

Системный дизайн

архитектура и сервисы

Как работает API на примере ресторана

Вы являетесь потребителем API.

Отправляем запрос на пункт меню команде кухни.

Официант - это API.

Он передает ваш запрос шеф-повару на кухне, поскольку вы не можете напрямую пойти на кухню и сделать заказ. Затем официант доставляет готовую еду, которая является ответом, из кухни к вашему столу. Официант выступает в роли передатчика данных и механизм доставки для взаимодействия с поварами на кухне.

Меню - это документация API.

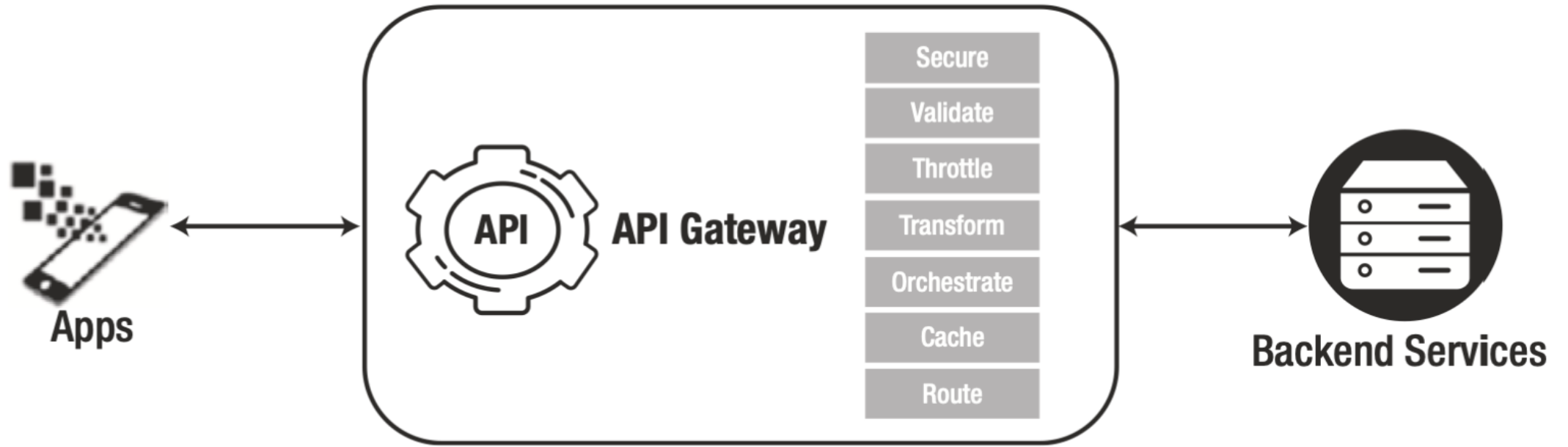
В ней говорится о том, что вы можете попросить у API, официанта. В нем есть информация о блюдах, которые могут быть приготовлены поварами на кухне, и их стоимости. Официант понимает только ту информацию, которая содержится в меню.

Кухня - это сервер API.

Она обслуживает запрос, готовя еду, запрошенную в меню. Шеф-повар на кухне знает, как приготовить еду, что является бизнес-логикой, которая обрабатывается сервером. Но вы не знаете ни имени, ни адреса шеф-повара.

Что API включает

- **Действия**, которые API может выполнять для обеспечения бизнес-функциональности
- **Место**, где можно получить доступ к API (для указания места обычно используется URI).
- **Входные и выходные параметры** API, включая имена параметров, формат сообщения и типы данных (формат сообщения может быть JSON или XML).
- **Сетевой протокол**, который клиент должен использовать для вызова API (распространенные сетевые протоколы включают HTTP/HTTPS, FTP/SFTP и JMS).
- **Информация о безопасности API**, которую должен соблюдать и предоставлять клиент для доступа к API (API может быть защищен с помощью базовой и многофакторной аутентификации, mTLS, проверки ввода, фильтрации IP-адресов и т. д.)
- Соглашение об уровне обслуживания (SLA), которого придерживается поставщик API, например, время отклика, пропускная способность и доступность.
- Технические **требования по ограничению скорости**, контролирующие количество запросов, которые приложение или пользователь может сделать за определенный период.
- Любые **юридические или деловые ограничения** на использование API (сюда могут входить условия коммерческого лицензирования, требования к брендингу, сборы и платежи за использование и т. д.).
- **Документация**, помогающая понять API



API Gateway

API Security

- **Аутентификация** - это процесс однозначного определения и подтверждения личности клиента.
- **Авторизация** контролирует уровень доступа, предоставляемый приложению, выполняющему вызов API. Она определяет, какие ресурсы и методы API может вызывать приложение.
- **Посредничество при идентификации**: API обычно используют протоколы OAuth для обеспечения безопасности. Однако внутренние сервисы могут быть защищены с помощью SAML или WS-Security. Следовательно, платформа управления API должна быть способна интегрироваться с внутренними IDM-платформами и выполнять посредничество при идентификации.
- **Конфиденциальность данных**: API-интерфейсы предоставляют данные, которые могут быть конфиденциальными; такие данные должны быть видны только тому, кому они предназначены. Любые конфиденциальные данные при передаче должны быть зашифрованы.
- **Защита от атак типа «отказ в обслуживании» (DoS)**: API открывают ценные данные и активы за пределами брандмауэров предприятия. Это увеличивает поверхность атаки и делает их более подверженными атакам. Хакеры могут попытаться вывести из строя внутренние системы, перекачивая через API неожиданно высокий трафик.
- **Обнаружение угроз**: Вероятность того, что злоумышленники совершают атаки с использованием вредоносного контента, высока для публичных API.

API Traffic Management

- **Квота потребления:** Определяет количество вызовов API вызовов, которые приложение может совершить к бэкграунду за определенный промежуток времени. Вызовы, превышающие лимит квоты, могут быть дросселированы или остановлены. Квота, разрешенная для приложения, зависит от бизнес-политики API и модели монетизации.
- **Spike arrest:** Определяет неожиданный рост трафика API трафика. Защищает внутренние системы, не рассчитанные на высокую нагрузку.
- **Отсечка использования:** Предоставляет механизм для замедления последующих вызовов API. Это позволяет повысить общую производительность и уменьшить воздействие в часы пик.
- **Приоритезация трафика:** Помогает платформе управления API определить, какой класс клиентов должен иметь более высокий приоритет. Вызовы API от клиентов с высоким приоритетом должны обрабатываться в первую очередь.

Interface Translation

- **Перевод форматов:** Внутренняя система может ожидать данные в формате SOAP, XML, CSV или любом другом собственном формате.
- **Перевод протоколов:** Большинство внутренних систем, в которых предоставляют потребителям интерфейс SOAP. Однако SOAP не является протоколом, подходящим для API для создания приложений для цифровых устройств, для которых лучше подходит WebSocker или REST.

Caching

это механизм оптимизации производительности, позволяющий отвечать на запросы статическими ответами, хранящимися в памяти.

API-прокси может хранить в памяти ответы внутреннего сервера, которые не часто меняются.

Когда приложения делают запросы на один и тот же URI, кэшированный ответ может быть использован для ответа вместо того, чтобы перенаправлять эти запросы на внутренний сервер. Таким образом, кэширование может повысить производительность API за счет снижения задержек и сетевого трафика.

Service Routing

- **Сопоставление URL-адресов:** Путь входящего URL может отличаться от пути внутреннего сервиса. Возможность сопоставления URL позволяет платформе изменить путь во входящем URL на путь внутреннего сервиса.
- **Диспетчеризация сервисов:** позволяет платформе управления API выбирать и вызывать нужный внутренний сервис. В некоторых случаях может потребоваться вызов нескольких сервисов для организации и возврата агрегированного ответа потребителю.
- **Пул соединений:** Платформа управления API должна быть способна поддерживать пул соединений с внутренним сервисом. Пул соединений повышает общую производительность.
- **Балансировка нагрузки:** Она распределяет трафик API между внутренними сервисами.

Сервис-ориентированная архитектура (SOA)

Это **стиль проектирования** программного обеспечения, при котором сервисы предоставляются посредством **протокола связи по сети**.

В сервис-ориентированной архитектуре несколько сервисов взаимодействуют друг с другом одним из двух способов: посредством непосредственной передачи данных или посредством двух или более сервисов, координирующих какую-либо деятельность.

Основные принципы SOA

- Основные строительные блоки SOA — это сервисы. **Сервис** — это автономный компонент, который выполняет определенную **бизнес-функцию**.
- Сервисы предоставляют четко определенные **интерфейсы**, через которые другие сервисы или клиенты могут взаимодействовать с ними.
- Сервисы должны быть **автономными и независимыми** от других сервисов. Это позволяет легко изменять, обновлять или заменять сервисы без влияния на другие части системы.
- Сервисы взаимодействуют друг с другом через сетевые протоколы, такие как HTTP, SOAP, REST, и т.д.
- Для облегчения поиска и использования сервисов, SOA часто включает в себя сервис-реестр, где регистрируются все доступные сервисы.

REST

- Является **архитектурным стилем**
- REST работает с **ресурсами по протоколу HTTP**
- Ресурсом может быть любая целостная и осмысленная структура, к которой можно обратиться
- В основе REST лежит **идентификация каждого ресурса с помощью URL**
- **REST** позволяет создать ресурс, удалить ресурс, получить или изменить ресурс

URL

Единый локатор ресурсов (URL) - это адрес ресурса (файла, записи JSON, изображения и т. д.) в Интернете. URL уникальны (могут ссылаться только на один ресурс) и полностью определены (это означает, что они могут быть разрешены к местоположению ресурса без двусмысленности).

Общий вид URL показан здесь:

`scheme://host[:port]/path[?query-string][#fragment-id]`

URL

- - **Схема**: Схема определяет протокол, который будет использоваться для доступа к ресурсу. Схемы состоят из ключевого слова и разделителя `://`. К распространенным протоколам относятся `http`, `https`, `ftp` и `file`.
- - **Хост**: Хост указывает местоположение узла, содержащего ресурс, обычно это сервер. Хост должен быть разрешаемым к сетевому местоположению через клиента, обычно с помощью DNS. Хосты могут быть локальными или полностью определенными доменными именами (FQDN) в публичном Интернете, а также могут использовать комбинацию локальных имен плюс FQDN, например, `server1.mydomain.com`.
- - **Порт**: необязательный параметр порт указывает порт протокола Интернета (IP), который будет использоваться на хосте. Если он не указан, будут использоваться порты по умолчанию - порт 80 для `http` и порт 443 для `https`. Использование портов позволяет размещать несколько серверов на одном физическом сетевом устройстве.
- - **Путь**: Путь указывает местоположение ресурса в системе хранения хоста. Это прямой аналог пути к файлу на локальном диске: пути иерархичны и вложены друг в друга.
- - **Строка запроса**: Необязательная строка запроса позволяет настроить запрос, чтобы предоставить серверу параметры для уточнения возвращаемого ресурса(ов). Запросы предваряются оператором `?` и разделяются оператором `&`; пары ключ-значение разделяются оператором `=`. Ниже приведен пример строки запроса:
`?first_name=Homer&last_name=Simpson`
- - **Идентификатор фрагмента**: Наконец, необязательный идентификатор фрагмента определяет местоположение в ресурсе; обычно он используется для поиска заголовков разделов в HTML-документе и редко применяется для API.

