического обмена данными с сервером. AJAX позволяет обмениваться данными в фоновом режиме: пользователь не замечает, когда его браузер запрашивает данные. Это асинхронная технология, то есть ожидание ответа сервера (то есть зависать при этом клиентскому приложению не нужно). Вместо этого в запросе указываются две функции, одна из которых будет отработана в случае успешного завершения запроса, а другая — в случае возникновения ошибки. Сама программа на клиентской части продолжает свое выполнение, не ожидая результата запроса. Совместно с DOM-моделью AJAX позволяет сделать странички web-приложений динамичными, интерактивными, *оживить* их поведение. Как это следует из названия, AJAX базируется на JavaScript, является его расширением. Технологии AJAX поддерживаются практически всеми современными браузерами. В основе работы AJAX лежит объект XMLHttpRequest. JQuery содержит набор функций *обертки* для работы с AJAX, что позволяет писать ассинхронные запросы максимально простым способом. Следующий пример загружает с сервера JavaScript код и выполняет его:

```
$.ajax ({
type: "GET",
url: "test.js",
dataType: "script"
});
```

Вызов $.ajax () возвращает объект типа XMLHttpRequest. В большей части случаев возвращаемое значение не используется, но оно все-таки доступно, например, при необходимости прервать выполняющийся запрос. Существует возможность установить параметры запроса по умолчанию, которые будут распространяться на все вызываемые в программе запросы. Делается это при помощи функции $.ajaxSetup ().

## 1.2.2. ТЕХНОЛОГИИ СОЗДАНИЯ СЕРВЕРНОЙ ЧАСТИ

### 1.2.2.1. Web-серверы

Web-сервер — это совокупность аппаратно-программных средств, которая принимает HTTP-запросы от клиентов (как правило, web-браузеров) и выдает HTTP-ответы. В простом случае *ответ* web-сервера — это содержание расположенных на нем файлов: документов, изображений, фильмов и так далее. Такие данные называют статическими: если пользователь запросит несколько раз информацию по одному и тому же адресу, web-сервер выдаст в точности одно и то же содержимое. Для динамического подхода, то есть когда выдаваемая информация будет меняться во времени и/или зависеть от параметров запроса, одного web-сервера недостаточно. В этом случае web-сервер служит переходным звеном между клиентом и некой программной средой, формирующей содержимое HTML-страницы. И здесь напрашивается вопрос: а зачем тогда нужен web-сервер, если динамическую страницу все равно формирует приложение.

При небольшой нагрузке на приложение — несколько одновременно подключенных пользователей (единицы) — web-сервер не является абсолютно необходимым звеном. Практически все существующие на сегодняшний день пакеты разработки web-приложений имеет в своем составе *легковесный* web-сервер. Достаточно запустить среду разработки на выполнение, и к ней можно подсоединиться браузером и работать с ней по протоколу HTTP.

Проблемы начинаются тогда, когда число одновременно работающих клиентов превышает некоторое критическое значение. И одна из функций web-сервера — это максимально возможное снижение нагрузки за счет ее распределения по разным частям системы. Оптимизация может затрагивать не только работу самого web-сервера, но и начинаться непосредственно с уровня ядра операционной системы. Современные операционные системы могут быть настроены таким образом, что после соединения с клиентом они не будут *беспокоить* web-сервер до тех пор, пока клиентская сокета не получит данные (с сокеты можно читать входной поток данных), либо пока не придет полный HTTP-запрос.

Такой подход позволяет web-серверу меньше времени простаивать в ожидании получения данных, повышая эффективность системы в целом.

Все современные (*полновесные*) web-серверы используют асинхронную обработку поступающих от клиентов запросов. То есть после получения запроса от клиента сервер переадресует этот запрос потоку обработчика (worker). Сам сервер при этом продолжает принимать запросы от клиентов и переадресовывать их обработчикам. Существует несколько стратегий работы с обработчиками.

— Количество worker может быть жестко задано. При старте сервера запускается, например, 10 потоков. При получении 11-го запроса сервер выдаст клиенту ответ *временно недоступен*.
— Worker могут порождаться сервером динамически в зависимости от загрузки. При этом оговаривается начальное число worker, максимально допустимое количество, минимальное и максимальное количество свободных worker.
— Worker может быть как процессом, так и потоком. Различные процессы не взаимодействуют между собой и не могут разделять данные. В отличие от процессов, потоки имеют общее разделяемое пространство данных и могут обмениваться информацией между собой.
— Worker-процессы могут запускаться как внутри родительского серверного процесса, так и снаружи. При последнем варианте worker могут быть распределены по разным компьютерам.
— Worker — вне *родительского* сервера могут представлять собой непосредственно процессы среды web-приложения.
— Сервер может отслеживать разные варианты распределения нагрузки между порожденными worker: равномерную или поочередную загрузку.

Комбинируя эти факторы, можно попытаться добиться наилучшей производительности системы в целом.

Для логического разделения, оптимального распределения нагрузки по вычислительным мощностям Worker могут запускаться и на другом (-их) компьютере (-ах), не обязательно на том же самом, где расположен сам принимающий запросы от клиентов сервер. В этом случае общение между сервером и worker происходит по определенному сетевому протоколу. Это уже не может быть HTTP, потому что вместе с полученным запросом worker передается расширенная информация об окружении сервера. В частности, в качестве такого протокола общения может применяться FastCGI.

В современном интернете существует как минимум несколько десятков решений на тему web-серверов, как бесплатных, так и коммерческих. Как правило, коммерческие продукты имеют графический интерфейс управления и настройки сервера, с ними работать гораздо проще. В противоположность этому свободно распространяемые программные продукты обычно конфигурируются при помощи текстовых файлов.

#### 1.2.2.2. Технология MVC

MVC — Model View Controller — это концепция, парадигма, подход в том плане, что MVC-подход не опирается на какой-то конкретный язык программирования, не зависит от выполняемых задач и может существовать как на клиенте, так и на сервере. Идеология заключается в том, что работу с данными (model), представление на экране (view) и обработку внешних запросов (controller) разделяют между собой (рис. 1.5).

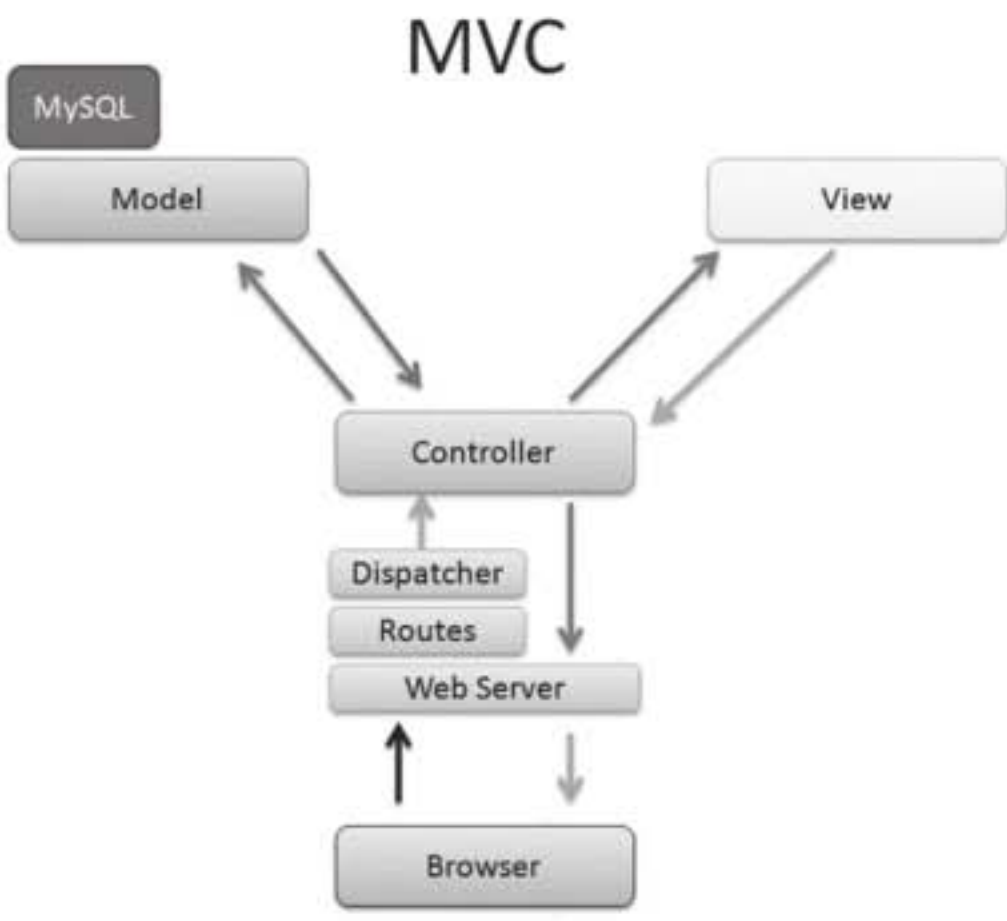


Рис. 1.5. Структура MVC-модели

При этом работающая с данными модель может быть либо *пассивной*, то есть ее каждый раз нужно спрашивать, не изменились ли данные, либо *активной*, в этом случае она сообщает всем зависящим от нее views, что данные изменились и надо бы их перерисовать.

Такое разбиение позволяет, во-первых, обезопасить себя от неожиданных изменений в поведении приложения в целом. При изменении модели мы можем быть уверены, что вносимые изменения отразятся только на данных и ни на чем больше. Во-вторых, подход позволяет одним и тем же данным сопоставить несколько представлений. И наоборот, сделать одно представление (предположим, статистику по месяцам), которое работало бы с разными типами исходных данных.

Забегая вперед, отметим, что достаточно часто в качестве view фигурируют шаблоны (например, FreeMaker), а в качестве model непосредственное отображение объектов на СУБД, то есть ORM.

Концепция MVC оказалась достаточно удачной, вследствие чего она служит основой для ряда framework, как серверных, так и клиентских. Из серверных Java framework можно отметить Play, из JavaScript клиентских backbone.js.

#### 1.2.2.3. Технологии объектно-реляционных отображений (ORM)

ORM — Object Relational Mapping — это такая технология работы с базами данных, при которой программа напрямую в СУБД не обращается. Вместо этого программа оперирует объектами, *самостоятельно отображающимися* на реляционную структуру базу данных (рис. 1.6.). Сама низкоуровневая работа с базой данных происходит за кадром.


Рис. 1.6. Структура ORM-технологии

Достаточно создать объект, например User, определенным образом описанный. И после этого можно совершать с ним различные действия, специфичные для базы данных: записать его в базу, выбрать из базы по определенным критериям, рассматривать другие объекты не как подчиненные таблицы, а как поля основного объекта и т. д.

