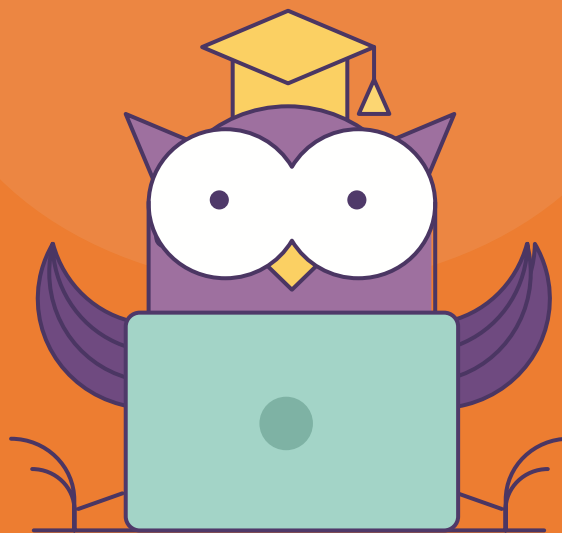


Меня хорошо слышно && видно?



Напишите в чат, если есть проблемы!

Ставьте  если все хорошо

Аутентификация и авторизация

Архитектор ПО



Преподаватель



Непомнящий Евгений

- 15 лет программировал контроллеры на C++ и руководил отделом разработки
- 3 года пишу на Java
- Последнее время пишу микросервисы на Java в Мвидео
- Телеграм @EvgeniyN

Правила вебинара



Активно участвуем



Задаем вопросы в чат



Off-topic обсуждаем в Slack #канал группы или #general



Вопросы вижу в чате, могу ответить не сразу

Карта вебинара

- Паттерны аутентификации в монолитах и микросервисах
- Identity Provider-ы и OIDC
- Token-based аутентификация и JWT

Терминология

Есть большая путаница в терминах аутентификация, авторизация, идентификация.

- **Идентификация** – это когда некий субъект идентифицировал себя
- **Аутентификация** – это процесс, в котором мы удостоверяемся, что субъект действительно тот, кто он есть
- **Авторизация** – это процесс, когда мы уже зная, кем является субъект, проверяем может ли он совершить действие или нет.

Пример

Пропускная система на режимном предприятии.

Заходит человек на КПП, охранник его спрашивает:

О: «Вы кто такой?»

Ч: «Я – Василий Пупкин» - **это идентификация**

О: «Покажите паспорт»

Ч: «Вот он».

Охранник сверяет имя и фамилию, и фотографию. Убеждается, что это действительно Вася Пупкин - это **аутентификация**

Охранник смотрит, что Вася Пупкин есть в списках. И пропускает его – это **авторизация**.

Терминология

Disclaimer:

В веб-сервисах, есть два типа аутентификации: при логине и сессионная аутентификация при каждом запросе.

В рамках нашего занятия, будем называть идентификацией – процесс аутентификации при логине.

Процессы, связанные с аутентификацией

- **Регистрация** – создание нового пользователя, управление анонимными пользователями
- **Идентификация (логин)** – логин пользователя
- **Управление пользователями** и пользовательскими данными – профиль клиента, блокирование, изменение данных и т.д.
- **Сессионная аутентификация** – логат пользователя и управление сессиями
- **Авторизация** – роли и права доступа

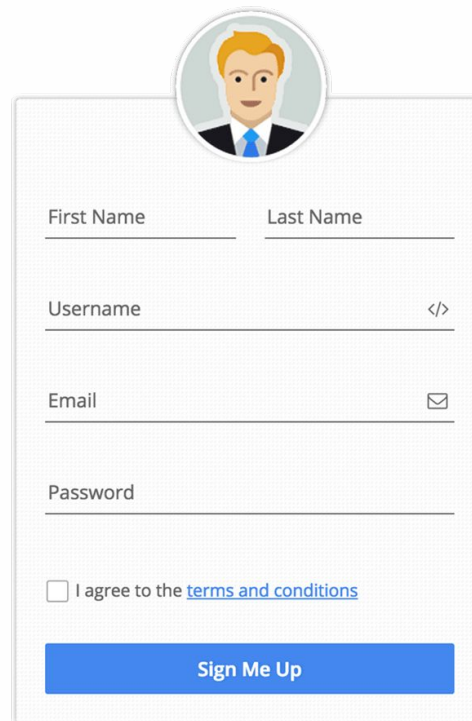
Это разные процессы, в которых используются разные паттерны. И процессы могут быть реализованы в рамках как одного сервиса, нескольких специализированных сервисов, так и как middleware слой в прикладных бизнес сервисах.

