Занятие 3. Протоколы, использующие HTTP в качестве транспорта: XML-API, JSON-API. Виды и схемы аутентификации протокола HTTP

Постоянный рост использования WEB, увеличение объемов и потоков информации, передаваемой по сети, и скорости ее обновления, увеличение количества услуг, «переезжающих» из офлайн-режима в онлайн и разнообразия источников как информации, так и услуг привело к необходимости не только предоставить пользователям возможность получить быстрый доступ ним, но и реализовать своевременное обновление информации в своих базах, получаемой от партнеров, а так же в случае сервисов-агрегаторов возможность выбора услуг от различных поставщиков в режиме реального времени. Это привело к появлению множества WEB-сервисов, которыми могли пользоваться как клиенты, так и другие сервисы. Сначала протоколы взаимодействия между такими сервисами определялись разработчиками, они могли работать как поверх сетевого или транспортного уровня, так и надстраивались над протоколами прикладного уровня. Чаще всего в качестве транспорта прикладного уровня используется протокол HTTP.

Со временем начали разрабатываться общие подходы, реализации взаимодействия WEB-сервисов, форматы запросов и ответов, другими словами API WEB-сервисов. Чтобы создаваемые протоколы были более универсальными и расширяемыми для содержимого запросов и ответов использовался расширяемый язык разметки XML (eXtensible Markup Language), позже с развитием фронтэнда (Front-End, клиентская сторона интерфейса) его место занял менее избыточный формат JSON (JavaScript Object Notation).

Сейчас наиболее распространенными протоколами доступа к API WEB-сервисов являются:

- XML-RPC
- REST (хотя это не протокол, а подход)
- SOAP

XML-RPC практически является прародителем протоколов доступа к API WEB-сервисов. Он разрабатывался по заказу компании Microsoft и по сути стал реализацией технологии RPC (Remote Procedure Call) поверх протокола HTTP с использованием форматирования сообщений в XML. Его преимуществом была простота и легкость реализации, что сослужило протоколу плохую службу - стандартом он так и не стал, зато стал использоваться многими разработчиками для быстрого запуска работающей системы.

Пример тела XML-RPC-запроса:

```
<?xml version="1.0"?>
 <methodCall>
   <methodName>examples.getAirportCode</methodName>
   <params>
     <param>
         <value><i4>567</i4></value>
     </param>
   </params>
 </methodCall>
```

Пример тела XML-RPC-ответа:

```
<?xml version="1.0"?>
 <methodResponse>
   <params>
     <param>
         <value><string>SVO</string></value>
     </param>
   </params>
 </methodResponse>
```

Из примеров хорошо видно, что они легко парсятся (англ. parse - разбирать) даже взглядом, и уж тем более сервисом. XML-RPC использует HTTP только в качестве транспорта, передавая всю информацию только в телах запросов и ответов, как об успешных, так и неудачных вызовах.

Сейчас так же широко используется фреймворк gRPC, изначально разработанный компанией Google. Он обеспечивает прозрачное для разработчиков клиент-серверное или межсервисное взаимодействие в том числе по протоколу HTTP, поддеживает возможность генерации кода для различных языков программирования, позволяя разрабатывать клиентскую и серверную часть на разных языках.

REST (Representational State Transfer) - это подход, архитектура работы и взаимодействия WEB-сервисов, созданных одним из разработчиков протокола HTTP и подразумевающий использование всех его возможностей: использование методов работы с объектами/обращения к ресурсам, возможности аутентификации, служебные заголовки, заголовки согласования контента, передача бинарных данных, коды ошибок. То есть разработчикам как серверной, так и клиентской части не нужно реализовывать в своих протоколах передачу информации о действиях, которые нужно совершить с объектом, а так же коды и описания ошибок, для этого используются уже существующие и описанные в стандартах методы и коды статуса ответа протокола HTTP.

Пример REST-запросов и ответов:

```
GET /airports/i4/567 HTTP/1.1

HTTP/1.1 200 OK
{ 'type':'airport', 'name':'SVO', 'i4':567 }


DELETE /airports/i4/123 HTTP/1.1

HTTP/1.1 403 Forbidden
```

Поскольку REST - это не протокол, а подход, то он дает разработчикам абсолютную гибкость реализации, а близость к протоколу HTTP упрощает и ускоряет его обработку WEB-сервисами.