Для работы с базами данных Java всегда имеет механизм JDBC, Java Data Base Connectivity. ORM — это следующий уровень, стоящий

над JDBC. Применение ORM скрывает детальность работы с СУБД, позволяет обойтись без громоздких конструкций на языке SQL и без присваивания параметров JDBC-запросу. Все вместе (ORM + JDBC) позволяет писать программы таким образом, что они становятся независимыми от базы данных. Одна и та же программа может работать как с Oracle, так и с MS SQL.

Оборотной стороной применения ORM является снижение скорости работы с базой данных. Правда, к этому утверждению нужно добавить *как правило*, т. к. в реальности на скорость работы оказывают влияние сразу несколько факторов: и структура данных, и наличие/правильное использование индексов, и построение запросов. Но в общем случае ORM снижает скорость работы приложения, повышая при этом его гибкость и мобильность.

# Часть 2.
# Технологии создания интернет-приложений

Эта часть книги позволяет на практике разобраться с технологиями создания интернет-приложений. В качестве основного языка программирования выступает Java. Также продемонстрированы технологии создания клиентской части: HTML, CSS, JavaScript, JSON, JQuery, AJAX и инструментарий, позволяющий создать серверную часть: ORM Hibernate, шаблонизатор freeMarker. Детально описана структура HTTP.

Вторая часть состоит из ряда лабораторных работ, последовательное выполнение которых позволит овладеть практическими навыками создания интернет-приложений. Каждую лабораторную работу может выполнять один человек (или команда максимум из двух человек). Некоторые лабораторные работы содержат индивидуальное задание, которое нужно выбрать. Выбранное задание не должно совпадать с выбором других команд в группе.

## 2.1. Лабораторная работа № 1

### 2.1.1. ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является ознакомление со средой разработки IntellijIDEA и написания простейшего серверного приложения на языке программирования Java, которое потом может быть использовано в качестве основы для дальнейших лабораторных работ.

### 2.1.2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Любое интернет-приложение работает через сокеты. Для создания интернет-приложения на java обычно используются два класса java.net.Socket и java.net.ServerSocket, с помощью которых и осуществляется передача данных по сети. В языке Java (точно так же, как во многих других языках программирования высокого уровня) понятия *серверная сокета* и *клиентская сокета* разделены и обеспечиваются разными классами.

#### 2.1.2.1. Описание класса Socket

Этот класс реализует клиентские сокеты. Сокета является конечной точкой для передачи данных между двумя машинами.

Основные конструкторы класса Socket:

— Socket (InetAddressaddress, intport) — создает экземпляр класса Socket с входящими параметрами IP-адресом и портом той машины, с которой мы связываемся.

— Socket (Stringhost, intport, InetAddresslocalAddr, intlocalPort) — более точная настройка сокета, создает экземпляр класса Socket с входящими данными о Ip-адресе и порте локальной и удаленной машины.

— Socket (InetAddresshost, intport, booleanstream) — позволяет настроить сокету на передачу данных по протоколу TCP/IP (stream = true) или UDP (stream = false), создает экземпляр класса Socket с входящими параметрами IP-адресом и портом той машины, с которой мы связываемся, и параметром *stream*.

#### 2.1.2.2. Основные методы класса Socket

— getInputStream() — метод, который возвращает экземпляр класса InputStream, отвечает за входящий поток данных сокеты.

— getOutputStream() — метод, который возвращает экземпляр класса OutputStream, отвечает за исходящий поток данных сокеты.

— close() — метод завершения работы сокеты, после выполнения этого метода дальнейшее взаимодействие с объектом Socket невозможно.

#### 2.1.2.3. Описание класса ServerSocket

Этот класс реализует сокеты сервера. Сокета сервера ожидает запросы от клиентского приложения, который обращается к ней либо через сеть, либо с того же компьютера. Сокета выполняет некоторую работу, основанную на том запросе, и затем возвращает результат запрашивающей стороне.

Основные конструкторы класса ServerSocket:

— ServerSocket (intport) — создает экземпляр класса ServerSocket, с входным параметром порт (локальной машины), при этом порт должен быть свободен, иначе произойдет событие ошибки.

— ServerSocket (intport, intbacklog, InetAddressbindAddr) — создает экземпляр класса ServerSocket, с входными параметрам и порт, ip-адрес (bindAddr) и количеством возможных подключений (backlog), при этом порт должен быть свободен, иначе произойдет событие ошибки.

#### 2.1.2.4. Основные методы класса ServerSocket

— accept() — возвращает экземпляр класса Socket, с которым в дальнейшем будет работа по обмену данными. Данный метод ожидает подключения к созданной ServerSocket (к открытому порту, описанному при создании ServerSocket), пока подключение не произошло, находится в состоянии ожидания.

— close() — метод завершения работы серверной сокеты, после выполнения этого метода дальнейшее взаимодействие с объектом ServerSocket невозможно.

Обычно программная часть сервера и клиента не сильно отличаются друг от друга:

Сервер:

```
ServerSocketserverSocket = new ServerSocket(12345);
Socket socket = serverSocket.accept();
```

Клиент:

```
Socketsocket = newSocket ("localhost",12345);
```

Дальнейшее взаимодействие осуществляется непосредственно с объектом сокеты.

### 2.1.2.5. Работа с входящим и исходящим потоком байт

Основные методы, необходимые для сокеты — это передача данных, то есть получение и отправление потока данных:

— для получения потока входящих данных нужно использовать метод getInputStream(). InputStream — абстрактный класс, задающий используемую в Java модель входных потоков;

— для отправки исходящего потока данных нужно использовать метод getOutputStream(). OutputStream — абстрактный класс. Он задает модель выходных потоков Java.

getInputStream() и getOutputStream() возвращают поток байт, с которыми не очень удобно, для этого существует множество стандартных классов, которые помогают представить входящий (исходящий) поток в более удобном для нас формате, например: в виде String, class,.. Перечислим основные классы, которые могут использоваться для преобразования потока байт в другой формат и выступать в качестве обертки: ObjectInputStream и ObjectOutputStream позволяют передавать/получать ранее сериализированные объекты, то есть мы можем передавать ранее сформированный экземпляр класса и получить его на другой стороне как структурированный объект.

Пример:

Листинг класс Student, который хранит два значения: имя, возраст; и метод getTitle (), возвращает некоторую структурированную информацию о содержании класса. Данный класс реализует интерфейс Serializable.Serialization, предоставляет стандартный механизм для создания сериализуемых объектов. Сериализация — это процесс сохранения состояния объекта в последовательность байт.

Пример POJO-класса Student:

```
1.  Public class Student implements Serializable {
2.  Private String name;//Имя
3.  private Integer age;//Возраст
4.  public String getName() {
5.  return name;
6.  }
7.  public void setName(String name) {
8.  this.name = name;
9.  }
10. public Integer getAge() {
11. return age;
```

```
12. }
13. public void setAge (Integer age) {
14. this.age = age;
15. }
16. public String getTitle() {
17. return getName()+" "+getAge();
18. }
19. }
```

Действия отправителя объекта через сокету.

При отправке объекта изначально его необходимо создать и заполнить данными. Для этого создается экземпляр класса Student и заполняется данными.

```
1. Student student = new Student();
2. student.setAge (18);
3. student.setName("Mr.Smith");
```

После того как экземпляр класса будет создан, его можно будет отправлять, для этого используем класс ObjectOutputStream. Этот класс позволяет отправлять данные объектом, который реализуют интерфейс java.io.Serializable. Создание переменой, присвоенной экземпляру класса ObjectOutputStream, с помощью которой в дальнейшем будет осуществляться передача данных. ObjectOutputStream использует в качестве ресурса выходной поток данных ранее созданной сокеты.

```
1. ObjectOutputStream out = new ObjectOutputStream (socket.
   getOutputStream());
2. out.writeObject (student);//происходит отправление объекта
```

Действия принятия объекта через сокету.

Для того чтобы принять объект, нужно использовать класс ObjectInputStream, он позволяет входящие данные принимать в виде объекта, а не потока байт. Для этого создаем экземпляр класса ObjectInputStream, а в качестве входного параметра используем входной поток данных ранее созданной сокеты.

```
1. ObjectInputStream in = new ObjectInputStream (socket.
   getInputStream());
2. try {
3. Student student = (Student)in.readObject();//получение объ-
   екта Student
4. System.out.println (student.getTitle());//использование метода
   getTitle() класса Student, для вывода результатов на консоль
```

```
5. } catch (ClassNotFoundException e) {
6. e.printStackTrace();
7. }
```

В результате выполнения данных операций на экране принимающей стороны должна появиться надпись “Mr.Smith 18”.

*Работа с входным потоком данных как со строками.*

InputStreamReader является мостом между потоком байт и потоком символов. Несет в себе информацию о кодировке (сам производит кодировку и декодировку, например UTF-8). Для создания экземпляра класса InputStreamReader на вход необходимо подать класс InputStream, например System.in или socket.getInputStream.

BufferedReader буферизует символы и позволяет извлекать из потока как сформированные строки, так и просто символы. Для создания экземпляра класса необходимо поместить в него любой класс, реализующий абстрактный класс Reader. Например, InputStreamReader, StringReader. Пример:

```
1. InputStreamReader isr = new InputStreamReader (System.
   in);//преобразование входного потока байт
2. BufferedReader br = new BufferedReader (isr);//буферизация
   символов
3. while (true){
4. String st = br.readLine();//чтение из буфера строку
5. if (st==null)
6. {
7. break;
8. }
9. }
```

*Работа с исходящим потоком данных как со строками.*

PrintWriter позволяет введенный печатный текст представить к байтовому потоку вывода. Для создания экземпляра класса необходимо иметь входные параметры наследников абстрактного класса Writer (например OutputStreamWriter) или классов, реализующих интерфейсы Closeable, Flushable (например: OutputStream). Пример:

```
1. PrintWriter printWriter = new PrintWriter (socket.getOutput
   Stream(), true);
2. printWriter.println("Введите значение a:";
```

### 2.1.3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

#### 2.1.3.1. Подготовка рабочего места:

1. На компьютере, на котором будет проходить лабораторный практикум, необходимо установить следующие средства разработки:
   1.1. Java Development Kit (jdk);
   1.2. IntelliJIDEA (CommunityEdition, свободно распространяемый вариант);
   1.3. СУБД на выбор: MS SQL Server, MySQL, PostgreSQL, Oracle.
2. Создать новый пустой проект (File — NewProject…). Записать его в каталог, название которого соответствует вашей фамилии.
   2.1. При создании проекта необходимо указать, какое JDK вы будете использовать, если его не будет в предложенном меню, то вручную укажите путь до него. Например: C:\Program Files\Java\jdk1.7.0_15.
3. Создать пустой класс, содержащий статический метод выполнения (JavaApplication). Реализация метода может быть примерно следующей:

```
public static void main (String argc []) {
System.out.println (“Hello, World!”);
}
```

Запустить созданную программу на выполнение, убедиться, что на консоль (стандартный поток вывода) выводится надпись.