01

Паттерны аутентификации в
монолитных приложениях

Регистрация

- Пользователь заходит на страничку регистрации
- Пользователь вводит данные и отправляет форму на сервер
- Приложение создает запись о пользователе



A registration form with a light gray background and rounded corners. At the top center is a circular profile picture placeholder showing a man with orange hair in a suit. Below it are five input fields: 'First Name' and 'Last Name' (split), 'Username' with a code icon, 'Email' with an envelope icon, and 'Password'. At the bottom is a checkbox for 'I agree to the [terms and conditions](#)' and a blue 'Sign Me Up' button.

First Name Last Name

Username </>

Email ✉

Password

☐ I agree to the [terms and conditions](#)

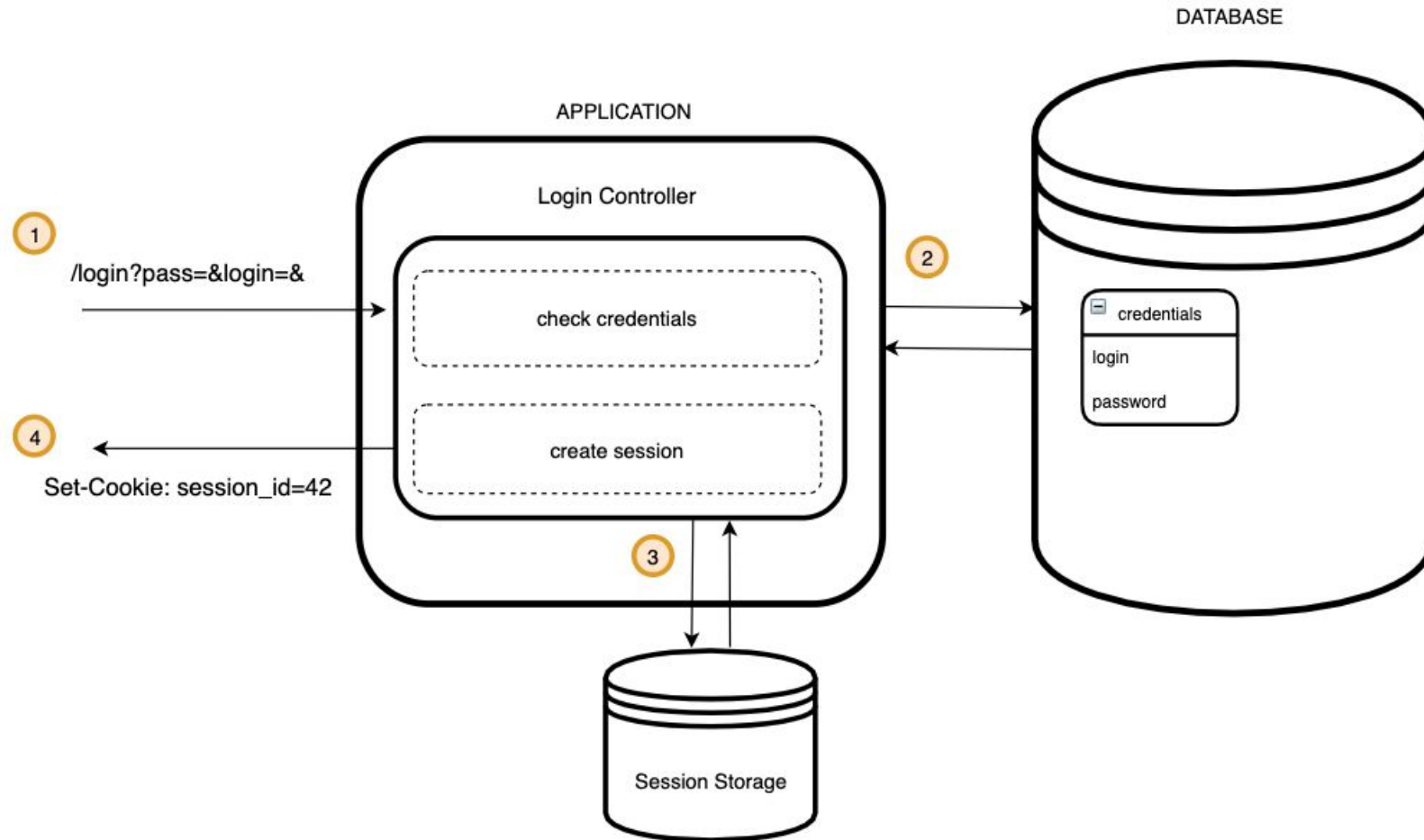
Sign Me Up

Идентификация

- Приложение отправляет неавторизованного пользователя на страничку логина
- Пользователь вводит логин и пароль
- Приложение проверяет, что логин и пароль правильные
- Приложение устанавливает сессионную куку

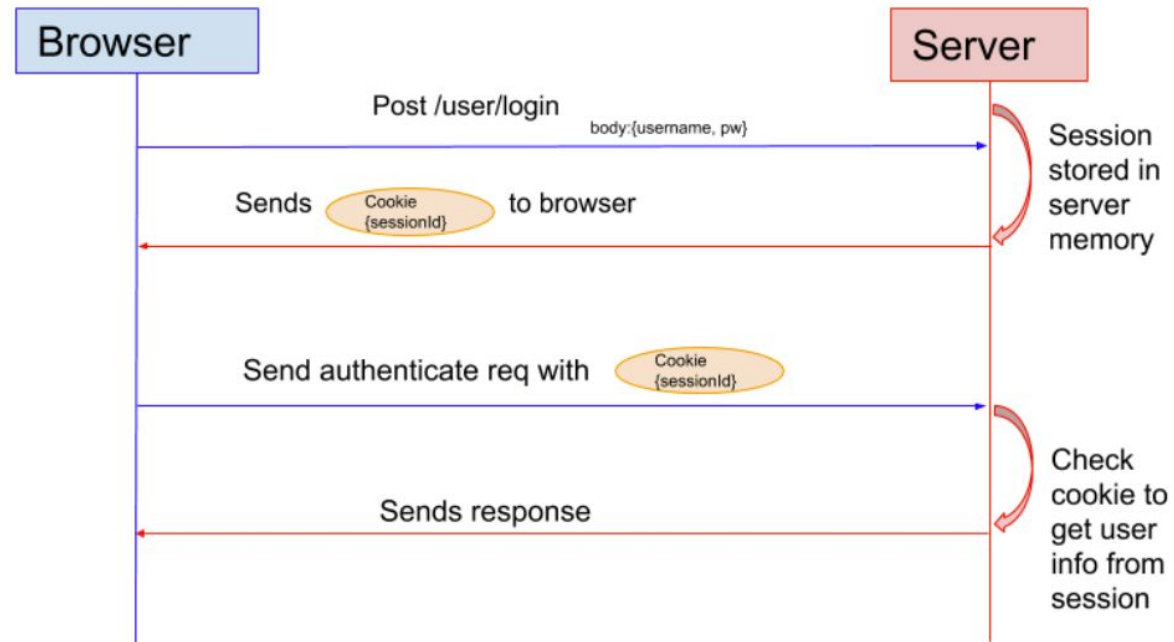
Идентификация по логину и паролю

- Идентификационные данные хранятся в общей БД.
- Логика идентификации находится в общей кодовой базе