Протокол HTTP

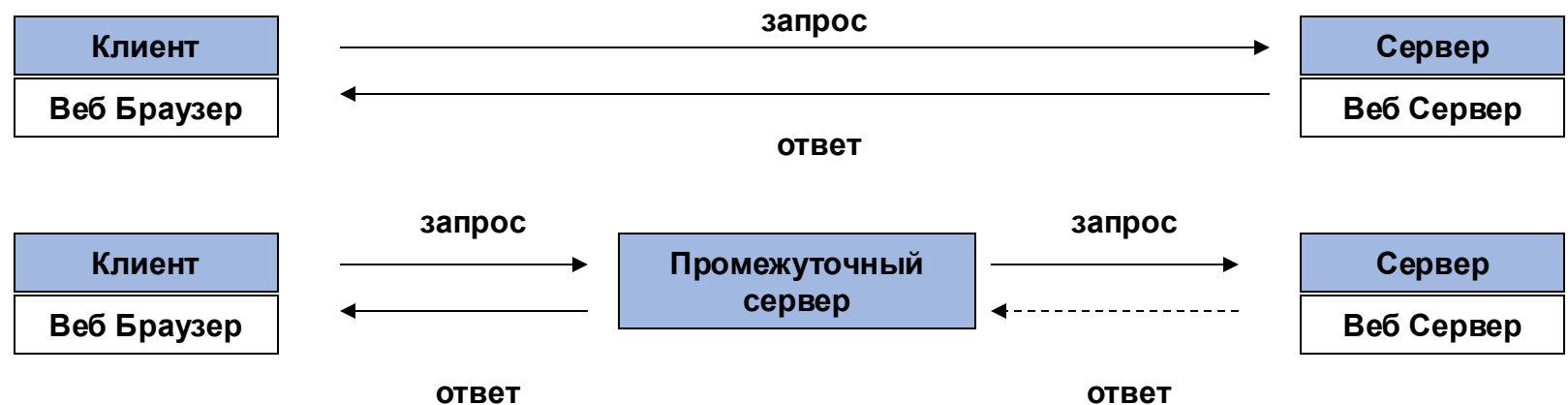
- Протокол передачи гипертекста HTTP является протоколом прикладного уровня для распределенных мультимедийных информационных систем. Это объектно-ориентированный протокол, пригодный для решения многих задач, таких, как создание серверов имен, распределенных объектно-ориентированных управляющих систем и др..
- Клиент обычно – конечный пользователь, в качестве сервера выступает Web- сервер.
- Протокол реализует сервис без соединения и состояния.
- Порт по умолчанию 80
- Адресация серверов в HTTP выглядит следующим образом:

<https://developer.mozilla.org/ru/docs/Web/HTTP/Overview>

Протокол HTTP

Стандартный сеанс работы выглядит следующим образом:

1. HTTP клиент инициирует соединение с сервером по протоколу TCP.
2. HTTP клиент посылает запрос серверу, указывая URL и параметры запроса.
3. Сервер получает сообщение от клиента.
4. Сервер отправляет клиенту результат запроса.
5. Сервер закрывает соединение.



HTTP Запрос

- **Состоит из**
 - Строка запроса «**Method SP Request-URI SP HTTP-Version CRLF**»
 - Заголовок запроса
 - Пустая строка
 - Тело сообщения-запроса (может отсутствовать)
- Возможные значения **Method**
 - **HEAD** – получить в ответ только заголовки, без данных
 - **GET** – запрос данных
 - **POST** – отправка данных
 - **PUT** – загрузка ресурса на сервер
 - **DELETE** – удаление ресурса с сервера
 - **TRACE** – запрос получение текста запрос обратно (для просмотра изменений, внесенных промежуточными серверами)
 - **OPTIONS** – получение методов, поддерживаемых сервером для данного URL
 - **CONNECT** – создание соединения для последующих взаимодействий (обычно для SSL).
- **Версия**
 - HTTP/1.0
 - HTTP/1.1
 - HTTP/2.0

HTTP Ответ

- **Состоит из**
 - Строка ответа «HTTP-Version SP Status-Code SP Reason-Phrase CRLF»
 - Заголовок ответа
 - Пустая строка
 - Тело сообщения-ответа (может отсутствовать)
- Возможные значения **Status-Code**
 - **1xx** - Informational
 - **2xx** - Success
 - **3xx** - Redirection
 - **4xx** - Client Error
 - **5xx** - Server Error
- **Версия**
 - HTTP/1.0
 - HTTP/1.1
 - HTTP/2.0

КОДЫ ОТВЕТОВ

Code ranges	Type of response	Details
100–199	Informational responses	Эти коды используются нечасто и указывают клиенту на промежуточный результат.
200–299	Successful responses	Указывает на успешный запрос. 200 - это OK и используется всеми методами, а 201 - это Created и используется методом POST.
300–399	Redirection messages	Указывает клиенту на то, что запрашиваемый ресурс переместился в другое место и что необходимо выполнить следующую операцию. 301 перемещается постоянно и является наиболее часто встречающимся.
400–499	Client error responses	Указывает, что клиент допустил ошибку в запросе. 400 - Bad request, 401 - Unauthorized, 403 - Forbidden и 404 - Not Found. Клиенты должны рассматривать отдельные случаи отдельно.
500–599	Server error responses	Указывает на то, что сервер столкнулся с ошибкой при обработке запроса. 500 - это внутренняя ошибка сервера, 501 - не реализовано, а 502 – bad gateway. Ошибки сервера могут быть временными и при повторной попытке могут быть успешными.

Сессии

Каждый HTTP-запрос является самодостаточным и не требует знания предыдущих или последующих запросов - для данного запроса сервер не знает, делал ли клиент предыдущие запросы. Это транзакция без статических данных.

Для полноценной работы клиента **важно, чтобы состояние сохранялось между последовательными операциями**, чтобы клиенты могли реализовать управление состоянием, то есть историю навигации, содержимое корзины, результаты поиска и так далее.

Веб-сессии можно сделать stateful с помощью cookies, которые представляют собой небольшие текстовые файлы, передаваемые сервером клиенту при первом запросе и сохраняемые и представляемые при последующих запросах к домену. Используя cookies, серверы могут отслеживать клиентов по содержимому cookie и отображать содержимое, соответствующее состоянию сессии - это и есть сессия с состоянием.

Другой подход к управлению состоянием - **включение в заголовки HTTP-запросов маркера**, который может использоваться сервером для отслеживания состояния клиента. Включив этот маркер в запрос, можно защитить всю транзакцию с помощью транспортной безопасности.

Свойства REST

1. Uniform Interface
2. Stateless Interactions
3. Cachable
4. Client-Server
5. Layered System

1 Uniform Interface

Принцип единого интерфейса является основополагающим при проектировании любого REST-сервиса. Единый интерфейс упрощает и развязывает архитектуру, что позволяет каждой части развиваться независимо.

Базовые принципы:

- Сервисы представляют ресурсы
- Сообщения, которые говорят сами за себя
- Представление состояния ресурса через HTTP

2 Stateless Interactions

Связь между клиентом и сервером ограничена тем, что между запросами на сервере не сохраняется контекст клиента. Каждый запрос от любого клиента содержит всю информацию, необходимую для обслуживания запроса, а состояние сессии хранится на клиенте.

3 Cachable

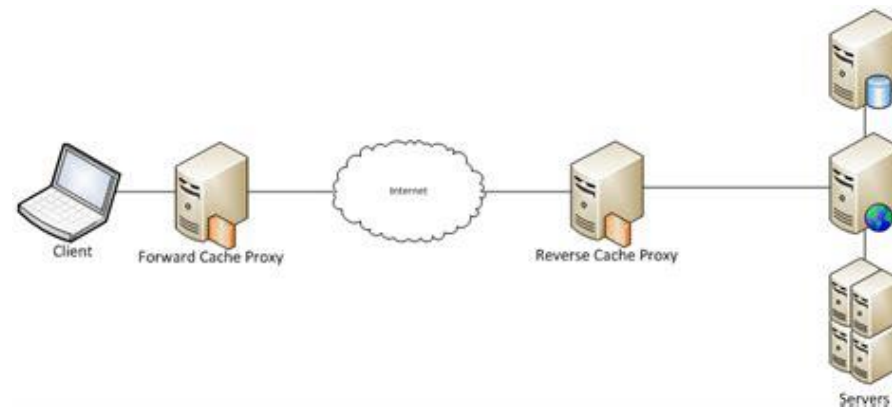
Как и во Всемирной паутине, клиенты и посредники могут кэшировать ответы. Поэтому ответы должны неявно или явно определять себя как кэшируемые или нет, чтобы предотвратить повторное использование клиентами устаревших или несоответствующих данных в ответ на последующие запросы.

4 Client-Server

- Это разделение обязанностей означает, что **клиенты не занимаются такими вопросами, как хранение данных**, которое остается внутренним для каждого сервера, что улучшает переносимость клиентского кода.
- **Серверы не имеют отношения к пользовательскому интерфейсу** или состоянию пользователя, поэтому серверы могут быть более простыми и масштабируемыми.
- **Серверы и клиенты также могут заменяться и разрабатываться независимо друг от друга**, при условии, что интерфейс между ними не изменяется.

5 Layered System

- Клиент обычно не может определить, подключен ли он непосредственно к конечному серверу или к посреднику на этом пути.
- Серверы-посредники могут улучшить масштабируемость системы, обеспечивая балансировку нагрузки и предоставляя общие кэши.
- Они также могут обеспечивать соблюдение политик безопасности.



Пример

Запрос

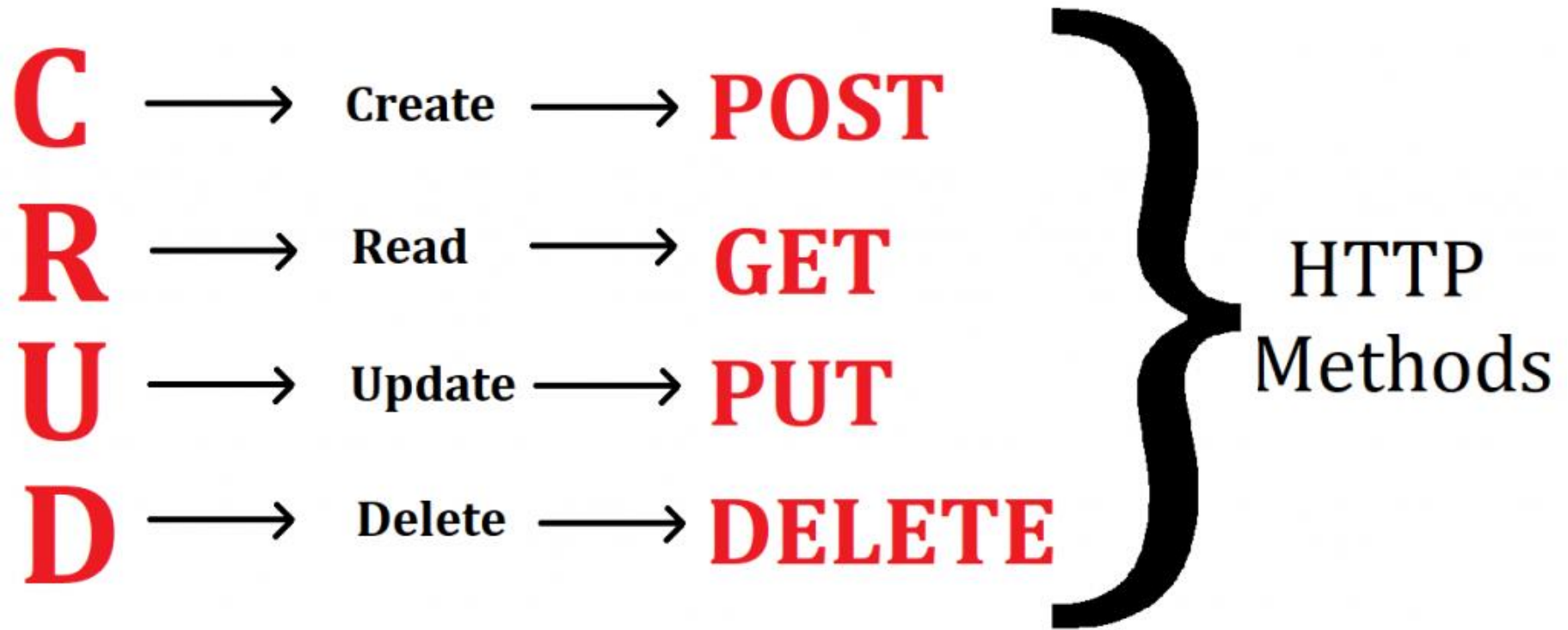
```
GET /account/12345 HTTP/1.1
Host: somebank.org
Accept: application/xml
...
```

Ответ

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">100.00</balance>
  <link rel="deposit" href="http://somebank.org/account/12345/deposit" />
  <link rel="withdraw" href="http://somebank.org/account/12345/withdraw" />
  <link rel="transfer" href="http://somebank.org/account/12345/transfer" />
  <link rel="close" href="http://somebank.org/account/12345/close" />
</account>
```

<https://en.wikipedia.org/wiki/HATEOAS>



Интерпретация методов HTTP

HTTP Method
позволяет
идентифицировать
операцию с
ресурсом

Method	URI Template	Эквивалентная SOA операция
POST	users/{username}	createUserAccount
GET	users/{username}	getUserAccount
PUT	users/{username}	updateUserAccount
DELETE	users/{username}	deleteUserAccount
GET	users/{username}/profile	getUserProfile
POST	users/{username}/bookmarks	createBookmark
PUT	users/{username}/bookmarks/{id}	updateBookmark
DELETE	users/{username}/bookmarks/{id}	deleteBookmark
GET	users/{username}/bookmarks/{id}	getBookmark
GET	users/{username}/bookmarks?tag={tag}	getUserBookmarks
GET	{username}?tag={tag}	getUserPublicBookmarks
GET	?tag={tag}	getPublicBookmarks

GET

GET используется клиентом для получения информации о запрашиваемом ресурсе, идентифицируемом URI запроса. Запросы, использующие GET, должны только получать данные и не должны изменять их каким-либо образом.