#### 2.1.3.2. Первая часть лабораторной работы

Требования к выполнению индивидуального задания.

— Индивидуальное задание должно быть оформлено в отдельном классе.
— Класс должен реализовывать созданный вами интерфейс Result, который будет иметь один метод, возвращающий строку результата вычислений getResult () без входных параметров.
— Все входные параметры должны поступать в класс с помощью перегрузок конструкторов и должны совпадать с индивидуальным заданием (пример: в задании 1 должно быть описано три перегруженных конструктора, которые будут принимать String, Double [], List<Double>).
— В классе должно быть описано поле, в котором будут храниться распарсенные значения (пример: в задание 1 в конструктор

класса поступает обычная строка «11, 32, 1, 22, 14», конструктор должен разбить данную строку по запятой на массив чисел и заполнить поле, с которым в дальнейшем мы будем работать для вычисления максимального значения).

— Для отладки и проверки работоспособности выполненного индивидуального задания создаем отдельный класс с методом main, который будет взаимодействовать с созданным ранее классом (с классом, в котором выполнено индивидуальное задание) и позволит вводить данные (в консоли) для вычисления, после чего вывести результат на экран.

Варианты индивидуального задания.

1. Найти максимальное значение среди множества чисел (без использования класса Math). Числа должны поступать в виде: строки с некоторым разделителем (пример: «11, 32, 1, 22, 14»); в массиве; списком чисел.
2. Сортировка списка слов, без использования сторонних классов сортировки, например Collections метод sort (). Слова должны поступать сплошным текстом с разделителем; списком; отдельными значениями данных.
3. Подсчет одинаковых слов. Слова должны поступать сплошным текстом и с разделителем; списком; отдельными значениями данных.
4. Подсчет четных и нечетных символов. Числа должны поступать в виде строки с некоторым разделителем (пример: «11, 32, 1, 22, 14»); в массиве; списком чисел.
5. Конвертер систем счисления. На вход поступает число, которое мы хотим конвертировать; система счисления конвертируемого числа; система счисления, в которую мы хотим преобразовать число. Числа должны поступать в виде строки с некоторым разделителем; в массиве; списком чисел.
6. Вывести уравнение прямой, проходящей через две точки. На вход поступает 4 числа: x1, y1, x2, y2. Числа должны поступать в виде строки с некоторым разделителем; в массиве; списком чисел.
7. Дан текст. Определите процентное отношение строчных и прописных букв к общему числу символов в нем. На вход поступает текст в виде строки.

8. Дана строка, состоящая из слов и чисел, отделенных друг от друга разделяющим символом (при этом один символ будет служебным, например «.», который будет характеризовать вещественные числа). Сформировать три строки, одна из которых содержит только целые числа, встречающиеся в исходной строке, вторая — только вещественные числа, а третья — оставшиеся слова. Текст должен поступать сплошным текстом с разделителем.
9. Подсчет одинаковых символов. Символы должны поступать сплошным текстом с разделителем; списком; отдельными значениями данных.
10. Наибольший общий делитель (НОД) чисел (Например: 3430 и 1365 — это 35. Другими словами, 35 — наибольшее число, на которое и 3430 и 1365 делятся без остатка). Числа должны поступать в виде строки с некоторым разделителем; в массиве; списком чисел.
11. Для каждого натурального числа в промежутке от m до n вывести все делители. Числа должны поступать в виде строки с некоторым разделителем; в массиве; отдельными значениями данных.
12. Перевод римских чисел в арабские (например XIIV — 13). Поступает в виде строки.
13. Калькулятор. Формула должна поступать в виде в виде текста (пример: «4/2»); отдельными значениями данных.
14. Перемножение матриц. Числа должны поступать в виде строки с некоторым разделителем; в массиве; отдельными значениями данных.
15. Пользователь вводит несколько целых чисел, представляющих собой последовательность. Требуется ее оценить: является ли последовательность возрастающей; есть ли в ней одинаковые элементы; является ли она знакочередующейся (положительные и отрицательные числа чередуются). Числа должны поступать в виде строки с некоторым разделителем; в массиве; отдельными значениями данных.

#### 2.1.3.3. Вторая часть лабораторной работы

1. Используя класс java.net.ServerSocket, написать простейший echo-сервер. Технические требования, предъявляемые к серверу:

1.1. Порт, на котором запускается сервер — 12345. Транспортный протокол — TCP.
1.2. Количество одновременно подключенных клиентов — 1. То есть использование потоков (экземпляров класса Thread) на первом этапе не предусматривается.
1.3. Сервер должен обеспечивать echo-функционал, то есть просто передавать обратно клиенту полученный буквенный символ.
1.4. На консоль (с помощью метода System.out.println ()) должны выводиться основные этапы работы программы (клиент установил соединение, был получен и передан обратно символ, соединение разорвано).
2. Изменить написанный класс echo-сервера. Сервер должен производить вычисления в соответствии с индивидуальным заданием, выполненным в пункте 2.1.3.2. В ответ клиент должен получить результат выполненной операции.
3. Написать клиентскую часть Socket:
3.1. Клиент должен будет подключиться к созданному ранее серверу.
3.2. Клиенту нужно передать данные параметров в соответствии с индивидуальным заданием, в ответ получить результат действий.
4. Произвести сборку серверной и клиентской части в jar файл.
5. Запустить jar-файлы серверной и клиентской части.

## 2.2. Лабораторная работа № 2

### 2.2.1. ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является ознакомление с HTTP-протоколом и разработка HTTP-сервера, с использованием наработок, сделанных в прошлой лабораторной работе.

## 2.2.2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

### 2.2.2.1. Hyper Text Transfer Protocol

Для создания интернет-приложений все чаще и чаще используют протокол прикладного уровня (модели OSI), который готовит компьютеры к обмену информацией: Hypertext Transfer Protocol, или HTTP. HTTP — протокол прикладного уровня поверх коммуникационного протокола TCP/IP. Изначально HTTP рассматривался как протокол передачи гипертекста, сейчас он широко используется и для передачи файлов.

HTTP оперирует запросами, на сайтах вы встретите чаще всего только два основных запроса:

— GET — запрос документа. Наиболее часто употребляемый метод; в HTTP/0.9 он был единственным.

— POST — этот метод применяется для передачи данных CGI-скриптам. Сами данные идут в следующих строках запроса в виде параметров.

У всех запросов (Request) и ответов (Response) используется одна и та же структура: строка запроса/ответа, заголовки, пустая разделительная строка и тело сообщения. При этом обязательной является только строка запроса.

#### 2.2.2.1.1. Структура HTTP-запросов и ответов

Посмотрим структуру входящего GET-запроса на сервер и разберем его (табл. 2–3):

*Таблица 2*

**Пример GET-запроса страницы сайта yandex.ru**

| Строка запроса (Request Line) | 1. GET/index.html HTTP/1.1 |
|---|---|
| Заголовки (Message Headers) | 2. Host: yandex.ru |
| | 3. Connection: keep-alive |
| | 4. Cache-Control: max-age=0 |
| | 5. Accept: text/html, application/xhtml+xml, application/xml; q=0.9, image/webp,*/*; q=0.8 |
| | 6. User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.124 Safari/537.36 |

| | |
|---|---|
| Заголовки (Message Headers) | 7. Referer: http://yandex.ru/… |
| | 8. Accept-Encoding: gzip, deflate, sdch |
| | 9. Accept-Language: ru-RU, ru; q=0.8, en-US; q=0.6, en; q=0.4 |
| | 10. Cookie: z=… |
| Пустая строка разделитель | |
| Тело сообщения (EntityBody) — необязательный параметр | Например: файл |

*Таблица 3*

**Пример GET-ответа сайта yandex.ru**

| | |
|---|---|
| Строка ответа (Response Line) | 1. HTTP/1.1 200 OK |
| Заголовки (Message Headers) | 2. Server: nginx |
| | 3. Date: Mon, 29 Sep 2014 08:04:48 GMT |
| | 4. Connection: close |
| | 5. Cache-Control: no-cache, no-store, max-age=0, must-revalidate |
| | 6. Content-Length: 5073 |
| | 7. Content-Type: text/html; charset=utf-8 |
| | 8. Content-Encoding: gzip |
| Пустая строка разделитель | |
| Тело сообщения (EntityBody) — необязательный параметр | HTML-документ |

*Строка запроса (Request Line)*

Указывает метод передачи, URL-адрес, к которому нужно обратиться, и версию протокола HTTP. В примере используется метод GET, адрес index.html, что позволяет серверу определить требуемый ресурс клиенту и версию HTTP.

*Строка ответа (Response Line):*

Указывает версию HTTP и код состояния запроса.

Коды состояния запроса HTTP

200 OK — запрос был получен и обработан

301 Ресурс перемещен постоянно
302 Ресурс перемещен временно
400 Неверный запрос — сообщение с запросом имеет некорректный формат
401 Несанкционированный доступ — у пользователя нет прав для доступа к запрошенному документу
402 Ресурс доступен за плату
408 Тайм-аут запроса
500 Внутренняя ошибка сервера — ошибка помешала HTTP-серверу обработать запрос

#### 2.2.2.1.2. Заголовки HTTP-запросов и -ответов

*Заголовки (Message Headers)*

*Host*: имя хоста, на который производится запрос, необходимо в ситуациях, когда на сервере имеется несколько виртуальных серверов под одним IP-адресом. В этом случае имя виртуального сервера определяется по этому полю.

*Connection*: может принимать значения Keep-Alive и close. Keep-Alive (*оставить в живых*) означает, что после выдачи данного документа соединение с сервером не разрывается, и можно выдавать еще запросы. Большинство браузеров работает именно в режиме Keep-Alive, так как он позволяет за одно соединение с сервером *скачать* html-страницу и рисунки к ней. Будучи однажды установленным, режим Keep-Alive сохраняется до первой ошибки или до явного указания в очередном запросе Connection: close. close (*закрыть*) — соединение закрывается после ответа на данный запрос.