SOAP (Simple Object Access Protocol) - это достаточно популярный у разработчиков тяжеловесный протокол, так же работающий поверх http. SOAP передает по http запросы и ответы в телах, полностью возможности протокола не использует, за исключением может быть аутентификации или передачи токенов в заголовках.

Пример SOAP-запроса:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
schemas.xmlsoap.org/soap/envelope">
  <soapenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

Пример SOAP-ответа:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
schemas.xmlsoap.org/soap/envelope">
  <soapenv:Header>
    <wsa:ReplyTo>
      <wsa:Address>schemas.xmlsoap.org/ws/2004/08/addressing/role/
anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:From>
      <wsa:Address>localhost:8080/axis2/services/MyService</wsa:Address>
    </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
  </soapenv:Header>
  <soapenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

Действительно, запросы и ответы протокола SOAP выглядят очень избыточными и слишком объемными, чтобы разработчикам было удобно с ними работать. Однако SOAP стал популярен вовсе не из-за структуры запросов или ответов, универсальности или расширяемости, а благодаря технологии WSDL (язык описания вэб-сервисов, web-service description language). Компании при разработке вэб-сервера, предоставляющего возможность взаимодействия по протоколу SOAP, создают WSDL. Он передается разработчиком, который хочет сделать стыковку в вэб-сервисом, в библиотеку, обеспечивающую работу с SOAP, и всю работу по формированию запросов и парсингу ответов библиотека делает сама, опираясь на описание из WSDL. Для языка JAVA так же существует возможность генерации серверного кода/библиотеки на основе WSDL, которая позволяет работать с вэб-сервисом используя внутренние вызовы сгенерированной локальной библиотеки. Конечно такое удобство работы с SOAP-сервисами не отменяет накладных расходов на передачу и обработку запросов/ответов.

Кроме полноценных web-сервисов с развитием фронтэнда появилось множество библиотек, написанных на языке JavaScript, позволяющих удобно и быстро разрабатывать пользовательские интерфейсы, а так же реализующих подходы асинхронного обращения к вэб-сервисам для получения и обновления данных, чтобы интерфейсы могли быть интерактивными. Одним из первых подобных широко используемых подходов был AJAX (Asynchronous Javascript and XML).

AJAX - это не столько самостоятельная технология, а скорее концепция использования нескольких технологий: асинхронного обращения к серверу для получения или обновления данных с использованием XMLHttpRequest (это API доступно в скриптовых языках браузеров) и динамического изменения содержимого страницы с использованием операций с элементами DOM (Document Object Model). Несмотря на наличие XML в аббревиатуре, технологии не накладывают ограничения на формат данных, передаваемая информация может быть в различных текстовых форматах: XML, HTML, JSON или же простым текстом.

AJAX по своему взаимодействию клиента с сервисом немного похож на SOAP. Однако подход REST набирал популярность не только во взаимодействии сервисов, но и во фронтэнде. Одним из «последователей» подхода REST можно назвать GraphQL (хоть он может использоваться и на клиенте и на сервере).

GraphQL - это синтаксис, описывающий как запрашивать данные, в основном используется во фронтенде для запросов клиентами данных с сервера. Его реализация находится где-то между клиентом и источниками данных (одним или несколькими), он принимает запросы от клиентов и возвращает необходимые данные, собранные из нескольких источников, в необходимом клиенту формате. GraphQL позволяет клиенту точно указать, какие данные ему нужны, облегчает агрегацию данных из нескольких источников, использует систему типов для описания данных, обладает механизмами ограничений/ разрешений на доступ к определенным типам данным. Обычно в качестве формата передаваемых запросов и ответов используется JSON.

АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ

Для защиты конфиденциальных данных от несанкционированного доступа в любых системах используются различные функции, реализующие и использующие различные политики безопасности. Одними из основных и взаимодополняющих функций являются аутентификация и авторизация.

Аутентификация - это функция **подтверждения личности** пользователя идентификатора (логина) пользователя и пароля.