```
GET https://www.foo.com/customers
```

```
GET https://www.foo.com/customers/{customerId}
```

POST

POST обычно используется для создания нового ресурса. В частности, он используется для создания подресурса, подчиненного родительскому ресурсу, идентифицированному URI запроса.

```
POST http://www.foo.com/customers HTTP/1.1
{
  "customers": {
    "customerId": "12345",
    "customerName": "Brajesh De", "Address": {
      "AddressLine1": "206 Lane 1", "AddressLine2": "22
Cross", "City": "Bangalore",
      "State": "Karnataka" }
    }
}
```

PUT

Метод PUT обычно используется для обновления существующего ресурса, идентифицированного URI запроса. Если ресурс, идентифицированный URI запроса, существует, то полезная нагрузка сообщения должна рассматриваться как измененная версия существующего ресурса. Если ресурс не существует, а URI может быть определен как новый ресурс, сервер может создать новый ресурс с помощью информации, предоставленной в полезной нагрузке сообщения.

Рекомендуется использовать POST для создания новых ресурсов и PUT для обновления уже существующего ресурса. **Используйте POST, если сервер отвечает за создание имени или ID ресурса** и, следовательно, URI нового ресурса.

PUT может использоваться для создания нового ресурса только в том случае, если клиент отвечает за определение нового URI (через имя ресурса или ID) для ресурса. Глагол POST следует использовать, если клиент не знает или не должен знать результирующий URI нового ресурса до его создания.

DELETE

DELETE используется для удаления ресурса, представленного URI запроса.

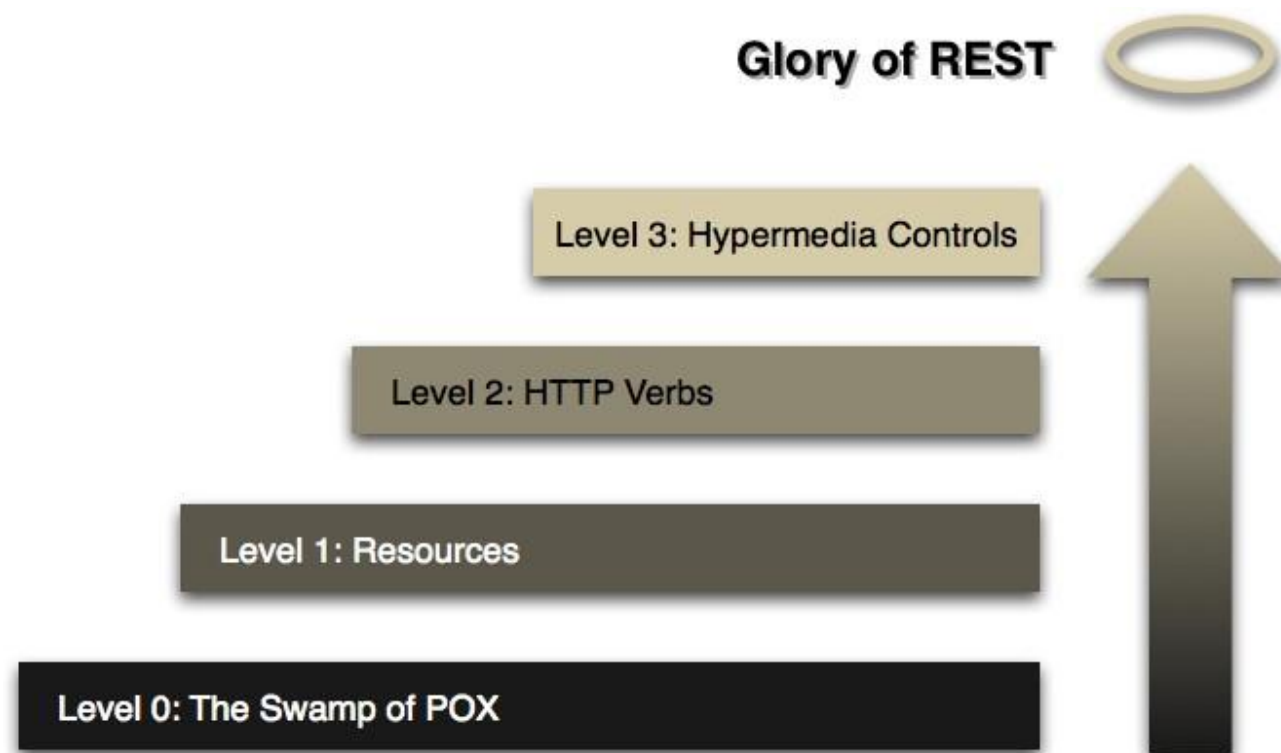
PATCH

Метод PATCH был добавлен в спецификации HTTP в марте 2010 года. Этот метод похож на метод PUT и может использоваться для обновления существующего определения ресурса. Разница между PUT и PATCH заключается в том, что PATCH можно использовать для частичного обновления существующего определения ресурса, в то время как PUT выполняет полное обновление. В методе PATCH для обновления могут быть указаны только определенные атрибуты ресурса.

```
PATCH http://www.foo.com/customers/12345
HTTP/1.1 {
  "customers": {
    "Address": {
      "AddressLine1": "205 Lane 2" }
    }
}
```

Модель зрелости Ричардсона

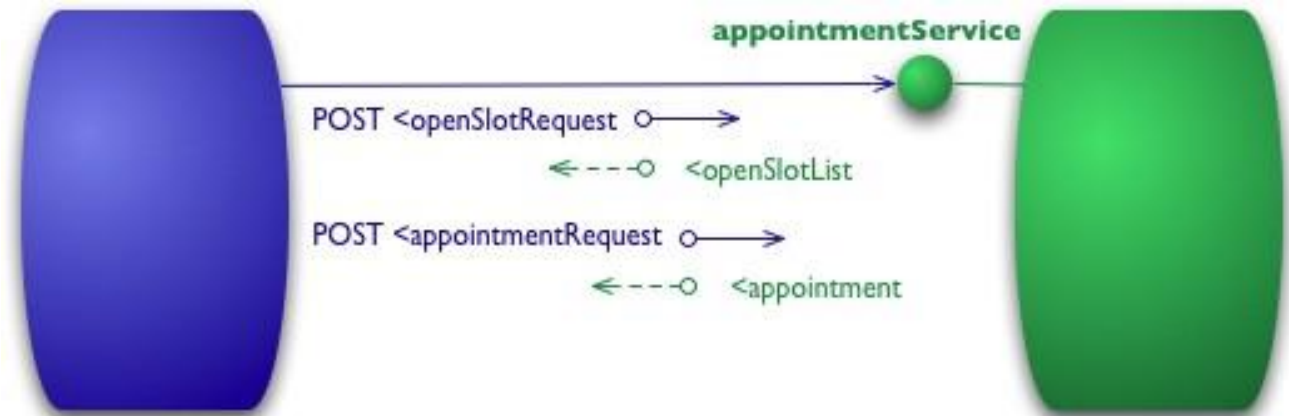
<https://martinfowler.com/articles/richardsonMaturityModel.html>



Уровень о

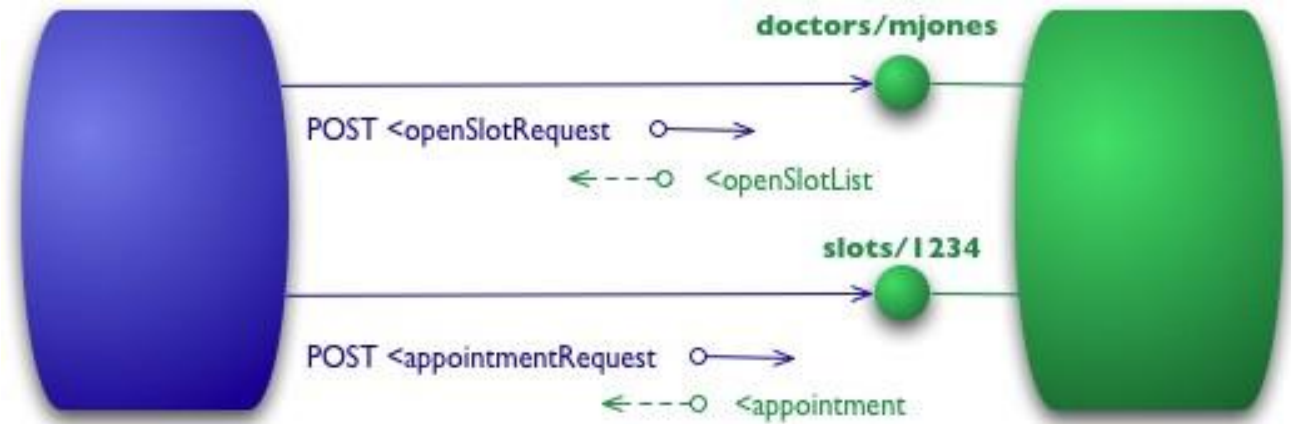
Использование HTTP как
транспортного протокола.

Все запросы идут на 1 endpoint и
одним методом



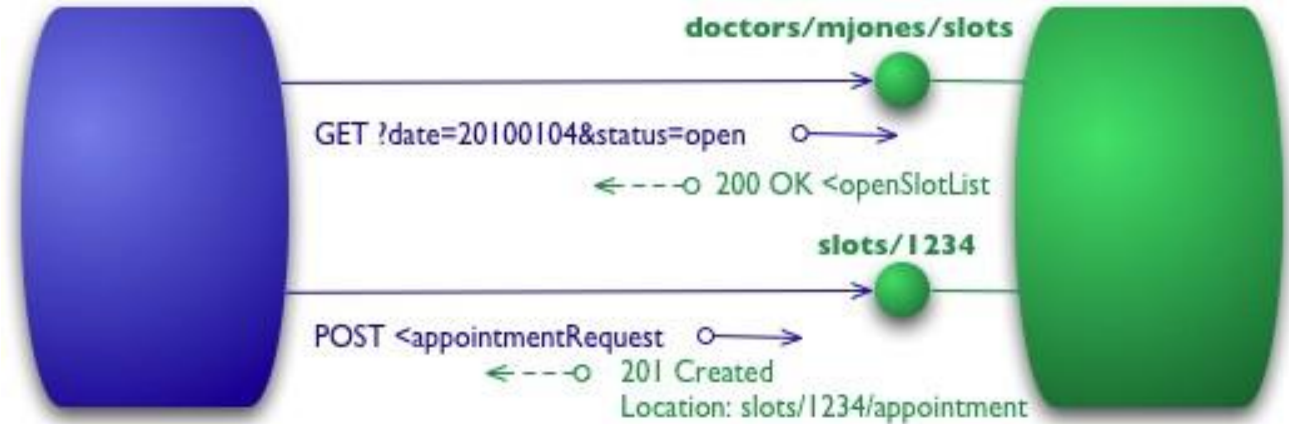
Уровень 1

Использование разных ресурсов для разных сущностей, но с одним методом



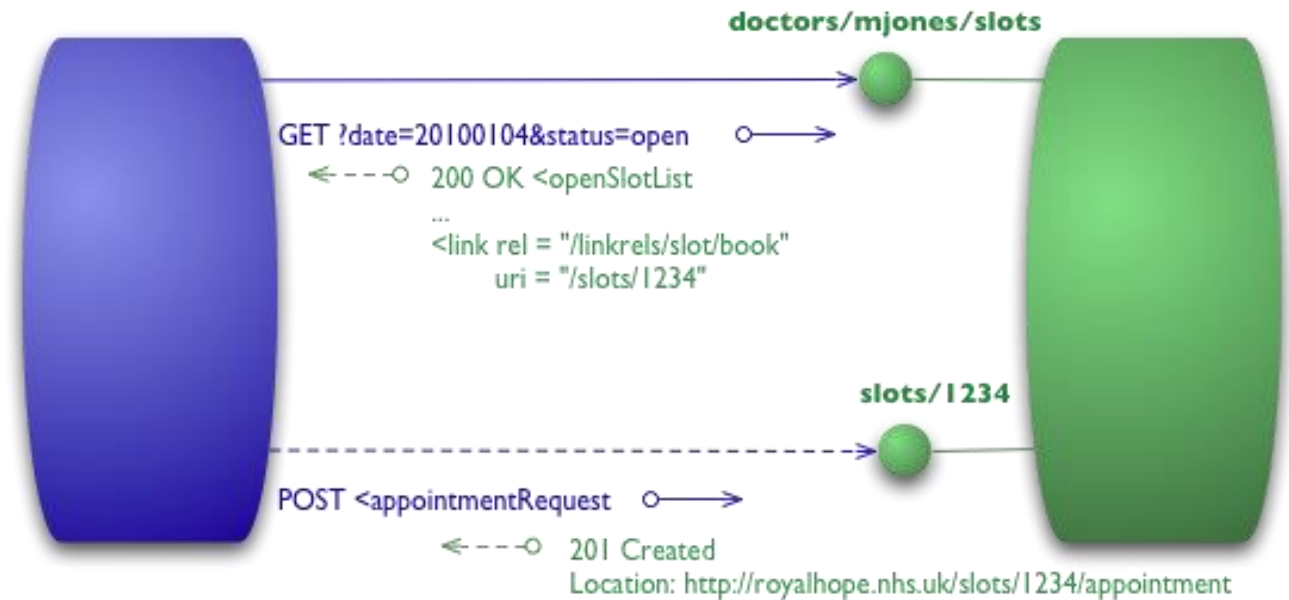
Уровень 2

Использование разных ресурсов
для разных сущностей.
Использование разных методов



Уровень 3

- Использование разных ресурсов для разных сущностей.
- Использование разных методов
- Использование HATEOS (hypertext as the engine of app state)



Python

ИСПОЛЬЗОВАНИЕ FastAPI

В этом примере мы создаем экземпляр FastAPI и определяем путь / с методом GET. Когда мы отправляем GET-запрос на этот путь, функция `read_root` возвращает JSON-ответ `{"Hello": "World"}`.