*Cache-Control*: срок годности содержимого в секундах:

— *no-cache*. Сервер не должен использовать кэшированный ответ.
— *no-store*. Ответ на этот запрос не должен кэшироваться.
— *max-age*=delta-seconds. Клиент допускает кэшированный ответ, если его возраст не превышает delta-seconds секунд; клиент не требует его валидации.
— *max-stale*=delta-seconds. Клиент допускает кэшированный ответ, если его возраст не превышает delta-seconds секунд.
— *min-fresh*=delta-seconds. Клиент допускает кэшированный ответ, если он будет оставаться действительным не менее delta-seconds секунд от момента запроса.

— *no-transform*. К запрашиваемому документу не должны применяться преобразования.
— *only-if-cached*. Допускается только кэшированный ответ. Если подходящего ответа нет в кэше, то не нужна ни валидация старого ответа, ни получение нового.

*Accept*: список поддерживаемых браузером типов содержимого в порядке их предпочтения данным браузером, например, для Google Chrome она будет выглядеть следующим образом:

text/html, application/xhtml+xml, application/xml; q=0.9, image/webp,*/*; q=0.8

Эти данные необходимы тогда, когда сервер может выдавать один и тот же документ в разных форматах.

*User-Agent* — значением является «кодовое обозначение» браузера.

Например: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.124 Safari/537.36

Этот заголовок несет в себе несколько видов информации:

— название браузера и его версию;
— название операционной системы и ее версию;
— язык системы по умолчанию.

Referer: Позволяет клиенту указать адрес (URI) ресурса, из которого получен запрашиваемый URI. Этот заголовок дает возможность серверу сгенерировать список обратных ссылок на ресурсы для будущего анализа, регистрации, оптимизированного кэширования и т. д. Он также позволяет прослеживать с целью последующего исправления устаревшие или введенные с ошибками ссылки.

*Accept-Encoding*: Большинство современных браузеров поддерживает метод сжатия Gzip, о чем они и сообщают в заголовке Accept-Encoding. Сервер может отправить страницу в сжатом виде. При этом объем трафика может уменьшиться до 80 %, что довольно неплохо разгружает интернет-канал сайтов.

*Server*: Информация о программном обеспечении сервера, отвечающего на запрос (это может быть как веб-, так и прокси-сервер).

*Date*: Дата и время формирования сообщения.

*Accept-Language*: поддерживаемый язык. Имеет значение для сервера, который может выдавать один и тот же документ в разных языковых версиях.

*Content-Type*: MIME тип возвращаемого документа.

*Content-Length*: размер возвращаемого документа.

Пустая строка-разделитель необходима для разделения частей сообщения *заголовков* и *тела сообщения*.

*Тело сообщения* (*EntityBody*) может содержать в себе любую информацию которую мы хотим передать на сервер, например, изображение, аудио, видео, файл, html и др.

#### 2.2.2.2. Управление потоками

В настоящее время пользователей интернета становится все больше и больше, и все больше вероятность того, что одновременно к серверу может обратиться несколько клиентов. Для решения данной задачи используются потоки, которые позволяют параллельно обрабатывать нескольких клиентов.

Потоки — это независимые последовательности операций. Существует два пути создания потока. Первый — наследование от класса java.lang.Thread и переопределение его метода run. Второй — реализация интерфейса java.lang.Runnable и создание потока на основе этой реализации. В принципе, эти методы эквивалентны, разница в деталях. Наследование от java.lang.Thread делает его единственным родителем класса, что не всегда удобно.

Таким образом, простейший поток может быть реализован так:

```
1.  public class SampleThread extends Thread {
2.  Integer counter = 0;
3.  public SampleThread (Integer integer) {
4.  counter=integer;
5.  }
6.  @Override
7.  public void run () {
8.  Integer result = counter*counter;
9.  System.out.println (result);
10. }
11. public static void main (String [] args) {
12. for (int i = 10; i > 0; i —) {
13. new Thread (new SampleThread (i)).start ();
14. }
15. }
16. }
```

При этом результат всегда будет выводиться в разной очередности, например, 81;100;64;49;36;4;16;9;25;1.

### 2.2.3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Технические требования к разрабатываемому серверу.

1. Порт, на котором запускается сервер — 8080.
2. Количество одновременно обрабатываемых клиентских запросов — не ограничено (создание многопоточности).
3. Сервер должен распознавать метод запроса и реагировать только на метод GET. Реакция на все остальные методы (POST, PUT, DELETE и др.) не оговаривается и может быть реализована по желанию, но при этом, если будет реализован только один метод GET, на другие методы ваш сервер не должен срабатывать.
4. В заголовке выдаваемого ответа нужно указать корректный тип (text/html) и длину тела сформированного сообщения.
5. В ответ на любой запрошенный ресурс сервер должен выдавать HTML-страницу с возможностью отображения русских слов.

*Примечание к выполнению работы*

1. Написать процедуру распознавания окончания запроса (наличия пустой строки, в которой есть символ возврата каретки, но нет ни одного значащего текстового символа).
2. По окончании запроса провести анализ полученного метода. Если это GET — перейти к формированию корректного ответа клиенту.
3. Сформировать ответ и выдать его клиенту. Корректно завершить соединение с клиентом. При следующем запросе клиент установит новое соединение.
4. Должен быть предусмотрен механизм обработки кодировка входящего/исходящего потока данных.

*Результат выполнения лабораторной работы*

1. В ответ на любой запрос к серверу ответ должен содержать одну и ту же HTML-страницу, состоящую из надписи:
   Работу выполняли: Иванов Иван Иванович

Номер группы: РИ-202201
Номер индивидуального задания: 3
Текст индивидуального задания:
«Создание калькулятора…»

2. Сервер должен работать по принципу echo-сервера, то есть то, что мы запрашиваем в адресной строке, то и отображаем на странице. Ответ должен приходить в виде HTML-страницы и должен содержать заполненные данные предыдущего этапа работы и сформированный ответ адресной строки, при этом ответ не должен содержать адреса сервера. Пример сформированного ответа адресной строки:
Запрос: http://localhost:8080/display_the_entered_value
Ответ: HTML-страница: <h1>display the entered value</h1>
3. Сервер должен принимать данные и выполнять вычисления в соответствии с индивидуальным заданием (лабораторная работа № 1), и отправлять результат клиенту в виде HTML-страницы. HTML-страница должна содержать ранее выполненные этапы лабораторной работы, а также информацию о произведенных вычислениях. Пример: при выполнении лабораторной работы было выбрано индивидуальное задание № 1 «Найти максимальное значение среди множества чисел…», значит, у нас в адресную строку в качестве ресурса должна быть введена цепочка чисел через разделительный символ: «11, 32, 1, 22, 14», сервер должен разбить строку, произвести вычисления и вернуть результат клиенту (браузеру).
4. Произвести сборку HTTP-сервера в jar-файл и проверку работоспособности jar-файла.

## 2.3. Лабораторная работа № 3

### 2.3.1. ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является знакомство с современными подходами создания клиентской части интернет-приложения и разработка простого web-сайта, основанного на наработках предыдущей лабораторной работы.

## 2.3.2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

### 2.3.2.1. Создание клиентской части

В первой части учебно-методического пособия было кратко рассказано о том, с помощью каких технологий возможно оформлять клиентскую часть. Далее будет описан процесс взаимодействия разных технологий на примерах, таких, как HTML, CSS, JavaScript, JQuery, AJAX. Основным, конечно же является HTML, то есть сам наш документ, а CSS и JavaScript — вспомогательные аппараты для упрощения оформления и интерактивности страницы, AJAX же позволяет обмениваться информацией с сервером асинхронными запросами.

Объединяет все эти технологии HTML, то есть ссылки на них (css-файлы и js-файлы) будут находиться в самом документе в виде следующих строк:

```
<script type="text/javascript" src="/js/example.js"></script>
<link rel="stylesheet" type="text/css" href="/css/example.css">
```

В фалах с расширением js обычно содержатся код JavaScript и элементы AJAX и JQuery.

В файлах с расширением css содержатся таблицы каскадных стилей.

Приведем простой пример применения этих технологий на практике. Пример HTML-файла:

```
1. <html>
2. <head>
3. <meta http-equiv="Content-Type" content="text/html;
   charset=utf-8">
4. <script type="text/javascript" src="jquery-1.6.2.min.
   js"></script>
5. <script type="text/javascript" src="example.js"></script>
6. <link rel="stylesheet" type="text/css" href="example.css">
7. <title>HardSite</title>
8. </head>
9. <body>
10. <input type="text" id="text" name="text" class="panda-search-
    field">
11. <div class="b-top"><input type="submit" value="Перейти"
    onclick="sendTextJQuery()" class="b-top-but"></div>
12. <div class="b-top"><input type="submit" value="Перейти"
    onclick="sendText()" class="b-top-but"></div>
```

```
13. </body>
14. </html>
```

На 4–5 строках листинга представлено подключение js-файлов. Один из них — это библиотека JQuery, другой — это созданный файл. Строка 6 содержит подключение файла — таблицы каскадных стилей. Изначально, когда обращаются в адресной строке, к определенному сайту, отправляется HTTP get-запрос, после принятия от сервера HTML-документа браузер смотрит на информацию, отмеченную в теге HEAD, там обычно указывается служебная информация, по ней браузер понимает, какие еще файлы необходимо запросить от сервера, и будет поочередно отправлять запросы на сервер, требуя сначала файл jquery-1.6.2.min.js, затем example.js и заканчивая файлом каскадных стилей example.css. Изображения, которые могут использоваться в HTML-документе, также будут запрашиваться у сервера отдельными запросами.

#### 2.3.2.2. Таблицы каскадных стилей

Существует три способа добавления стилей в документ.

1. *Внутренние стили* определяются атрибутом style в тегах.
   Пример: Абзац с текстом синего цвета
2. *Глобальные стили* располагаются в контейнере <style>…</style>, который в свою очередь находится в теге <head>…</head>.
   Пример:
   ```
   1. <html>
   2. <head>
   3. … ….
   4. <style type="text/css">
   5. p {color:#808080;}
   6. </style>
   7. </head>
   8. <body>
   9. <p>Серый цвет текста во всех абзацах Web-страницы</p>
   10. <p>Серый цвет текста во всех абзацах Web-страницы</p>
   11. </body>
   12. </html>
   ```
3. *Внешние стили* содержатся в отдельном файле с расширением css. Внешние стили позволяют всем страницам сайта использо-

вать одни и те же стили, тем самым позволяя страницам выглядеть единообразно.

Для связи с файлом стилей используется тег <link>, расположенный в теге <head>…</head>. В нем задается два атрибута: rel=«stylesheet» и href, определяющий адрес файла стилей.

Пример:

```
1. <html>
2. <head>
3. … ….
4. <link rel="stylesheet" href="style.css">
5. … ….
6. </head>
7. <body>
8. … ….
9. </body>
10. </html>
```

*Подключение стилей*

Правило подключения глобальных и внешних стилей состоит из селектора и объявлений стиля.