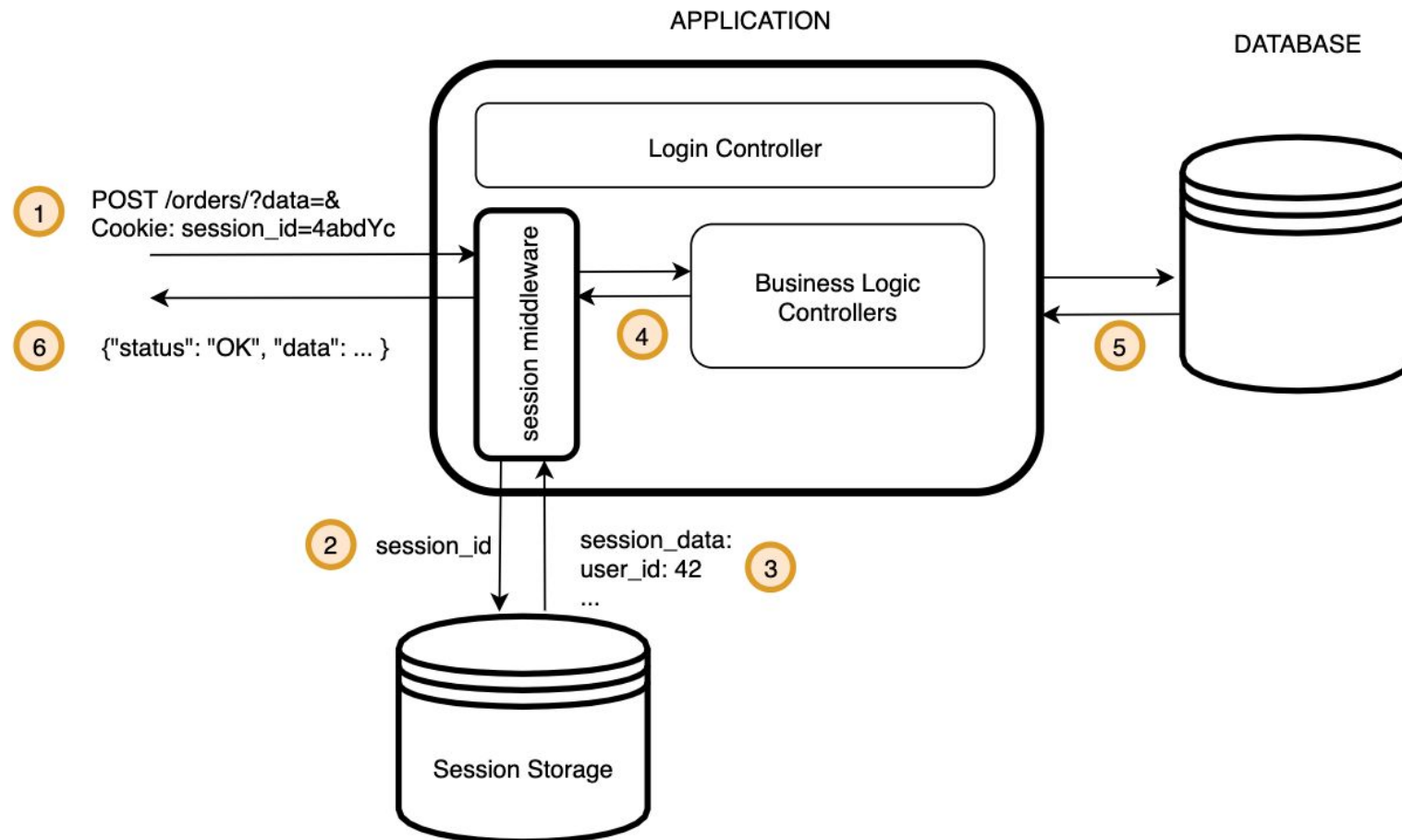
Сессия

- 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.







Сессия

- Логика проверки сессии находится в middleware, которая проверяет для всех авторизованных запросов аутентифицирован пользователь или нет
- Сессии хранятся либо в общей БД, либо в отдельном хранилище (kv хранилище)



Рекомендации

- 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

				
Password	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz
Salt	-	-	et52ed	ye5sf8
Hash	f4c31aa	f4c31aa	1vn49sa	z32i6t0

<https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>

Рекомендации к сессии на основе cookie

- Не передавать в куки sensitive информацию. Лучше всего в качестве session_id – случайный и длинный ключ
- Использовать **HttpOnly**
Куки с таким флагом недоступны из Javascript, что снижает XSS угрозы
- Использовать **Secure**
Куки с таким флагом доступны только по HTTPS, что снижает вероятность MITM атаки
- Использовать **SameSite** (пока не всеми поддерживается)
Куки с таким флагом не будут передаваться с third party сайтов, что снижает CSRF угрозы