```
# main.py
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}
```

```
> uvicorn main:app --reload
```

Теперь, когда сервер запущен, вы можете открыть браузер и перейти по адресу `http://127.0.0.1:8000/`, чтобы увидеть ответ от вашего приложения.

Тестируем
производительность REST
сервиса

Скачиваем и компилируем <https://github.com/wg/wrk>

```
Usage: wrk <options> <url>
```

```
Options:
```

```
-c, --connections <N>  Connections to keep open
-d, --duration      <T>  Duration of test
-t, --threads       <N>  Number of threads to use

-s, --script        <S>  Load Lua script file
-H, --header        <H>  Add header to request
    --latency                Print latency statistics
    --timeout               <T>  Socket/request timeout
-v, --version                Print version details
```

```
Numeric arguments may include a SI unit (1k, 1M, 1G)
```

```
Time arguments may include a time unit (2s, 2m, 2h)
```

Как возвращать ошибки?

КОДЫ

- 400 Bad Request – ошибка в теле запроса (например, недостаточно параметров)
- 401 Unauthorized – ошибка в аутентификации
- 403 Forbidden – ошибка в авторизации
- 404 Not Found – ресурс не найден
- 500 Internal Server Error – в сервере произошла ошибка при обработке запроса
- 503 Service Unavailable – в настоящий момент сервис недоступен

Как возвращать ошибки?

тело ответа

- **type** – URI который категоризирует ошибку
- **title** – короткое описание ошибки
- **status** –HTTP code (опционально)
- **detail** – полное описание ошибки
- **instance** – URI запроса, по которому произошла ошибка

```
{  
    "type": "/errors/incorrect-user-pass",  
    "title": "Incorrect username or password.",  
    "status": 401,  
    "detail": "Authentication failed due to  
incorrect username or password.",  
    "instance": "/login/log/abc123"  
}
```

<https://www.rfc-editor.org/rfc/rfc7807>

Open API

<https://swagger.io/docs/specification/about/>

Для чего нужна документация API

- Быстрое начало работы пользователей
- Включает полезную и актуальную информацию
- Предоставляет примеры кода
- Документирует список конечных точек REST
- Документирует содержимое сообщения
- Описывает коды состояния ответа и сообщения об ошибках

Open API структура

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Simple Artist API
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://example.io/v1

# Basic authentication
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

paths: {}
```

Title of the API

Server URL where API is hosted

Security Scheme of the API

Описание end-point

```
paths:
  /artists:
    get:
      description: Returns a list of artists
      # ----- Added lines -----
      parameters:
        - name: limit
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: offset
          in: query
          description: Specifies the page number of the artists to be displayed
          schema:
            type: integer
      # ---- /Added lines -----
```

URI information for the API

HTTP verb to invoke the API

Query parameters for the API

Response Codes

```
responses:
  '200':
    description: Successfully created a new artist
  '400':
    description: Invalid request
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
```

Response Code

Error Response Code

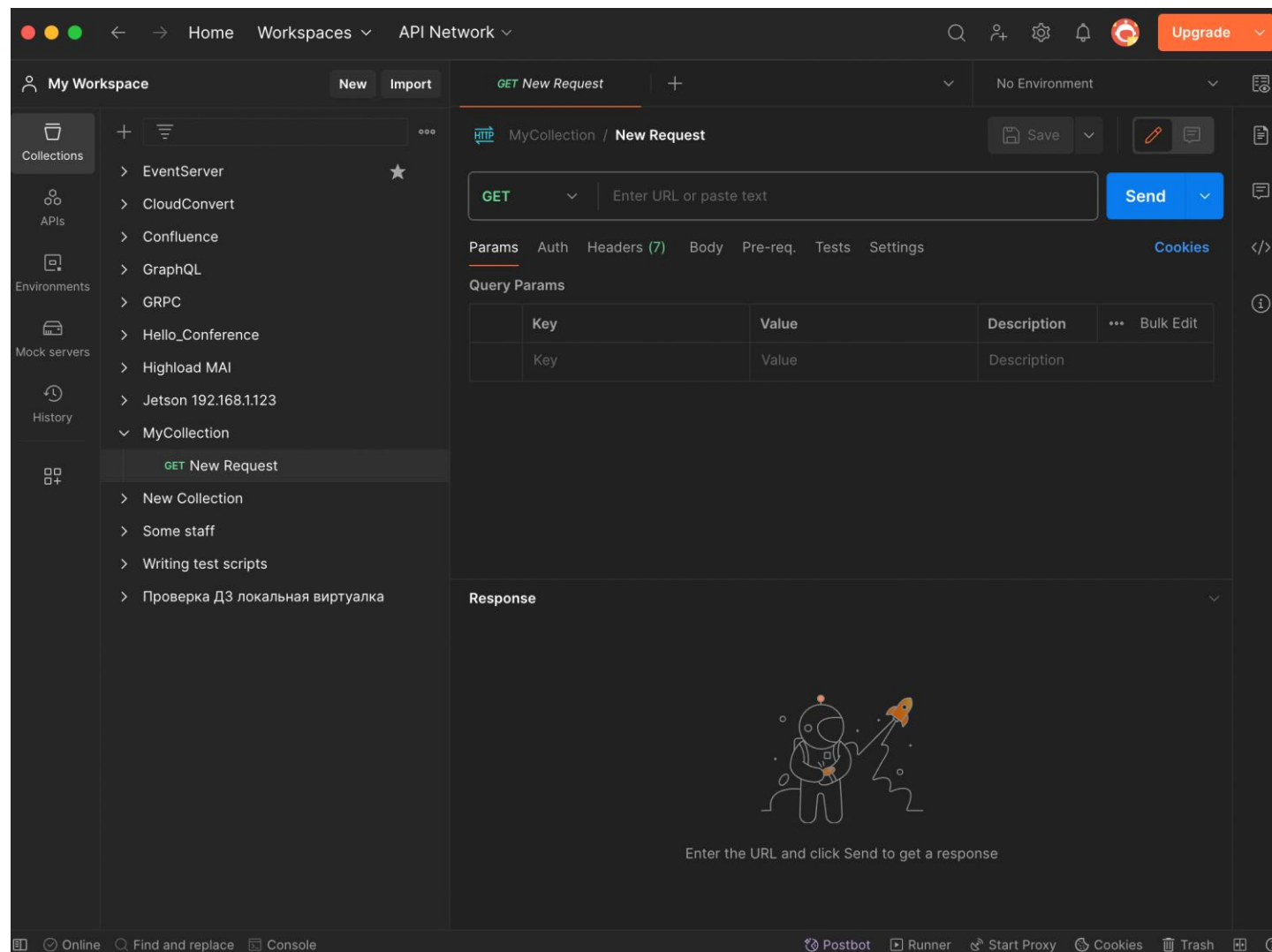
Error Response Payload

Пример

```
openapi: '3.0.0'
info:
  version: '1.0.0'
  title: 'Sample Server'
  description: Demo of REST API
servers:
  - url: http://192.168.64.6:8080
    description: Sample server
paths:
  /request:
    parameters:
      - in: query
        name: session_id
        description: The unique identifier of some session
        required: true
        schema:
          $ref: '#/components/schemas/SessionId'
    get:
      summary: Read a session
      responses:
        '200':
          description: The session corresponding to the provided `sessionId`
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Sessions'
```

Как протестировать API?

<https://www.postman.com/>



RESTful проблемы

- Фильтрация и querying
- Работа с иерархией ресурсов в рамках одного запроса
- Пагинация
- Ссылки на другие сущности
- Асинхронные запросы
- Батч запросы
- Формат сообщений об ошибках
- Семантику HTTP кодов в приложении к конкретным ситуациям

JSON-API

JSON-API – спецификация на RESTful API на JSON.

Эта спецификация использует максимально возможности HTTP и отвечает на вопросы:

- Формат ответа
- Фильтрация, *quering*, пагинация
- Ссылки на связанные ресурсы
- Обновление, удаление ресурсов
- Формат ошибок
- Семантика HTTP кодов

<https://jsonapi.org>

ODATA

ODATA – очень развернутый и максимально REST-like протокол, который специфицирует все что только можно. В том числе определяет batch запросы.

- <https://www.odata.org>

Access Control

No authentication

Публичные API, предназначенные для доступа только для чтения, могут не требовать аутентификации, если они предназначены для анонимного (неаутентифицированного) доступа. Обычно такие конечные точки используются для получения информации или статуса, например, API статуса для онлайн-сервиса.