Селектор, расположенный в левой части правила, определяет элемент (элементы), для которых установлено правило. Далее в фигурных скобках перечисляются объявления стиля, разделенные точкой с запятой. Например:

```
1. p {
2. text-indent: 30px;
3. font-size: 14px;
4. color: #666;
5. }
```

Объявление стиля — это пара свойство CSS: значение CSS.

Например: color: red

color  свойство CSS, определяющее цвет текста;

red  значение CSS, определяющее красный цвет.

*Существует несколько типов селекторов*: селекторы CSS, селекторы тегов, селекторы идентификаторов и селекторы классов.

*Селекторы тегов*

В качестве селектора может выступать любой html-тег, для которого определяются правила стилевого оформления. Например:

```
h1 {color: red; text-align: center;}
```

*Селекторы идентификаторов*

HTML предоставляет возможность присвоить уникальный идентификатор любому тегу. Идентификатор задается атрибутом id. Например:

```
<div id="a1">…</div>
```

В CSS-коде селектор идентификатора обозначается знаком «решетка» (#). Так как идентификатор id применяется только к уникальным элементам, название тега перед знаком «решетка» (#) обычно опускают. Пример:

```
#a1 {color: green;}
```

*Селекторы классов*

Для стилевого оформления чаще всего используются селекторы классов. Класс для тега задается атрибутом class. Например:

```
<div class="c1">…</div>
```

Если атрибут id применяется для уникальной идентификации, то при помощи атрибута class тег относят к той или иной группе.

В CSS-коде селектор класса обозначается знаком «точка» (.). Разные теги можно отнести к одному классу. В таком случае имя тега перед знаком «точка» (.) опускают:

```
i.green {color: #008000;}
b.red {color: #f00;}
.blue {color: #00f;}
```

Для тега можно одновременно указать несколько классов, перечисляя их в атрибуте class через пробел. В этом случае к элементу применяются стили каждого из указанных классов.

```
<div class="left w100">…</div>
```

Если некоторые из этих классов содержат одинаковые свойства стиля, но с разными значениями, то будут применены значения стиля класса, который в CSS-коде расположен ниже.

*Селекторы CSS*

Имеется набор стандартных селекторов, предоставляемый css. Например: E: focus; E: hover и т. д. Также он позволяет воспринимать структуры вида E#myid и E.myclass, которые позволяют более точно настроить стили.

E#myid позволяет применить стили к элементу с id равным myid который обязательно должен находиться в теге E.

E.myclass позволяет применить стили к элементам класса myclass, но только те, которые находятся в теге E.

### 2.3.2.3. Отправка HTTP-запросов с помощью языка JavaScript

JavaScript позволяет разработчикам:

— изменять страницу, писать на ней текст, добавлять и удалять теги, менять стили элементов;
— реагировать на события: скрипт может ждать, когда что-нибудь случится (клик мыши, окончание загрузки страницы) и реагировать на это выполнением функции;
— выполнять запросы к серверу и загружать данные без перезагрузки страницы;
— устанавливать и считывать cookie, валидировать данные, выводить сообщения и многое другое.

Основное внимание в нашей работе уделяется взаимодействию клиента с сервером, в связи с этим разберем пример отправки и получения данных серверу.

В данном примере будет показано, как можно отправить POST-запрос на сервер с помощью JavaScript.

Пример использования объекта XMLHttpRequest:

```
1. function sendText () {
2. var text = document.getElementById("text").value;
3. var xmlhttp = new XMLHttpRequest();
4. xmlhttp.open ('POST', '/xhr/test.html', false);
5. xmlhttp.send(text);
6. xmlhttp.onreadystatechange = function(e) {
7. if (this.readyState == 4 && this.status == 200) {
8. ...
9. }
10. }
11. }
```

Пример использования объекта XMLHttpRequest:

```
1. function sendTextJQuery ()
2. {
3. var text = $ ("#text").val ();
4. $.ajax ({
5. url: "/form.php",
6. type: "post",//тип запроса
7. data: text//отправляемые данные, в данный момент это простой текст
8. success: function (data, textStatus) {
```

```
9.  ...
10. }
11. });
12. }
```

Обращение к объектам документа можно производить через команду document.getElementById, при этом будет возвращаться тот элемент, которому соответствует указанный идентификатор, и в дальнейшем можно при желании извлечь из него данные для последующей обработки.

В листинге показаны способы отправления запросов с данными на сервер с помощью компонента XMLHttpRequest и AJAX.

Функция sendText осуществляет общение с сервером через объект XMLHttpRequest. Для того что бы с ней работать, необходимо описать с кем и как необходимо соединиться, это позволяет сделать метод open с входными параметрами: тип запроса, url, асинхронной или синхронный тип запроса. После чего нужно послать сформированный HTTP-запрос, делает это функция send, входным параметром являются данные, которые необходимо передать серверу. Далее необходимо ожидать ответа от сервера, это позволяет осуществлять метод onreadystatechange.

Функция sendTextJQuery отправляет данные серверу с помощью технологий AJAX. Выполняется это функцией $.ajax, в которой указываются необходимые параметры: url, тип запроса и данные. Функция success позволяет получать ответ от сервера, который в дальнейшем необходимо обрабатывать.

XMLHttpRequest — это объект JavaScript, содержащий API, который обеспечивает клиентский скрипт функциональностью для обмена данными между клиентом и сервером. Может работать как асинхронно, так и синхронно с серверным приложением. Использует запросы HTTP или HTTPS напрямую к серверу и загружает данные ответа сервера напрямую в вызывающий скрипт. Позволяет осуществлять HTTP-запросы к серверу без перезагрузки страницы.

В основе работы AJAX лежит объект XMLHttpRequest. Только в отличие от XMLHttpRequest требует еще и библиотеку JQuery, но и имеет более гибкий функционал и более простое взаимодействие с сервером.

### 2.3.3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Описание выполнения работы:

1. Описание HTML-страниц (не менее трех страниц):

1.1. Первая страница «О себе» содержит информацию:

Работу выполняли: Иванов Иван Иванович

Номер группы: РИ-202201

Номер индивидуального задания: 3

Текст индивидуального задания:

«Создание калькулятора…»

1.2. Вторая страница — реализация индивидуального задания. На странице отображены следующие элементы:

1.2.1. Поля для занесения информации, необходимой для вычисления.

1.2.2. Кнопка для отправки результатов на сервер.

1.2.3. Поле, в котором будет выводиться результат вычислений.

1.3. Третья страница — работа с таблицей, студент выбирает некоторую узкую тему, с которой он дальше будет работать. Например, таблица студенты. На странице отображается информация о студентах, при этом дана возможность добавлять нового студента, удалять и изменять данные уже существующего студента.

2. Изменения сервера в соответствии с требованиями.

2.1. Сервер должен обрабатывать запросы и в соответствии с ним возвращать необходимую клиенту информацию.

2.2. От клиента будут запросы на загрузку данных о css, js и других файлах, используемых страницей. Пример: все страницы используют файл каскадных стилей (css), в HTML-документе ссылка на него. При загрузке страницы браузером он обнаруживает, что данная страница использует css-файл и начинает автоматически требовать его с сервера. Сервер должен в итоге отправить ему этот файл.

2.3. Для сохранения данных третьей страницы необходимо:

2.3.1. Создать структуру таблицы. Например, таблица студенты содержит информацию: Имя, Фамилия, Отчество, Год рождения. Для этого мы создаем новый класс под названием Student с описанием полей Id, Name, LastName, FirstName, YearBirth и их геттеры и сеттеры (getName (), setName ()…).

2.3.2. Для хранения данных создаем список со структурой таблицы (Пример: List<Student>). Сохранение в файл или БД не требует-

ся. То есть при каждом перезапуске сервера у нас будут обнуляться записи.

3. Технические требования:

3.1. Требования к серверу:

3.1.1. Все страницы должны храниться как html-страницы на сервере.

3.1.2. При первом подключении к серверу должна отобразиться первая HTML-страница, или если в адресной строке указан какой-то определенный ресурс, необходимо вывести информацию о той странице, которая соответствует данному запросу. При указании несуществующего ресурса система должна выводить первую HTML-страницу.

3.1.3. Все поля ввода информации должны иметь проверку входных значений (с помощью JavaScript).

3.2. Требования к оформлению HTML-страниц:

3.2.1. Общие требования:

3.2.1.1. У каждой группы студентов должен быть уникальный стиль сайта и уникальная тема для третьей страницы.

3.2.1.2. Все страницы должны быть красиво оформлены, стили описаны в отдельном css-файле.

3.2.1.3. У каждой страницы должно быть реализовано единое меню, у которого будет минимум три ссылки (переход на каждую созданную страницу).

3.2.2. Индивидуальные требования к страницам:

3.2.2.1. Первая страница: вывод статической информации.

3.2.2.2. Вторая страница: для выполнения вычислений мы должны использовать возможность отправки данных на сервер POST-запросом и получать от сервера ответ, который должен помещаться в специально отведенное для этого поле. Данная функция должна быть реализована с помощью JavaScript и храниться отдельном js-файле.

3.2.2.3. Третья страница: при запуске сервера и при обращении к этой странице у нас будет отображаться пустая таблица (то есть сервер сохраненные данные не хранит постоянно, а только временно держит их в оперативной памяти). Данная страница должна позволять:

— добавлять строки и сохранять их на сервере;

— изменять строки и сохранять изменение на сервере;

— удалять строки и удалять информацию с сервера.

3.2.2.3.1. Операции удаления, добавления и изменения должны быть реализованы с помощью JavaScript, данные операции должны

быть реализованы на одной странице, действие каждой команды должно отправляться на сервер.

3.2.2.3.2. Операция удаления: напротив каждой строки должен быть значок удаления, после нажатия на который строка должна пропасть (удалиться и со страницы и с сервера).

3.2.2.3.3. Операция создания: для создания должна быть отдельная кнопка, по нажатию на которую будет блокироваться основное окно с таблицей и появляться другое, в котором будут описаны поля, необходимые для создания новой строки, и кнопка *сохранить*, по нажатию на которую произведется добавление на сервер и на страницу.

3.2.2.3.4. Операция изменения: напротив каждой строки должен быть значок, нажатие которого должно работать по аналогии операции создания. Только все поля должны быть заполнены данными изменяемой строки.

## 2.4. Лабораторная работа № 4

### 2.4.1. ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является ознакомление с технологиями ORM и реализация взаимодействия с базой данных посредством ORM Hibernate.