Поскольку обычная проверка логина и пароля (учетных данных) не может обеспечить абсолютную (или иногда достаточную) безопасность идентификации пользователя, поэтому для более надежной защиты были разработаны категории учетных данных (факторов):

- *Однофакторная аутентификация* (SFA, Single Factor Authentication) - базовый метод, предусматривающий проверку только одной категории учетных данных, обычно логина и пароля.
- *Двухфакторная аутентификация* (2FA, 2 Factor Authentication) - двухступенчатый процесс проверки двух не связанных друг с другом категорий данных, например, дополнительно к проверке логина и пароля может быть запрошен специальный одноразовый код, отправляемый сервисом в SMS или письме электронной почты.
- *Многофакторная аутентификация* (MFA, Multi Factor Authentication) - современный метод проверки подлинности, включающий в себя проверку более чем двух категорий независимых учетных данных, например, в дополнении в логину и паролю может быть запрошено подтверждение через брелок-токен, передача дополнительных заголовков или параметров запроса, подтверждение не только личности пользователя, но и его рабочего места (компьютера). Обычно используется в финансовых организациях, государственных органах и т.п.

Авторизация - это функция подтверждения наличия у аутентифицированного пользователя **прав на доступ** к запрашиваемому ресурсу или к выполнению запрашиваемых действий. Реализация проверки авторизации, степени и уровни доступа, различие применения проверок для различных ресурсов, политик доступа всегда остается за разработчиками сервисов. В то время как аутентификация может быть реализована в протоколе прикладного уровня.

ВИДЫ И СХЕМЫ АУТЕНТИФИКАЦИИ В ПРОТОКОЛЕ HTTP

В протоколе HTTP реализованы два вида аутентификации пользователей:

- Аутентификация на заголовках WWW-Authenticate, Authorization
- Аутентификация на формах или куках (заголовки Cookie, Set-Cookie)

АУТЕНТИФИКАЦИЯ НА ЗАГОЛОВКАХ WWW-AUTHENTICATE, AUTHORIZATION

В описаниях протокола HTTP такой вид аутентификации называется авторизацией, так как с точки зрения протокола результат проведения аутентификации пользователя/ клиента позволяет определить наличие у него прав на доступ к указанному в запросе ресурсу.

При попытке неавторизованного клиента сделать запрос на закрытый HTTP-авторизацией ресурс сервер отправляет клиенту ответ с кодом 401 Unauthorized, содержащий заголовок WWW-Authenticate, сообщающий клиенту о необходимости передавать заголовок Authorization с учетными данными в определенном виде (вид определяется схемой аутентификации). После первой успешной попытки передать серверу учетные данные в заголовке Authorization клиент (браузер) должен будет так же передавать его во всех последующих запросах к этому же серверу. А это означает, что перехват любого запроса от клиента может дать возможность злоумышленнику получить доступ к серверу и скомпрометировать клиента.

Схемы аутентификации определяют тип и вид передаваемых в заголовке данных, их защищенность, а так же сложность реализации на клиенте и сервере:

Basic Authentication - самая простая и самая небезопасная реализация аутентификации клиента. В ответе от сервера на запрос неавторизованного клиента возвращается заголовок *WWW-Authenticate: Basic realm='Произвольное название'*, в браузере при получении такого ответа появится всплывающее окно для ввода логина и пароля, в заголовке которого будет указана строка из параметра realm. После ввода логина и пароля пользователем они упаковываются через двоеточие в строку (username:password), закодированную Base64 (метод кодирования, позволяет легко осуществлять как операции кодирования, так и декодирования), и отправляются на сервер в заголовке *Authorization: Basic 'dXNlcm5hbWU6cGFzc3dvcmQK'* . В случае успешной аутентификации клиент во всех запросах к серверу передает заголовок *Authorization: Basic 'dXNlcm5hbWU6cGFzc3dvcmQK',* который может быть перехвачен злоумышленником.