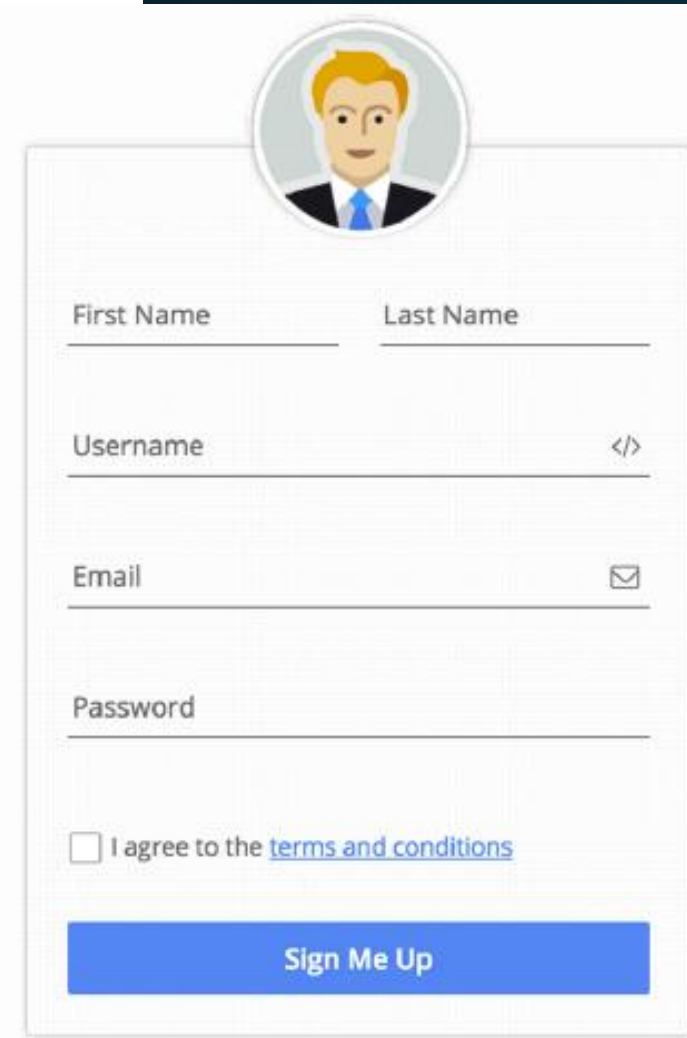
Authenticate users.

Определяет что пользователь действительно тот, которым назвался. Например, различные схемы с паролями, цифровые сертификаты, биометрические сканеры.

Аутентификация по паролю

Этот метод основывается на том, что пользователь должен предоставить username и password для успешной идентификации и аутентификации в системе.

Пара username/password задается пользователем при его регистрации в системе, при этом в качестве username может выступать адрес электронной почты пользователя.



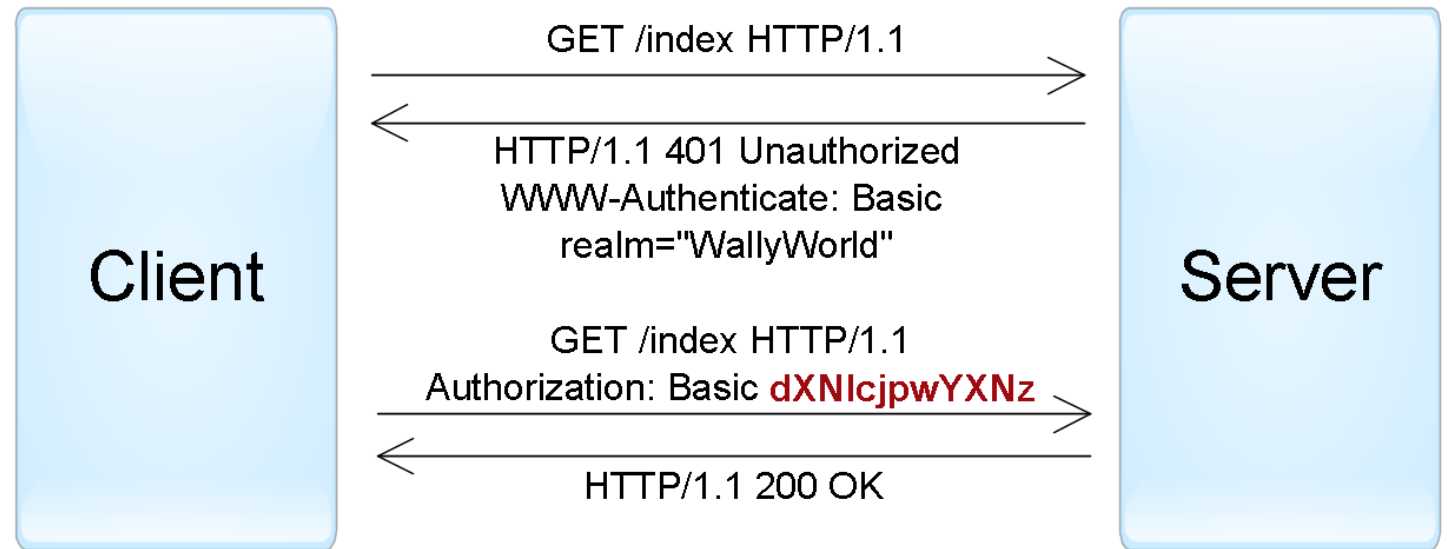
A user registration form with a circular profile picture placeholder at the top showing a man with blonde hair in a suit. Below the picture are input fields for 'First Name' and 'Last Name'. A 'Username' field with a code icon (</>) is next. An 'Email' field with an envelope icon is next. A 'Password' field is next. Below these is a checkbox labeled 'I agree to the [terms and conditions](#)'. At the bottom is a blue button labeled 'Sign Me Up'.

First Name	Last Name
Username </>	
Email	
Password	
<input type="checkbox"/> I agree to the terms and conditions	
Sign Me Up	

Пример: HTTP authentication

- Сервер, при обращении неавторизованного клиента к защищенному ресурсу, отправляет HTTP статус “**401 Unauthorized**” и добавляет заголовок “**WWW-Authenticate**” с указанием схемы и параметров аутентификации.
- Браузер, при получении такого ответа, автоматически показывает диалог ввода **username** и **password**. Пользователь вводит детали своей учетной записи.
- Во всех последующих запросах к этому веб-сайту браузер автоматически добавляет HTTP заголовок “**Authorization**”, в котором передаются данные пользователя для аутентификации сервером.
- Сервер аутентифицирует пользователя по данным из этого заголовка. Решение о предоставлении доступа (авторизация) производится отдельно на основании роли пользователя, ACL или других данных учетной записи.

Basic http authorization & authorization



Base64 encoding username:password

проблема с basic http

Передача учетных данных в виде чистого текста: Если комбинация имени пользователя/пароля в формате Base64 передается по HTTP, ее можно легко перехватить в процессе передачи.

- Атаки с набиванием паролей: Злоумышленник может легко перебрать логин, используя словари паролей.

- Уязвимость к атакам повторного воспроизведения: Это связано с тем, что комбинация имени пользователя/пароля в формате Base64 идентична при каждом вычислении одного и того же мандата. Захватив это значение, его можно легко воспроизвести против сервера в более позднее время.

- Подделка сервера: У клиента нет возможности проверить подлинность сервера, что позволяет осуществлять поддельные атаки.

- Уязвимость для атак типа «машина посередине»: При использовании HTTP возможны атаки типа «человек посередине».

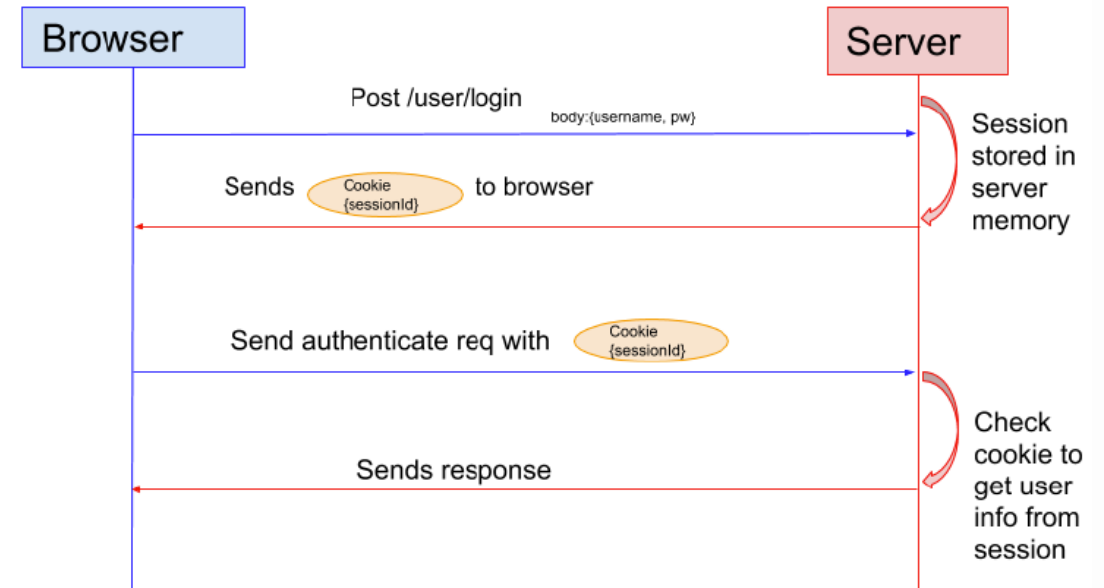
НМАС

НМАС (Hash-based Message Authentication Code) — это метод аутентификации сообщений, который использует хеш-функцию в сочетании с секретным ключом. НМАС обеспечивает целостность и аутентичность данных, гарантируя, что сообщение не было изменено и что оно действительно пришло от предполагаемого отправителя.

- 1. Секретный ключ:** Оба участника обмена сообщениями (отправитель и получатель) знают секретный ключ, который не должен быть известен третьим лицам.
- 2. Хеш-функция:** Используется криптографическая хеш-функция, такая как SHA-256, SHA-512, MD5 и другие.
- 3. Создание НМАС:**
 1. Отправитель объединяет сообщение и секретный ключ.
 2. Применяет хеш-функцию к объединенным данным.
 3. Результат хеширования (НМАС) добавляется к сообщению.
- 4. Проверка НМАС:**
 1. Получатель разделяет сообщение и НМАС.
 2. Применяет ту же хеш-функцию к сообщению с использованием секретного ключа.
 3. Сравнивает результат с полученным НМАС.
 4. Если результаты совпадают, сообщение считается аутентичным и не измененным.