### 2.4.2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Краткое введение в технологии ORM было осуществлено в первой части учебно-методического пособия. Существует множество реализаций технологии ORM на языке Java. Одна из этих реализаций — это Hibernate. Для того, чтобы использовать функционал hibernate, необходимо иметь:

— библиотеку ORM Hibernate, ее можно скачать с официального сайта производителя;
— JDBC (Java Data Base Connectivity) — это библиотека, позволяющая взаимодействие с различными СУБД. JDBC основан на концепции драйверов, позволяющих получить соедине-

ние с БД по специальному URL. Каждая БД имеет свою JDBC-библиотеку, которую можно загрузить с официального сайта используемой БД.

#### 2.4.2.1. Создание конфигурации подключения к БД.

Основным классом hibernate-технологии, позволяющей читать, изменять, добавлять или удалять данные из СУБД, является Session. Для получения сессии необходимо изначально установить конфигурацию соединения с БД и получить с помощью созданной конфигурации фабрику сессий (SessionFactory), которая в итоге будет нам поставлять сессии. Конфигурацию соединения можно прописать двумя способами:

— через файл hibernate.cfg.xml;
— через класс org.hibernate.cfg.Configuration.

Сейчас рассмотрим эти два способа на примерах:

Файл hibernate.cfg.xml

```
1. <! DOCTYPE hibernate-configuration SYSTEM
2. "http://www.hibernate.org/dtd/hibernate-
   configuration-3.0.dtd">
3. <hibernate-configuration>
4. <session-factory>
5. <property name="connection.url">jdbc:
   sqlserver://localhost:1433; database=data_base</property>
6. <property name="connection.driver_class">com.microsoft.
   sqlserver.jdbc.SQLServerDriver</property>
7. <property name="connection.username">sa</property>
8. <property name="connection.password">1</property>
9. <property name="connection.pool_size">1</property>
10. <property name="current_session_context_
    class">thread</property>
11. <property name="hbm2ddl.auto">update</property>
12. <property name="show_sql">true</property>
13. <property name="dialect">org.hibernate.dialect.
    SQLServerDialect</property>
14. </session-factory>
15. </hibernate-configuration>
```

Первые две строчки — подключение спецификации (шаблона), с его помощью мы не ошибемся в синтаксисе и будем точно знать, где какой тег должен будет находиться и какие атрибуты он может иметь.

Необходимая информация содержится в строчках с 5 по 13, в них прописаны непосредственно свойства подключения в БД:

— connection.url — url-строка подключения к БД. В данном примере осуществляется подключение к MS SQL Server. Разберем строчку детально: jdbc: sqlserver: — это некоторая спецификация, через что мы подключаемся и к какому серверу;://localhost:1433 — указываем ip-адресс или доменное имя компьютера с портом, к которому мы будем подключаться; database=data_base — указание, к какой базе мы будем подключаться;
— connection.driver_class — указание, какой драйвер будет использоваться в качестве связующего звена с СУБД. Здесь указывается класс JDBC-драйвера, который подключен к проекту, в примере указан драйвер MS SQL Server;
— connection.username — логин зарегистрированного пользователя СУБД;
— connection.password — пароль;
— connection.pool_size — данное свойство показывает, сколько соединений к БД может быть одновременно открыто;
— hbm2ddl.auto — свойство, указывает что нужно сделать со схемой БД при инициализации. Может принимать такие значения:
  - update — сверяет схему БД с имеющимися конфигурациями классов, если были внесены какие-то изменения, они автоматически занесутся в БД. При этом данные, которые были занесены в базу, не изменятся;
  - create — каждый раз при запуске приложения схема БД будет создаваться заново. Все данные, которые были занесены раньше, будут удалены;
  - create-drop — каждый раз при запуске приложения схема БД будет создаваться заново, а при завершении — удаляться. Все данные, которые были занесены во время работы приложения, будут удалены по завершению приложения;
  - validate — при инициализации будет совершена проверка соответствия конфигурации классов и схемы БД. Если мы внесли изменение в конфигурацию какого-то класса, а схема в БД не была изменена, сработает исключение;
— show_sql — данное свойство указывает, будут ли выводится SQL-запросы на консоль, которые отправляются на СУБД. В процессе отладки это бывает очень удобно;

— dialect — hibernate может работать с разными БД, и каждая имеет свои особенности (генерация первичного ключа, страничный вывод, функции), нужно указать какой диалект будем использовать для создания запросов. В данном случае — диалект MS SQL Server.

Класс org.hibernate.cfg.Configuration используется для подготовки данных соединения к СУБД.

```
1. public static Configuration getConfiguration (){
2. Configuration cfg = new Configuration ();
3. cfg.setProperty ("hibernate.connection.url", "jdbc:
   sqlserver://localhost:1433; database=data_base");
4. cfg.setProperty ("hibernate.connection.driver_class", "com.
   microsoft.sqlserver.jdbc.SQLServerDriver");
5. cfg.setProperty ("hibernate.connection.username","sa");
6. cfg.setProperty ("hibernate.connection.password","1");
7. cfg.setProperty ("hibernate.connection.pool_size","1");
8. cfg.setProperty ("hibernate.current_session_context_
   class","thread");
9. cfg.setProperty ("hibernate.hbm2ddl.auto","update");
10. cfg.setProperty ("hibernate.show_sql","true");
11. cfg.setProperty ("hibernate.dialect","org.hibernate.dialect.
    SQLServerDialect");
12. return cfg;
13. }
```

Класс подключения к БД.

Когда создана конфигурация нужно создать класс отвечающий за создание сессий, так называемая фабрика сессий. Пример кода:

```
1. private static SessionFactory sessionFactory = buildSessionFactory();
2. private static SessionFactory buildSessionFactory () {
3. try {
4. if (sessionFactory == null) {
5. Configuration configuration = new Configuration ().configure
   ("HibernateUtil/hibernate.cfg.xml");
6. StandardServiceRegistryBuilder builder = new
   StandardServiceRegistryBuilder ().applySettings (configuration.
   getProperties ());
7. sessionFactory = configuration.buildSessionFactory (builder.
   build());
```

```
8.  }
9.  return sessionFactory;
10. } catch (Throwable ex) {
11. System.err.println ("Initial SessionFactory creation failed." + ex);
12. throw new ExceptionInInitializerError (ex);
13. }
14. }
15. public static Session getSession () {
16. return sessionFactory.openSession ();
17. }
```

В данном примере показано, как можно создать сессию с БД, используя ранее созданные конфигурационные параметры. В пятой строчке происходит использование конфигурационных данных, созданных с помощью файла hibernate.cfg.xml. Для реализации второго способа описания конфигурации можно использовать следующую строчку:

```
5.  Configuration configuration = getConfiguration();
```

Метод getSession создает сессию, с помощью которой и будет происходить общение с БД.

#### 2.4.2.2. Создание класса-сущности

БД позволяет хранить информацию в таблицах, для того чтобы это осуществить, у hibernate существует специальный механизм — mapping. Mapping-проецирование (сопоставление) Java классов с таблицами базы данных. Реализация маппинга возможна через использование конфигурационных файлов XML, либо через аннотации. Так как описание маппинга через файлы XML неудобно и уже устаревает, в примере использованы аннотации. Пример:

```
1.  import javax.persistence.*;
2.  @Entity
3.  @Table (name = "people")
4.  public class People {
5.  @Id
6.  @GeneratedValue (strategy = GenerationType.IDENTITY)
7.  @Column (name = "id", unique = true, nullable = false)
8.  private Integer id;
9.  @Column (name = "firstName")
10. private String firstName;
```

```
11. @Column (name = "lastName")
12. private String lastName;
13. @Column (name = "middleName")
14. private String middleName;
15. @Column ()
16. private int year;
17.
18. public Integer getId () {
19. return id;
20. }
21. public void setId (Integer id) {
22. this.id = id;
23. }
24. …
25. }
```

Для того чтобы созданный класс воспринимался hibernate как некоторая сущность бизнес-модели, необходимо пометить аннотацией @Entity.

Аннотация @Table позволяет задать имя, под которым данная сущность будет существовать в БД (позволяет задать имя таблицы), а также другие конфигурационные параметры. Если имя таблицы совпадает с именем класса, аннотацию можно не использовать.

В каждой сущности должен быть указан первичный ключ, который помечается @Id. Аннотация @GeneratedValue указывает, что данное поле будет генерироваться автоматически при создании нового элемента в таблице.

@Column — один из основных аннотаций, который указывает, что поле, описанное после него, будет храниться в БД.

Поле, над которым стоит аннотация Id, GeneratedValue, Column, должно быть либо примитивом, либо оберткой над этим примитивом: String; java.util.Date; java.sql.Date; java.math.BigDecimal; java.math.BigInteger.

По правилам хорошего тона класс нужно оформлять как просто POJO-объект, то есть поля должны иметь спецификатор доступа private. А доступ к этим полям осуществляется с помощью геттеров и сеттеров, как показано в примере, строчкиа 18–23.

В БД данная сущность будет выглядеть следующим образом (рис. 2.1).


Рис. 2.1. Структура класса-сущности People в БД

#### 2.4.2.3. Регистрация классов-сущностей

Для того чтобы hibernate при создании SessionFactory учитывала новую сущность, ее необходимо указать в конфигурации. Пример для файла hibernate.cfg.xml:

```
1. <hibernate-configuration>
2. <session-factory>
3. …
4. <mapping class="Entity.People"/>
5. </session-factory>
6. </hibernate-configuration>
```

Тег mapping позволяет указать, какие классы-сущности мы должны учитывать при создании SessionFactory.

Пример для конфигурационного класса org.hibernate.cfg. Configuration:

```
1. Configuration cfg = new Configuration ();
2. cfg.addAnnotatedClass (People.class);
```

Взаимодействие с БД: создание, удаление, чтение и изменение сущностей.

#### 2.4.2.4. Создание объекта в БД

Пример создания объекта (строки) в БД.

```
1. public static void create (Object o)
2. {
3. try {
4. Session session = HibernateUtil.getSession ();
5. session.beginTransaction ();
6. session.save (o);
7. session.getTransaction ().commit ();
8. session.close ();
```

```
9.  } catch (HibernateException e) {
10. e.printStackTrace ();
11. }
```

В третьей строке мы обращаемся к созданному классу для получения сессии.

Четвертая строчка — начало транзакции, шестая — конец транзакции или просто коммит, то есть те запросы, которые были между этими командами описаны, отправятся на сервер БД, и там появятся данные.

Пятая строчка отвечает за генерацию кода, обеспечивающую создание объекта, при этом запрос будет выглядеть примерно следующим образом:

Hibernate: insert into people (firstName, lastName, middleName, year) values (?,?,?,?).