Digest Authentication - по сравнению с Basic более защищенная схема аутентификации, так как для передачи учетных данных пользователя и его идентификации используется передача хешированных данных, содержащих случайные уникальные строки. Заголовок WWW-Authenticate в ответ на запрос неавторизованного клиента содержит описание схемы, алгоритма хеширования, параметров аутентификации, часть параметров передается клиентом в последующем запросе без изменений, часть используется как части хешей учетных данных. Логин клиента передается в заголовке Authorization в открытом виде, пароль содержится в части хеша, содержащего в себе в том или ином виде все части заголовков WWW-Authenticate и Authorization, включая URI запроса. При перехвате такого запроса злоумышленником не происходит раскрытия учетных данных пользователя и использование перехваченного заголовка не позволит использовать его для отправки злоумышленником других запросов под учетными данными пользователя, поскольку при изменении URI все хеши придется пересчитывать заново. Реализация такого вида аутентификации сложнее и затратнее как для клиента, так и для сервера. Однако она имеет несколько очевидных проблем, в том числе относящихся к безопасности. Поскольку алгоритмы хеширования однонаправленные, то есть можно получить из строки ее хеш, но проделать обратную информацию невозможно, то для проверки передаваемых клиентом хешей учетных данных на сервере необходимо собирать аналогичные хеши, а значит пароли

пользователей на сервере должны храниться в открытом виде. В некоторых реализациях сервер вынужден хранить у себя уникальную строку, идентифицирующую сессию с клиентом, которая генерируется в момент отдачи первого 401 кода и участвует в запросах и ответах, а это приводит к большим накладным расходам при достаточно большом количестве клиентов. Так же по умолчанию используется алгоритм хеширования MD5, слабостью которого является большое число коллизий (когда две и более различных строк имеют одинаковое значение хеша), а это значит, что подобрав нужную коллизию злоумышленник может скомпрометировать пользователя (хоть это и очень трудоемкий и долгий процесс).

Bearer Authentication - это аутентификация по некоторому токену (Token, некоторая уникальная строка, идентифицирующая клиента), выданному клиенту сервером после прохождения аутентификации, например, по схеме Basic. Притом сервер, выдающий токен после прохождения аутентификации (его еще называют Identity Provider), и сервер, к которому обращается клиент с этим токеном (Service Provider), в большинстве случаев разные. Токен и необходимая информация для аутентификации клиента сервером так же передаются клиентом в заголовке Authorization. Содержимое и формат токена и заголовка согласовываются между клиентом и сервером для того, чтобы обеспечить необходимый уровень защищенности клиента от компрометации при перехвате токена из запроса. Такой тип аутентификации обычно используется для межсервисного взаимодействия.

АУТЕНТИФИКАЦИЯ НА ФОРМАХ ИЛИ КУКАХ (ЗАГОЛОВКИ COOKIE, SET-COOKIE)

Если процессы и параметры аутентификации на заголовках WWW-Authenticate, Authorization описываются стандартами протокола HTTP, то реализация аутентификации пользователя на формах (использующая заголовки Cookie, Set-Cookie) зависит полностью от разработчиков сервиса, включая коды статусов ответа, названия и количество полей учетных данных, количество факторов аутентификации.

Особенность пары заголовков Cookie, Set-Cookie заключается в том, что все, что было передано сервером в HTTP-ответе в заголовке Set-Cookie, клиент будет передавать в последующих заголовках к этому же серверу в заголовке Cookie. Получается, что для такого вида аутентификации клиент только при первом запросе передает серверу свои учетные данные для прохождения аутентификации, все последующие запросы будут авторизованы при помощи содержимого заголовка Cookie. Такой вид аутентификации называется аутентификацией на формах, так как для ввода учетных данных и прохождения аутентификации на сервере клиент заполняет HTML-форму, которую отдает сервер.

Cookie представляет собой заголовок, содержащий название куки (Cookie) и текстовую строку произвольной длины и содержания (определяется сервером), например: *«Cookie: yandexuid=1222674211612747462»*. В запросе и ответе может быть несколько разных заголовков Cookie и Set-Cookie.

Заголовок Cookie не только будет передаваться клиентом серверу в том же виде, в котором пришел в ответе, но и в том, что при передаче сервером клиенту заголовка Cookie (так же называется «установка Cookie») сервер может определять при каких условиях, как долго и в каких запросах клиент будет использовать в заголовках ту или иную строку Cookie. Это определяется в параметрах заголовка Set-Cookie в ответе сервера, например: *«Set-Cookie: yandexuid=1222674211612747462; Expires=Thu, 06-Feb-2031 01:24:22 GMT; Domain=.ya.ru; Path=/»*

Параметры заголовка Set-Cookie записываются в самом заголовке:

- Самым первым параметром заголовка Set-Cookie является название куки и ее значение, например, yandexuid=1222674211612747462
- **Expires**=<дата> - дата, после которой кука устаревает и становится недействительной, например, Expires=Thu, 06-Feb-2031 01:24:22 GMT
- **Max-Age**=<количество секунд> - количество секунд, по истечение которых кука устаревает и становится недействительной, например, Max-Age=86400
- **Domain**=<имя хоста> - имя домена/хоста, в запросах на который клиентом будет передаваться полученная кука, указание нескольких имен невозможно, например, Domain=www.ya.ru - кука будет передаваться только в запросах с заголовком Host: www.ya.ru, Domain=.ya.ru - кука будет передаваться в запросах с заголовком Host: ya.ru, Host: www.ya.ru, Host: 123.ya.ru и т.п., то есть будет работать для всех поддонов ya.ru
- **Path**=<путь от корня сайта> - определяет часть URL от корня сайта, при запросах на который будет передаваться полученная кука, например, Path=/ - кука будет передаваться в запросах на все страницы сайта, Path=/docs/ - кука будет передаваться в запросах на все страницы сайта, начинающиеся с /docs/
- **Secure** - наличие этого параметра говорит о том, что кука будет передаваться только в запросах по протоколу HTTPS

- **HttpOnly** - наличие этого параметра запрещает javascript-у на запрашиваемой клиентом странице получать доступ к куке
- **SameSite**=<параметр> - параметр, определяющий, какие дополнительные ограничения на запрос накладываются на отправку куки в запросе, например, SameSite=Strict - кука будет отправляться только в запросах, пришедших с того же самого хоста (домена), SameSite=Lax - кука не будет передаваться в кросс-сайтовых (например, загрузка изображений на странице) запросах или в запросах из фреймов, но будет передаваться при переходе клиентом по ссылке, SameSite=None - клиент будет отправлять куку во всех запросах (включая кросс-сайтовые и запросы из фреймов)

Процесс аутентификации на куках состоит из нескольких этапов:

- Неаутентифицированный клиент пытается запросить ресурс, требующий аутентификации клиента, сервер в ответ возвращает клиенту информацию о необходимости пройти аутентификацию. Это может быть реализовано как с отправкой 401 кода статуса, так и с отправкой 302 или даже 200 кода.
- Клиент переправляется на страницу ввода учетных данных (Identity Provider, как и в случае с Bearer может быть отличным от сервера, требующего аутентификацию клиента при попытке обращения к ресурсу), в зависимости от реализации сервиса клиенту может быть предложено ввести только логин и пароль, или же предоставить другое количество и вид учетных данных, пройти двухфакторную аутентификацию и т.п.
- После ввода учетных данных и проверки их сервером в случае успешной аутентификации клиенту в ответе возвращаются заголовки Set-Cookie с необходимыми куками для прохождения аутентификации на сервисе (Service Provider) и возможно дальнейшие перенаправления клиента обратно на сервис
- Клиент уже с установленными заголовками Cookie запрашивает ресурс
- В случае неудачной аутентификации Identity Provider может либо предложить клиенту другие попытки ввода учетных данных, либо отказать в доступе к ресурсу

Поскольку Cookie - это заголовки HTTP-запроса, которые могут быть перехвачены злоумышленником, то в целях повышения безопасности содержимое одного заголовка Cookie не должно без дополнительных условий аутентифицировать клиента. Поэтому очень часто при успешной аутентификации клиента ему возвращается не один заголовок Set-Cookie, а несколько взаимозависимых, описывающих сессию клиента и набор его параметров или параметров конкретного подключения. Набор этой информации известен только разработчику сервисов аутентификации и авторизации клиента, а сама информация доступна только в текущем соединении клиента и сервера.

Реализация аутентификации клиента на формах для разработчиков сложнее (как серверных, так и клиентских), но позволяет иметь больше возможностей и гибкости в части передаваемых в куках данных, как с точки зрения безопасности, так и с точки зрения функционала сервиса (например, реализация двухфакторной или многофакторной аутентификации, встраивание в куки дополнительной информации и для авторизации клиента внутри сервисов). Плюс такая реализация позволяет гибко настраивать внешнее отображение страницы аутентификации для клиента.