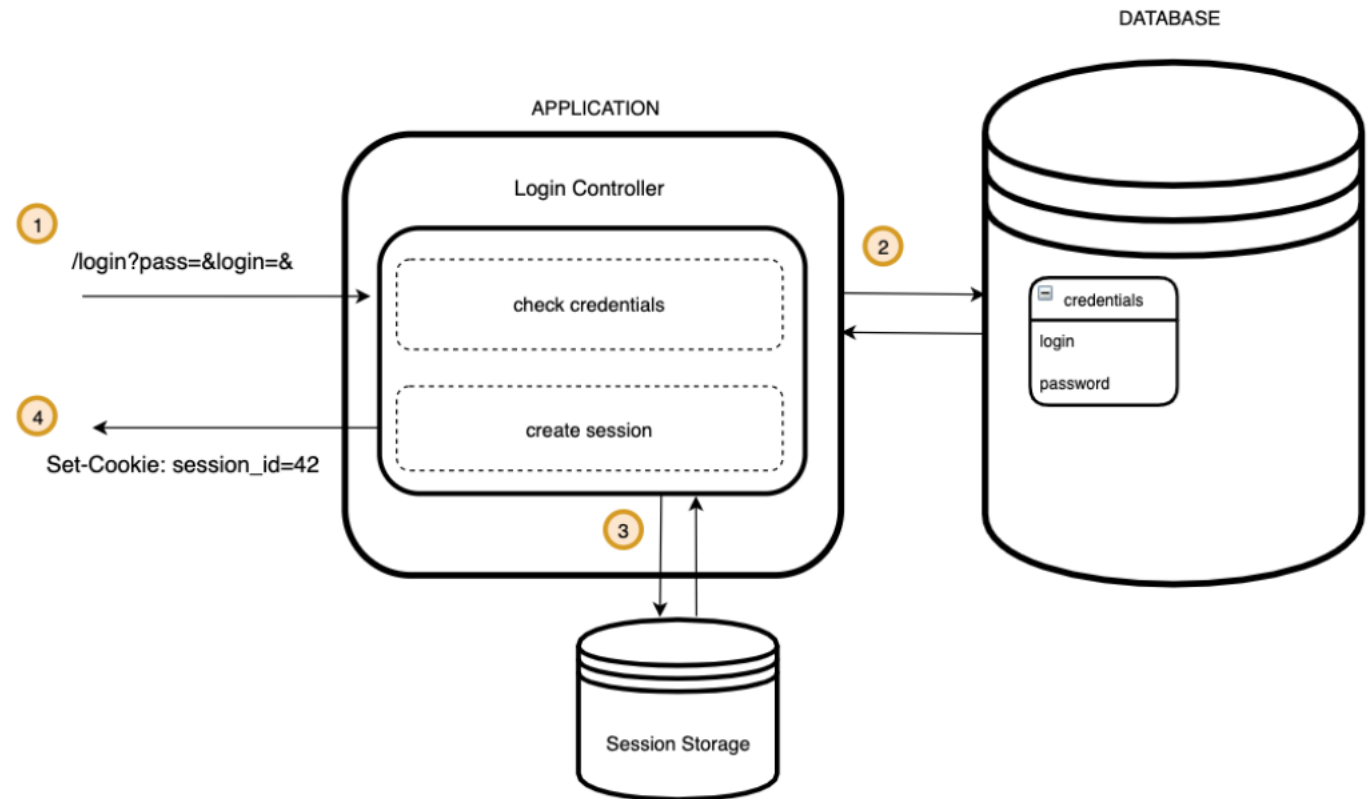
Session cookies

- Пользователь с каждым запросом присылает сессионную куку
- Приложение по сессионной куке находит данные по сессии, в том числе пользователя, время жизни и т.д.
- Когда пользователь нажимает «Выход», приложение удаляет сессию на сервере и удаляет куку на клиенте.
- Если кука отсутствует или данные отсутствуют, пользователь считает неавторизованным и его перекидывает на страничку авторизации.



Аутентификация на стороне сервера

- ✓ хранилище для данных пользователя
- ✓ хранилище для данных сессий



- Password никогда не храните в открытом виде в БД. Храните hash от пароля с солью
- Не передавайте login/password в открытом виде по HTTP, только по HTTPS

Проблемные области в аутентификации по паролю [1/2]

1. Веб-приложение позволяет пользователям создавать простые пароли.
2. Веб-приложение не защищено от возможности перебора паролей (brute-force attacks).
3. Веб-приложение само генерирует и распространяет пароли пользователям, однако не требует смены пароля после первого входа (т.е. текущий пароль где-то записан).
4. Веб-приложение допускает передачу паролей по незащищенному HTTP-соединению либо в строке URL.
5. Веб-приложение не использует безопасные хэш-функции для хранения паролей пользователей.
6. Веб-приложение не предоставляет пользователям возможность изменения пароля либо не уведомляет пользователей об изменении их паролей.

Проблемные области в аутентификации по паролю [2/2]

7. Веб-приложение использует уязвимую функцию восстановления пароля, которую можно использовать для получения несанкционированного доступа к другим учетным записям.
8. Веб-приложение не требует повторной аутентификации пользователя для важных действий: смена пароля, изменения адреса доставки товаров и т. п.
9. Веб-приложение создает session tokens таким образом, что они могут быть подобраны или предсказаны для других пользователей.
10. Веб-приложение допускает передачу session tokens по незащищенному HTTP-соединению, либо в строке URL.
11. Веб-приложение уязвимо для session fixation-атак (т. е. не заменяет session token при переходе анонимной сессии пользователя в аутентифицированную).
12. Веб-приложение не устанавливает флаги HttpOnly и Secure для browser cookies, содержащих session tokens.
13. Веб-приложение не уничтожает сессии пользователя после короткого периода неактивности либо не предоставляет функцию выхода из аутентифицированной сессии.

Использование https

1. Подключение к серверу: Клиент (браузер) отправляет запрос на подключение к HTTPS-серверу.
2. Выбор протокола и алгоритмов: Клиент и сервер договариваются о версии протокола HTTPS и используемых криптографических алгоритмах.
3. Сертификат SSL/TLS: Сервер отправляет клиенту свой сертификат SSL/TLS, который содержит информацию о сервере и его публичный ключ.
4. Проверка сертификата: Клиент проверяет сертификат на валидность, например, с помощью системы доверия, такой как корневой центр сертификации.
5. Генерация сессионного ключа: Клиент генерирует случайный сессионный ключ, который будет использоваться для шифрования данных во время сеанса.
6. Обмен ключами: Клиент шифрует сессионный ключ с помощью публичного ключа сервера и отправляет его на сервер.
7. Шифрование данных: Сервер расшифровывает полученный сессионный ключ и начинает использовать его для шифрования и дешифрования данных, которые обмениваются между клиентом и сервером.
8. Завершение соединения: После завершения обмена данными клиент и сервер могут завершить соединение.

- помогает удостовериться что мы подключились к нужному серверу
- защищает от подмена/перехвата информации

Многофакторная аутентификация (MFA)

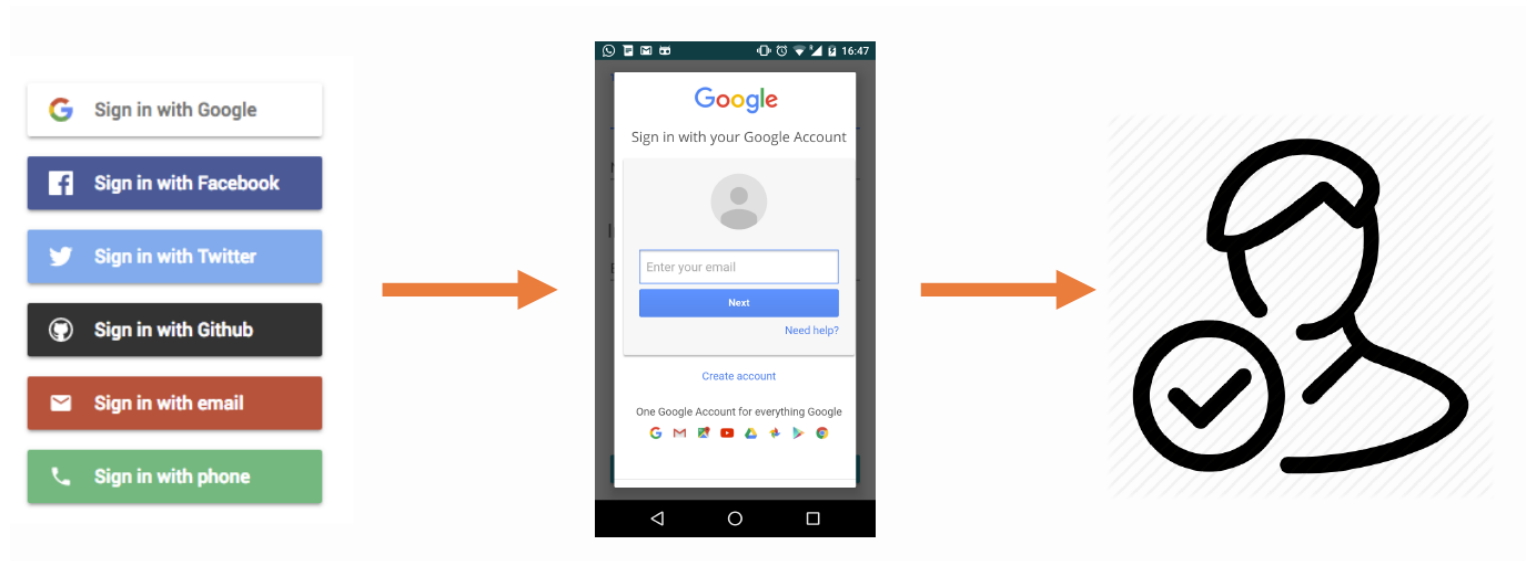
В многофакторной аутентификации человек должен представить два или более факторов аутентификации.

Примеры факторов:

1. **Фактор знания:** что-то, что человек знает, например пароль или коэффициент PIN
2. **Фактор владения:** что-то, что есть у человека, например, сотовый телефон, который может получить код, или идентификационная карточка компании
3. **Фактор наследования:** Что-то, что определяет конкретные свойства человека, например, использование сканера отпечатков пальцев, считывателя ладоней, сканера сетчатки или другого типа биометрической аутентификации



Использование внешних провайдеров



Стандарты идентификации

OAuth2.0/OpenID Connect

SAML

Самописные флоу

OAuth 2.0 - это фреймворк для организации взаимодействия, которые бы давали возможность не только идентифицировать пользователя внешним провайдером, но давать возможность совершать действия от лица этого пользователя с различным ресурсами. Основная проблема, что OAuth2.0 не стандартизирует способ получения данных о пользователе (Identity). Т.е. OAuth2.0 про аутентификацию и авторизацию.

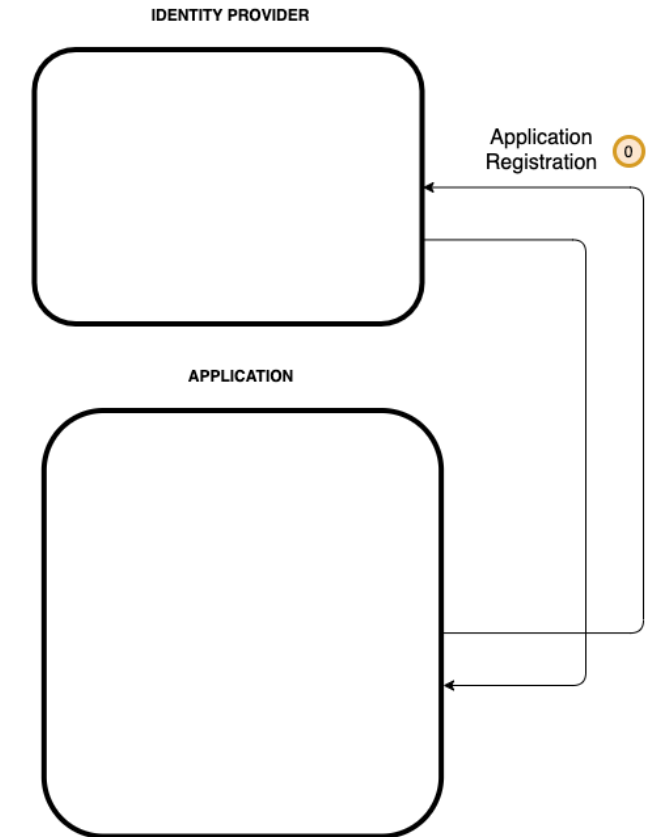


<https://openid.net/developers/certified/>

OpenID Connect (OIDC) решает эту проблему. Добавляется `scope=openid`, и в ответ приходит ID token – это информация о пользователе специальным образом закодированная и подписанная

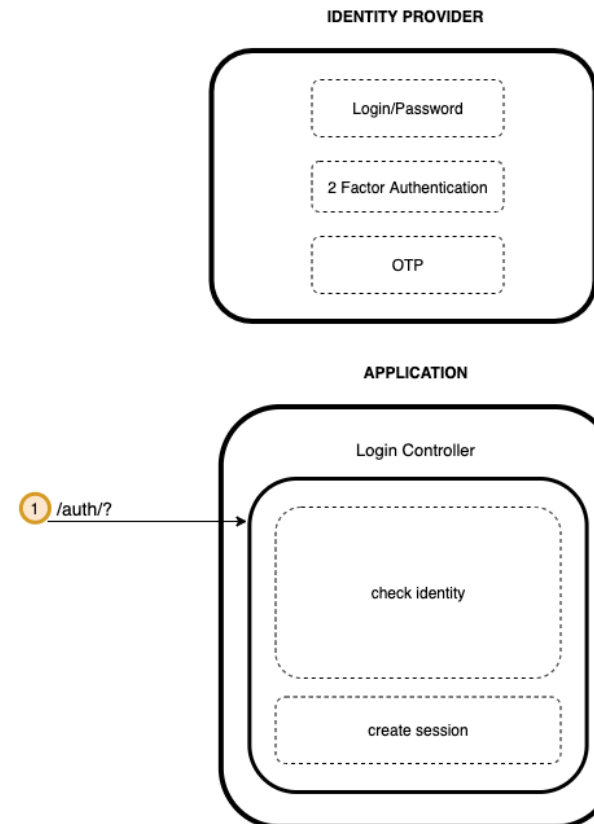
Регистрация приложения у Identity провайдера