Пример сохранения объекта:

```
1. People people = new People ();
2. people.setFirstName ("Иванов");
3. people.setLastName ("Иван");
4. people.setMiddleName ("Иванович");
5. people.setYear (21);
6. create (people);
```

В БД появится запись, как показано на рис. 2.2.

Результаты | Сообщения

| | id | firstName | lastName | middleName | year |
|---|---|---|---|---|---|
| 1 | 4 | Иванов | Иван | Иванович | 21 |

Рис. 2.2. Результат добавления данных через функцию hibernate

### 2.4.2.5. Удаление объекта из БД.

Существует два распространенных способа удаления объектов из БД: через объект или через id.

Через объект метод будет в точности такой же, как и при создании, только вместо save будет delete:

```
5. session.beginTransaction ();
6. session.delete (o);
7. session.getTransaction ().commit ();
```

Удаление с помощью HQL-запросов по id:

```
1.  public static void delete (int id){
2.  try {
3.  Session session = HibernateUtil.getSession();
4.  String stringQuery = "delete from People where id =: id";
5.  Query query = session.createQuery (stringQuery);
6.  query.setParameter ("id", id);
7.  query.executeUpdate();
8.  session.close();
9.  } catch (HibernateException e) {
10. e.printStackTrace();
11. }
12. }
```

В примере прописывается строка HQL-запроса (Hibernate Query Language — язык запросов Hibernate), и на его основе формируется сам запрос, в который мы передаем параметры через метод setParameter. Метод executeUpdate непосредственно выполняет запрос, после его выполнения в БД будет произведено удаление объекта.

#### 2.4.2.6. Изменение объекта в БД

Изменение объекта ничем не отличается от метода создания объекта, только в место save будет использоваться метод update:

```
5.  session.beginTransaction ();
6.  session.update (o);
7.  session.getTransaction ().commit ();
```

#### 2.4.2.7. Чтение из БД

Есть два основных способа чтения данных из БД — это чтение списком и чтение по id. Для чтения из базы данных используется специальный вид запросов под названием Criteria. Пример вызова списка без фильтрации по параметрам:

```
List<People> peoples = session.createCriteria(People.class).list();
```

То есть нужно только указать сущность, из которой мы хотим извлечь данные и вызвать метод list, который вернет список всех объектов, находящихся в данной таблице.

Если же нам нужен поиск по идентификатору, то нужно использовать некоторую фильтрацию:

```
1. People people = (People) session.createCriteria (People.class)
2. .add (Restrictions.eq (“id”, id)).uniqueResult ();
```

Как и в прошлый раз, мы указываем, к какой таблице обращаемся. Дальше используется метод add, в который прописываются условия отбора, это сделать нам позволяет класс Restrictions. Метод же eq означает, что будет осуществляться сравнение с объектом, который мы в него поместим. Метод uniqueResult позволяет вернуть результат объектом, но у него есть и недостатки: если объект не будет найден или по каким-то причинам объектов будет больше одного, произойдет ошибка. Для обхода ошибки рекомендуется прописывать немного другой код:

```
1. People people = null;
2. List<People> peoples = session.createCriteria (People.class)
3. .add (Restrictions.eq ("id", id)).list();
4. if (peoples!=null&&peoples.size()==1)
5. {
6. people = peoples.get(0);
7. }
```

Весь результат будет собираться в список, если не будет найдено ни одного элемента или элементов списка будет больше 1, то if не сработает, но и не произойдет ошибки.

#### 2.4.2.8. Связи между таблицами

Обычно при создании проектов необходимо связывать таблицы между собой, это нужно для логической связи объектов между собой. Так как реляционная база данных представляет собой множество взаимосвязанных таблиц, то необходимо уделить отдельное внимание описанию отношений между сущностями с помощью технологии hibernate. Всего существует три основных типа связи: one-to-one, one-to-many, many-to-one; существует также связь many-to-many, но она может быть решена как специальной аннотацией @many-to-many, так и промежуточной таблицей.

#### 2.4.2.9. Связь many-to-one

Разберем пример. Есть сущность people, которая связана с phone, то есть две таблицы: люди и телефоны. У одного человека может быть несколько телефонов, но не наоборот. Для этого мы создаем связь many-to-one, при этом в таблице phone появится новое поле, ссылаю-

щееся на элемент таблицы people, в нем будет храниться идентификатор. Класс people был уже описан ранее, в данном случае он остается неизменным. Класс phone выглядит следующим образом:

```
1.  @Entity
2.  @Table (name = "phones")
3.  public class Phone {
4.  @Id
5.  @GeneratedValue (strategy = GenerationType.IDENTITY)
6.  @Column (name = "id")
7.  private int id;
8.  @Column (name = "number")
9.  private String number;
10. @ManyToOne (targetEntity = People.class, fetch = FetchType.
    LAZY)
11. @JoinColumn (name = "id_poeple")
12. private People people;
13. … (описываем геттеры и сеттер)
14. }
```

Аннотация @ManyToOne позволяет создать связь между двумя таблицами, в базе данных данная связь будет выглядеть, как показано на рис. 2.3. Также при использовании связи many-to-one автоматически создается вторичный ключ.

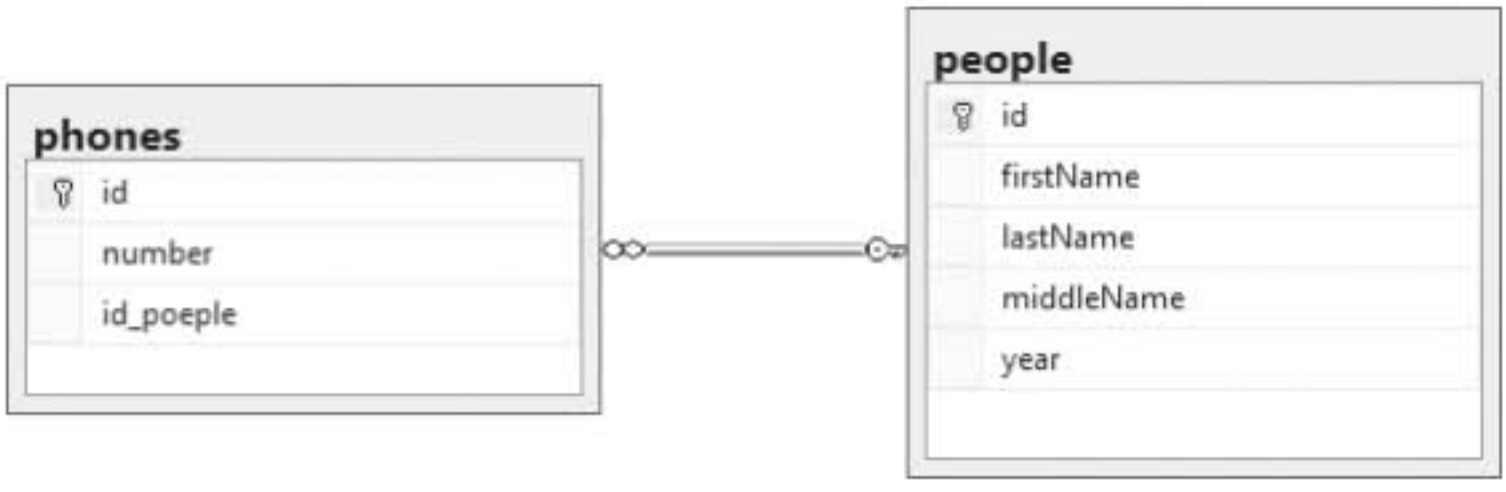


Рис. 2.3. Связь таблиц many-to-one

Свойство targetEntity указывает, с какой сущностью будет происходить соединение по внешнему ключу.

Свойство fetch (очень важное свойство) может иметь два состояния: LAZY или EAGER — по умолчанию это EAGER. В случае если па-

раметром будет выступать EAGER, связанный элемент будет сразу же подгружаться автоматически. LAZY — пока не обратятся к элементу people, данные не будут загружены.

Ниже показано, как будут формироваться запросы в зависимости от установленного свойства fetch.

При запросе к БД:

1. Phone phone = (Phone) session.createCriteria (Phone.class).add (Restrictions.eq ("id", 1)).uniqueResult ();
2. People people = phone.getPeople ();

Простой запрос на получение элемента таблицы *телефон* и связанного с ним человека.

fetch = FetchType.LAZY

При выполнении первой строчки будет сформирован запрос на получение данных только о таблице Phone, и только когда идет обращение к связанному объекту People, формируется новый запрос.

1. Hibernate: select this_.id as id1_1_0_, this_.number as number2_1_0_, this_.id_poeple as id_poepl3_1_0_ from phones this_ where this_.id=?
2. Hibernate: select people0_.id as id1_0_0_, people0_.firstName as firstNam2_0_0_, people0_.lastName as lastName3_0_0_, people0_.middleName as middleNa4_0_0_, people0_.year as year5_0_0_ from people people0_ where people0_.id=?

При этом важно понимать, что если сессия будет закрыта, а после этого будет обращение к объекту, который еще не подгружался, то произойдет ошибка. Для инициализации можно применить следующий код:

Hibernate.initialize (phone.getPeople());

fetch = FetchType.EAGER

При выполнении первой строчки кода будет формироваться один большой запрос, который автоматически будет подгружать связанного с данным телефоном человека.

Пример сформированного запроса:

Hibernate: select this_.id as id1_1_1_, this_.number as number2_1_1_, this_.id_poeple as id_poepl3_1_1_, people2_.id as id1_0_0_, people2_.firstName as firstNam2_0_0_, people2_.lastName as lastName3_0_0_, people2_.middleName as middleNa4_0_0_, people2_.year as year5_0_0_ from phones this_ left outer join people people2_ on this_.id_poeple=people2_.id where this_.id=?

Аннотация @JoinColumn в данном случае является не обязательной, она просто содержит параметр, в котором можно указать название колонки.

Связь one-to-many.

Часто требуется делать и обратную связь, то есть, имея объект people, получать список связанных с ним телефонов.

```
1. @Entity
2. @Table (name = "people")
3. public class People {
4. …
5. @OneToMany (targetEntity = Phone.class, mappedBy = "people",
   fetch = FetchType.LAZY)
6. private List<Phone> phones;
7. …
8. }
```

Аннотация @OneToMany позволяет осуществить эту мнимую связь, которая обеспечивается hibernate, но не БД. Параметр targetEntity указывает, с какой сущностью идет связь. В параметре mappedBy прописывается имя связующего поля с аннотацией @ManyToOne, то есть его имя people. Параметр fetch был ранее описан, в данном случае он играет ту же самую функцию.

### 2.4.3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Постановка задачи:

1. Разобраться с ORM-технологией и Hibernate библиотекой, а также с JDBC концепцией.

2. Опираясь на наработки предыдущей лабораторной работы, реализовать подключение к базе данных с помощью библиотек JDBC (для каждой базы данных своя JDBC-библиотека), дальнейшее взаимодействие с базой данных должно осуществляться с помощью Hibernate. (Определение используемой базы данных на выбор студента, но желательно PostgreSQL).

2.1. Необходимо создать отдельный класс, который будет отвечать за описание подключения к базе данных с помощью технологий Hibernate.

2.2. У данного класса должен быть описан метод, возвращающий экземпляр класса Session (библиотеки Hibernate), то есть уже подключенную сессию к базе данных.

3. Создание таблицы в базе данных:

3.1. Структура таблицы берется из предыдущей лабораторной работы.

3.2. Создать класс, соответствующий структуре таблицы, с элементами аннотаций для взаимодействия с базой данных через библиотеку Hibernate.

4. Взаимодействие с таблицей необходимо осуществить на вебстранице, которая была создана ранее и должна позволить выполнить следующие функции: удаления, обновления, добавления и отображения строк.

## 2.5. Лабораторная работа № 5

### 2.5.1. ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является ознакомление с технологией MVC, а также разработка веб-сайта с использованием «шаблонизатора» на основе предыдущей лабораторной работы.

### 2.5.2. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Про технологии MVC было подробно рассказано в теоретической части учебно-методического пособия. А сейчас мы подробно рассмотрим тему библиотеки FreeMarker.

FreeMaker — это написанная на Java библиотека, предназначенная для реализации механизма шаблонов. Шаблоны работают с текстом, подставляя в текст значения переменных. Цепочка следующая. Есть исходный текст, он представляет собой HTML с особыми тэгами (выражениями — не HTML, а именно шаблонизатора; язык FreeMaker называется FTL, FreeMaker Template Language). Далее этот текст проходит через некоторый процесс, называемый рендерингом. Выходом является текст. Но в нем воспринимаемые шаблонизатором

тэги уже заменены на конкретные значения. На этом этапе обычно остается «чистый» HTML — страница в том виде, в каком она будет передана клиенту.

Структура работы библиотеки FreeMarker хорошо видна на рис. 2.4.

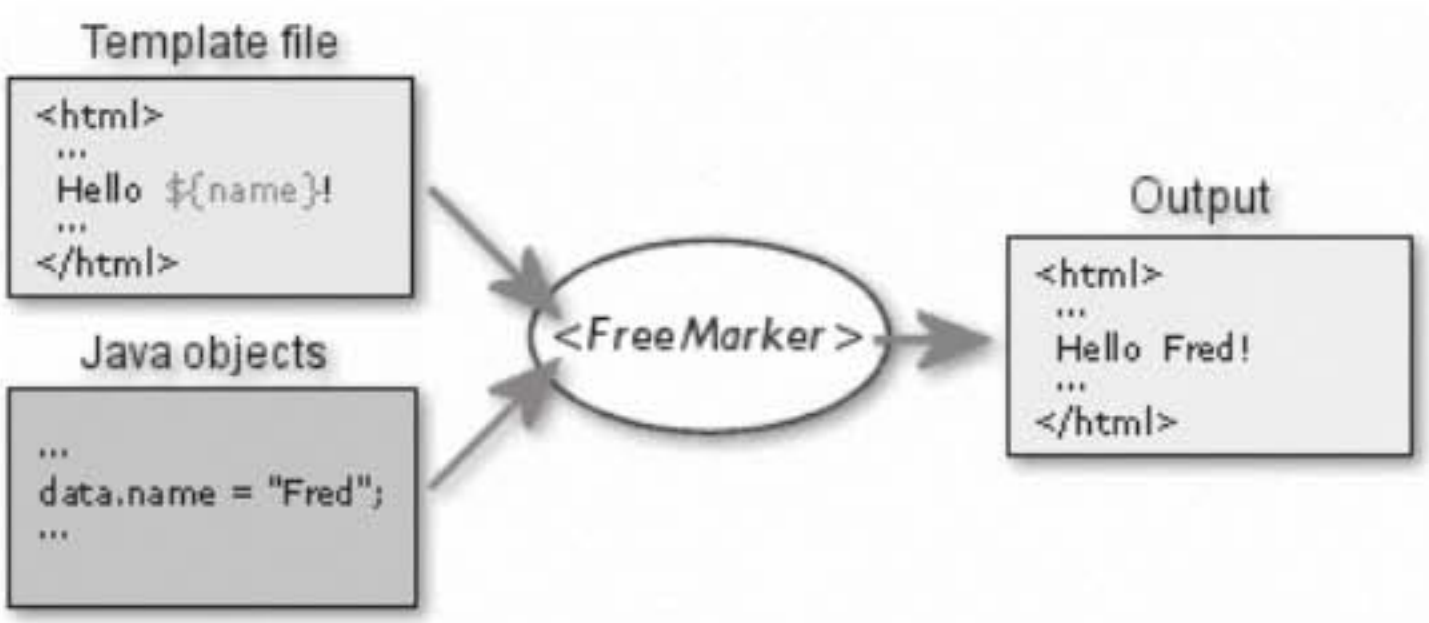


Рис. 2.4. Структура работы библиотеки FreeMarker

Наиболее часто шаблонизатор применяется следующим образом. Создается класс, в котором собирается вся необходимая для отображения информация (изъятая из model MVC). И этот класс используется в качестве *основы* (контекста) для работы шаблонизатора, для получения HTML из заготовки (шаблона).

Рассмотрим следующий пример. Есть шаблон, текстовый файл с именем ourTemplate.html, содержащий следующую строку.

<h1>Здравствуйте, ${userName}!</h1>

Данный пример позволяет нам не прописывать статично имя пользователя, а задать его через функцию шаблонизатора.

В Java-классе необходимо выполнить следующие действия.

1. HashMap renderContext = new HashMap ();
2. renderContext.put ("userName", getCurrentUser ().getName ());
3. Template template = fmConfig.getTemplate ("ourTemplate.html");
4. StringWriter writer = new StringWriter ();
5. template.process (renderContext, writer);
6. String html = writer.toString ();
7. sendToClient (html);

Создается словарь (HashMap). В элемент с именем userName помещается ФИО текущего пользователя, полученное вызовом функций getCurrentUser ().getName (). Дальше создается объект-шаблон (template) основанный на ранее сформированной конфигурации (fmConfig) из файла outTemplate.html, и к нему применяем операцию process.

На выходе получаем текстовую строку, которая будет содержать примерно следующее: <h1>Здравствуйте, Иванов Иван Иванович!</h1>

Мы *оживили* страницу, заменив шаблон на реальное значение переменной. Такую HTML-строку можно отправлять.

FreeMaker способен заменять шаблоны на реальное значение переменных — это значит сильно упрощать ситуацию. FreeMaker имеет достаточно развитый язык, позволяющий создавать списки, производить условные переходы и т. д. Кроме того, если чего-то не хватило в самом FTL, можно определить и свои собственные директивы (практически это вряд ли понадобится, но такая возможность есть).

Пример списка, сделанного при помощи механизма шаблонов. Пример шаблона:

```
1. <table>
2. <#list Users as u >
3. <tr>
4. <td>${u.getLastName ()}</td>
5. <td>${u.getFirstName ()}</td>
6. </tr>
7. </#list>
8. </table>
```

Строка в таблице фигурирует один раз. Класс контекста мог бы выглядеть следующим образом:

```
1. HashMap context = new HashMap ();
2. //Собираем всех пользователей, которых мы хотим получить
3. //в таблице на экране — в вектор.
4. Vector Users = new Vector ();
5. User u = new User ("Иванов", "Иван", "Иванович"); Users.add (u1);
6. User u2 = new User ("Петров", "Петр", "Петрович"); Users.add
   (u2);
7. User u3 = new User ("Николаев", "Николай", "Николаевич");
   Users.add (u3);
```

```
8. //Теперь в контекст в качестве именованной переменной по-
   мещаем сам вектор.
9. context.put ("Users", Users);
```

Полученный после рендеринга HTML будет выглядеть примерно следующим образом:

```
1. <table>
2. <tr>
3. <td>Иванов</td>
4. <td>Иван</td>
5. </tr>
6. <tr>
7. <td>Петров</td>
8. <td>Петр</td>
9. </tr>
10. <tr>
11. <td>Николаев</td>
12. <td>Николай</td>
13. </tr>
14. </table>
```

Отметим две вещи:

1) строк в таблице стало столько, сколько элементов в коллекции Users в контексте. Если их будет 100, значит, директива FreeMaker <#list> выполнится 100 раз, и получаем HTML с таблицей, состоящей из 100 строк;

2) в качестве *переменной*, которая заменится шаблонизатором на реальное значение, не обязательно должно выступать именно *значение*. Это может быть и функция, которая вызовется автоматически, и в шаблон будет подставлен результат работы этой функции.

FreeMaker работает с текстом. У него текст на входе и текст на выходе. HTML — это частный случай применения механизма шаблонов. Это может быть и JavaScript, и CSS, и вообще все что угодно. В случае с рендерингом через шаблонизатор JavaScript появляется удобный механизм начального заполнения переменных на клиенте значениями из серверной части.

### 2.5.3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Постановка задачи:

1. Разобраться с технологией MVC и библиотекой FreeMarker.

Используя библиотеку FreeMarker, изменить структуру вашего сайта, результата предыдущей лабораторной работы.

1.1. Отделить одинаковую (статичную) часть от всех страниц (одна из которых обязательно является меню) в отдельный HTML-файл.

1.2. Все созданные ранее страницы изменить в соответствии с требованиями работы с библиотекой FreeeeMarker.

2. При организации ответа нужно объединить страницы меню и запрашиваемую страницу с помощью FreeMarker.

## 2.6. Защита лабораторных работ

Результаты каждой выполненной лабораторной работы:

1) сформированный проект, выполненный по заданию лабораторной работы;
2) оформленный отчет, содержащий все этапы выполненной работы.

Защита лабораторной работы:

1) продемонстрировать разработанное приложение, выполненное в соответствии с заданием;
2) предоставить отчет по лабораторной работе в печатном виде;
3) давать четкие ответы по выполненной работе (студент должен владеть теоретическими знаниями, свободно комментировать все строчки кода программы и уметь формулировать выводы о проделанной работе).

### 2.6.1. ПРАВИЛА ОФОРМЛЕНИЯ ОТЧЕТА

Отчет по лабораторным работам должен содержать следующие структурные части:

— титульный лист (1 стр.);
— содержание (1 стр.);
— цели и задачи лабораторной работы (1 стр.);