- До того, как пришел пользователь, приложение, чтобы работать с Identity Provider, должна зарегистрироваться у него.
- И получить `client_id` – свой идентификатор, и `client_secret` – ключ, с помощью которого провайдер поймет, что запрос идет этого конкретного приложения



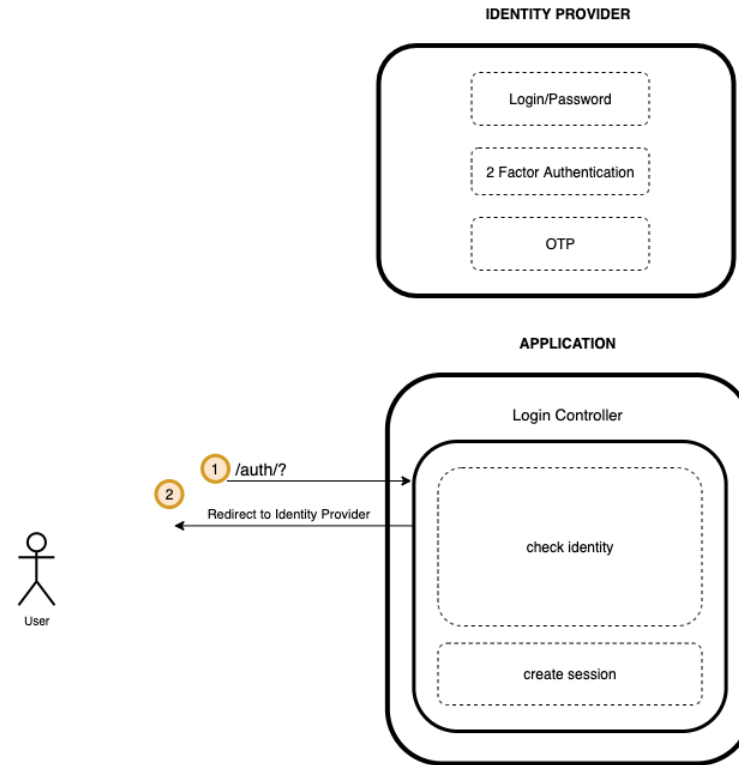
Пользователь делает запрос

- К нам приходит пользователь и делает неавторизованный запрос



redirect пользователя к Identity provider-y

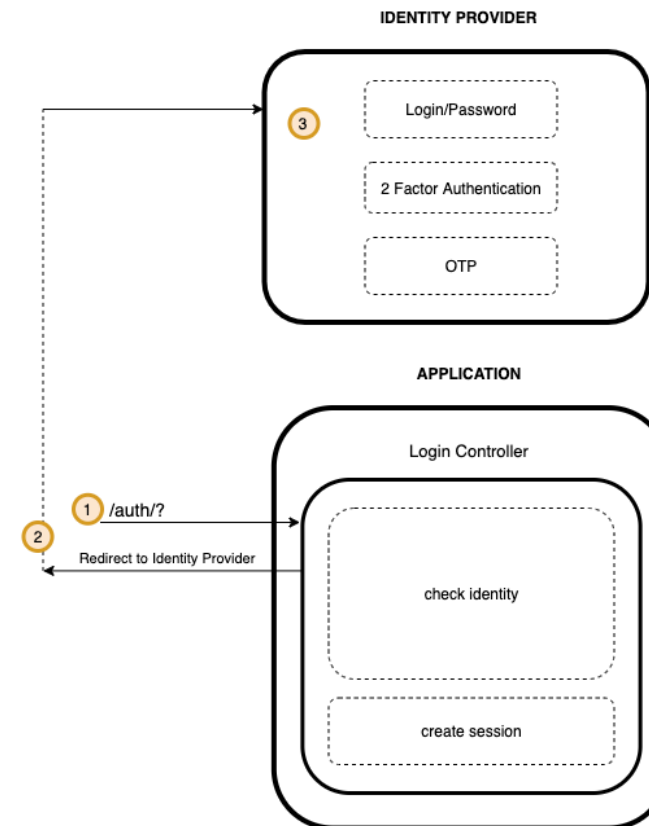
- Как только сервис понял, пользователь пришел неавторизованный, посылает ему редирект к identity провайдеру.
- В запросе к провайдеру приложение передает
 - 1) **client_id** – идентификатор своего приложения
 - 2) **redirect_uri** – url, на котором приложение будет ждать ответ от провайдера
 - 3) **scope** – уровень доступа, который мы хотим получить. Для идентификации openid (можно дополнительно profile/email)
 - 4) **state** – случайное число идентифицирующее клиента



`https://github.com/login/oauth/authorize?response_type=code&client_id=<CLIENT_ID>&redirect_uri=<CALLBACK_ENDPOINT>&state=<STATE>&scope=email%20identity`

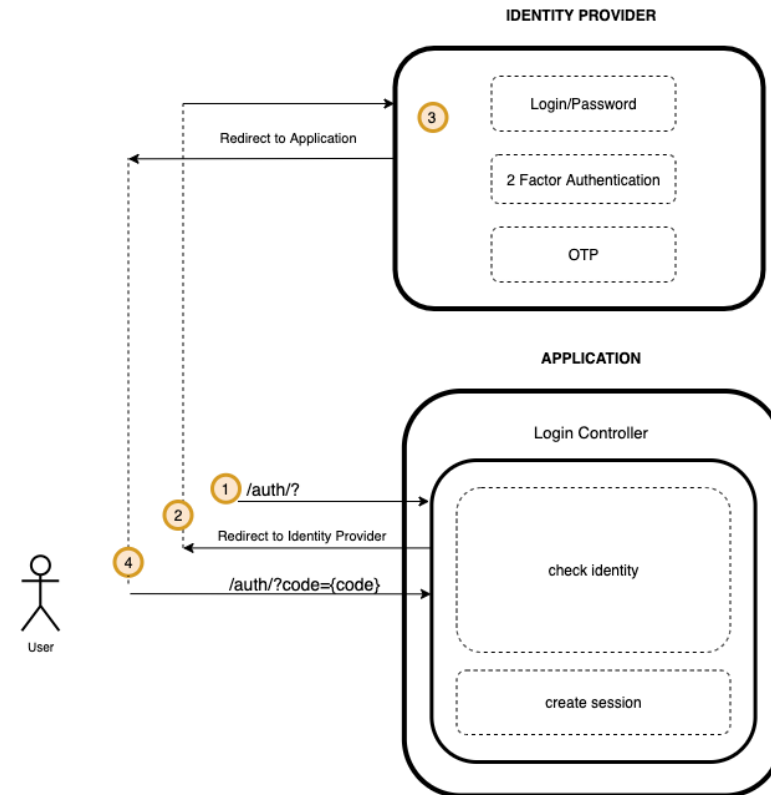
Identity провайдер делает свою работу

- Identity провайдер делает свою работу – он идентифицирует пользователя.
- Способ, которым он это делает – остается на его усмотрение.
- Может быть провайдер увидит, что у пользователя сейчас активна сессия, и пользователь даже не заметит, как он побывал на сайте Провайдера.
- Или наоборот, IdP решит, что надо не только спросить логин и пароль, но и еще код из СМС.



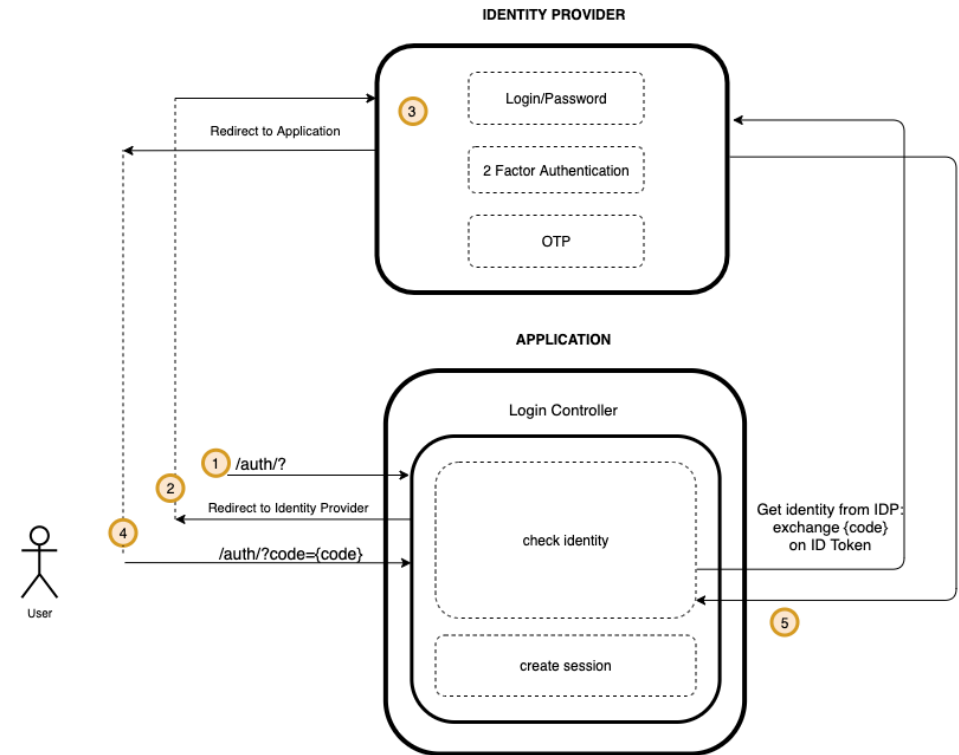
Identity provider redirect обратно в сервис

- После того, как identity провайдер выполнил свою работу и идентифицировал пользователя, он делает редирект обратно в сервис (приложение) и в get-параметрах передает «квиток» - авторизационный код, с помощью которого наше приложение сможет потом у IdP узнать информацию о пользователе.



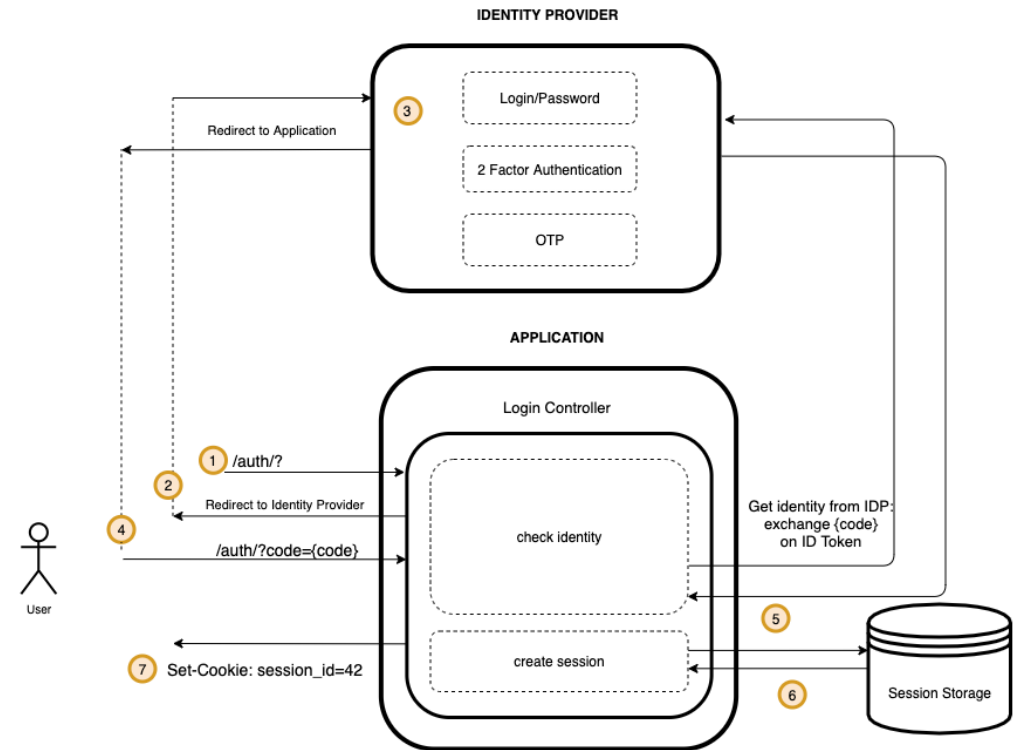
Узнаем у провайдера identity пользователя

- На этом этапе (5) приложение делает запросы к провайдеру, для того, чтобы узнать по «квитку», кому он был выписан.
- При запросе к провайдеру сообщаем ключ своего приложения.
- Обмениваем одноразовый авторизационный токен (квиток) на **access_token**, **refresh_token**, **id_token**
- **Access_token** используется для того, чтобы делать запросы от лица пользователя
- **ID_token** это в специальном виде закодированная и подписанная информация о пользователе.

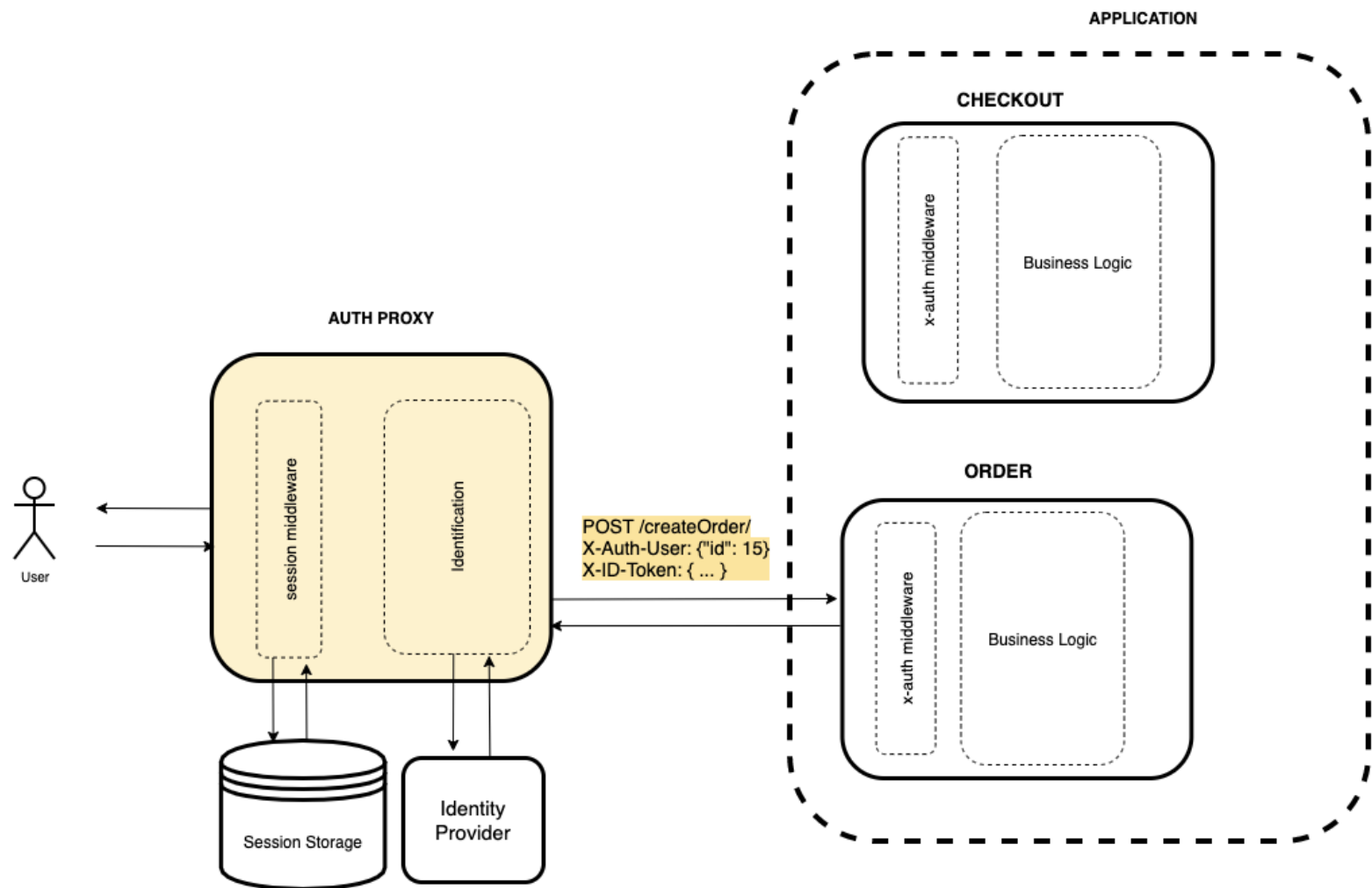


Создаем сессию

- Шаги (7) и (6) не являются частью стандарта, но они должны быть реализованы в нашем приложении.
- Как только получен ID токен, и найден пользователь приложения, создается сессия.

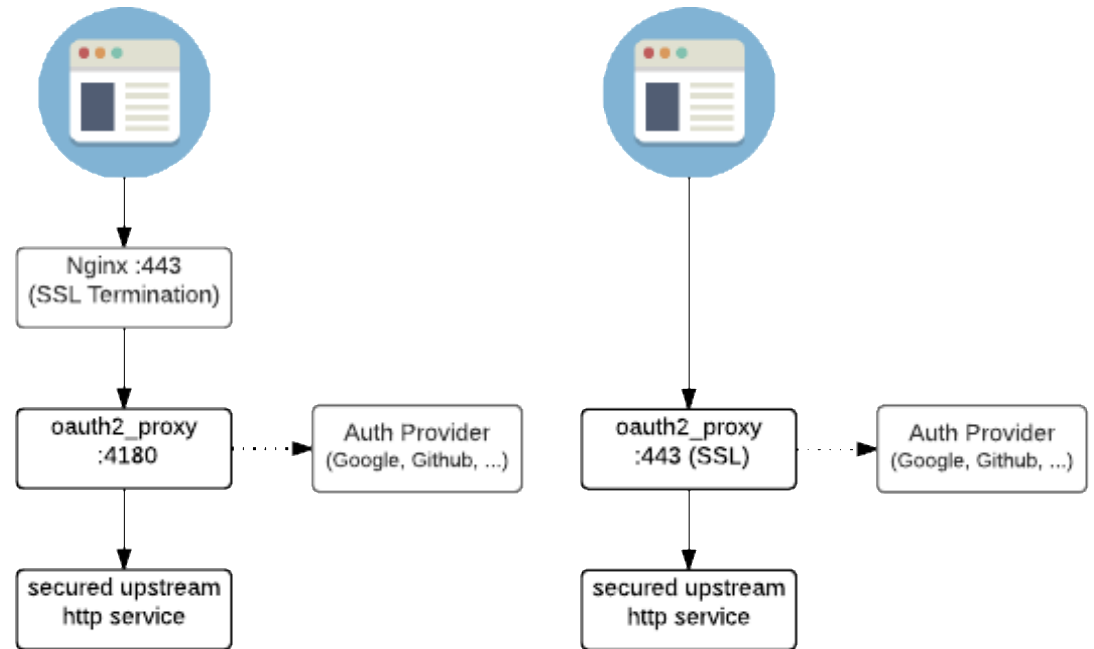


Auth прокси



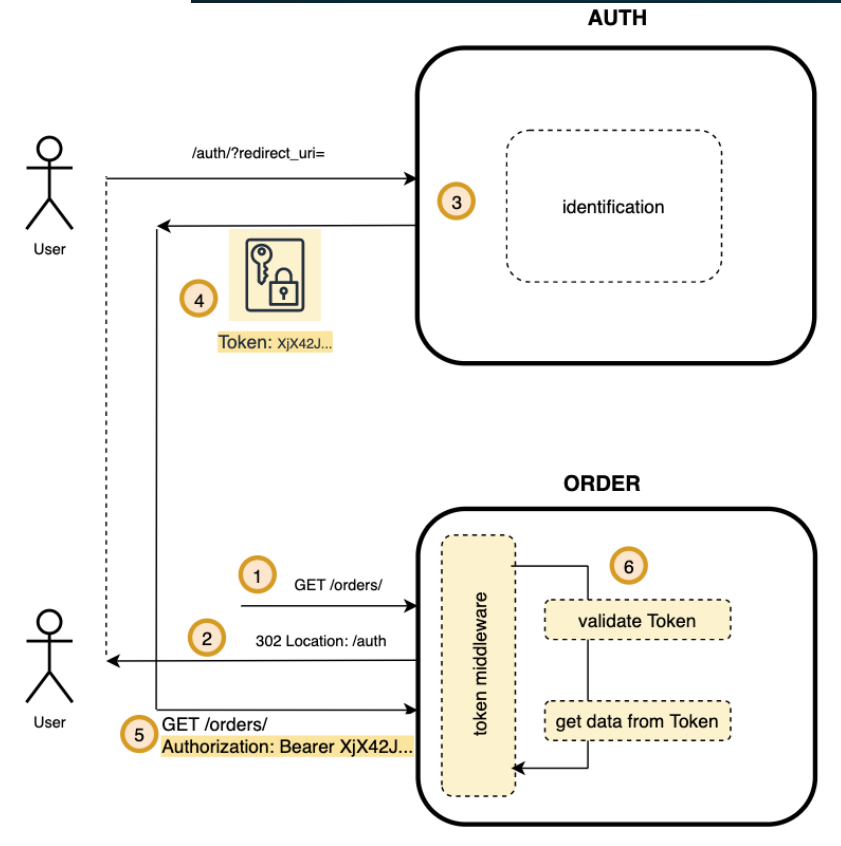
OAuth2-proxy

<https://oauth2-proxy.github.io/oauth2-proxy/>



Token-based сессии

- Чтобы из каждого микросервиса не ходить в сервис Сессий (или в Хранилище), можно использовать механизм сессий, основанных на токенах.
- Сервис, который осуществляет идентификацию отдает данные о сессии или пользователе и подписывает эти данные своим ключом. Такие данные назовем токеном.
- Пользователь использует этот токен для доступа к другим сервисам.
- Сервисам при этом не надо ходить, чтобы проверить сессию. Если токен валиден, то сервис доверяет тем данным, которые он получил из токена.



JWT

- JWT-токен – это просто закодированный в base64url с данными + заголовок с метаданной и подписью.
- JWT = B64(Header).B64(Payload).SIGN

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

HEADER: ALGORITHM & TOKEN TYPE	
<pre>{ "alg": "HS256", "typ": "JWT" }</pre>	Header will include encryption algorithm
PAYLOAD: DATA	
<pre>{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }</pre>	Payload (also referred as claims) will include data that we want to transfer
VERIFY SIGNATURE	
<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret) <input type="checkbox"/> secret base64 encoded</pre>	Signature is created by signing encoded header and payload with signature using header algorithm

JWT Payload

Данные в Payload должны приходить не в произвольном формате, и есть набор специальных или зарегистрированных атрибутов (claims) и возможность использовать свои.

Например:

- iss – кто выпустил токен
 - sub – о ком данный токен
 - exp – дата протухания
- и другие

HEADER: ALGORITHM & TOKEN TYPE	
<pre>{ "alg": "HS256", "typ": "JWT" }</pre>	Header will include encryption algorithm
PAYLOAD: DATA	
<pre>{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }</pre>	Payload (also referred as claims) will include data that we want to transfer
VERIFY SIGNATURE	
<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret) <input type="checkbox"/> secret base64 encoded</pre>	Signature is created by signing encoded header and payload with signature using header algorithm

<https://www.iana.org/assignments/jwt/jwt.xhtml>

<https://jwt.io/#libraries-io>

На сегодня все

ddzuba@yandex.ru