<https://web.dev/samesite-cookies-explained/> - про SameSite куки в Chrome

02

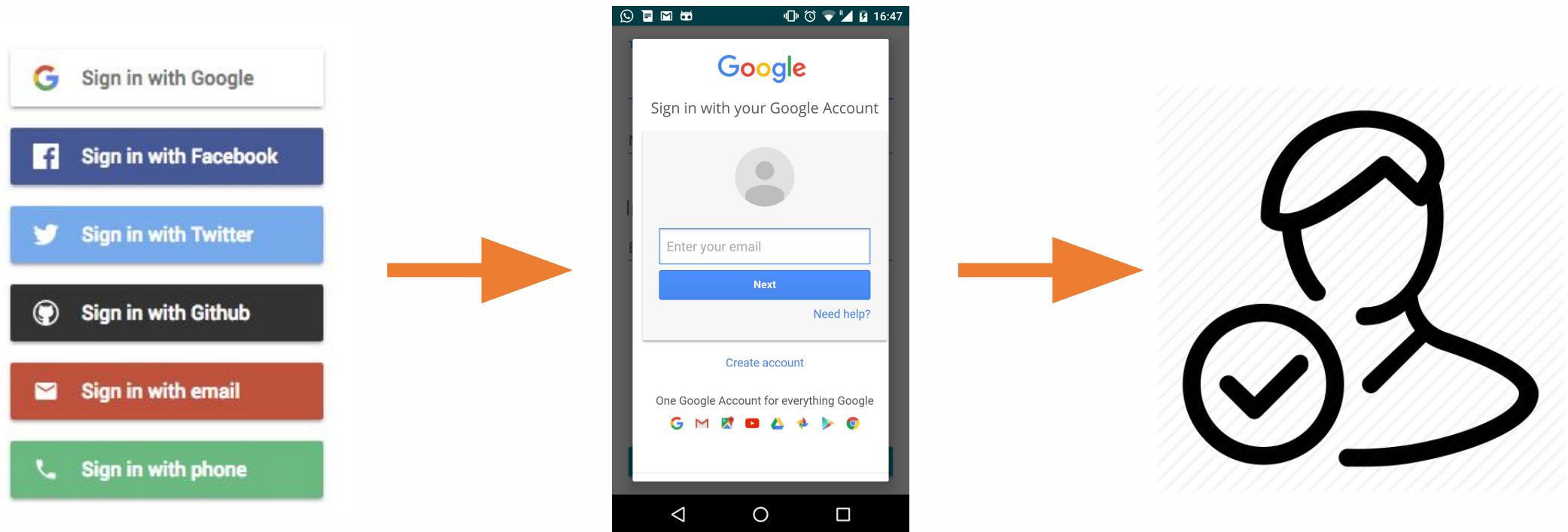
Внешний Identity Provider

Идентификация через внешнего провайдера

- Пользователю не удобно придумывать пароль и логин для каждого сервиса
- Сервисам не удобно делать идентификацию самим: 2-ух факторную авторизацию, думать о безопасности для логинов и паролей

Идентификация через внешнего провайдера

- Пользователь видит возможность залогиниться через Провайдера - Google, Facebook, Twitter и т.д.
- Пользователь нажимает на кнопку «Войти через»
- Пользователь авторизуется в Провайдере
- После авторизации пользователь в Провайдере перенаправляется в сервис, которые его авторизует.



Идентификация через внешнего провайдера

Существует несколько спецификаций и стандартов для идентификации через Внешнего провайдера:

- **OAuth2.0/OpenID Connect**
- SAML
- Самописные флоу

Самый популярный и дефакто стандарт – это OIDC & OAuth2.0

OAuth2.0 и OpenID Connect

OAuth 2.0 - это фреймворк для построения флоу, которые бы давали возможность не только идентифицировать пользователя внешним провайдером, но давать возможность совершать действия от лица этого пользователя с различным ресурсами.

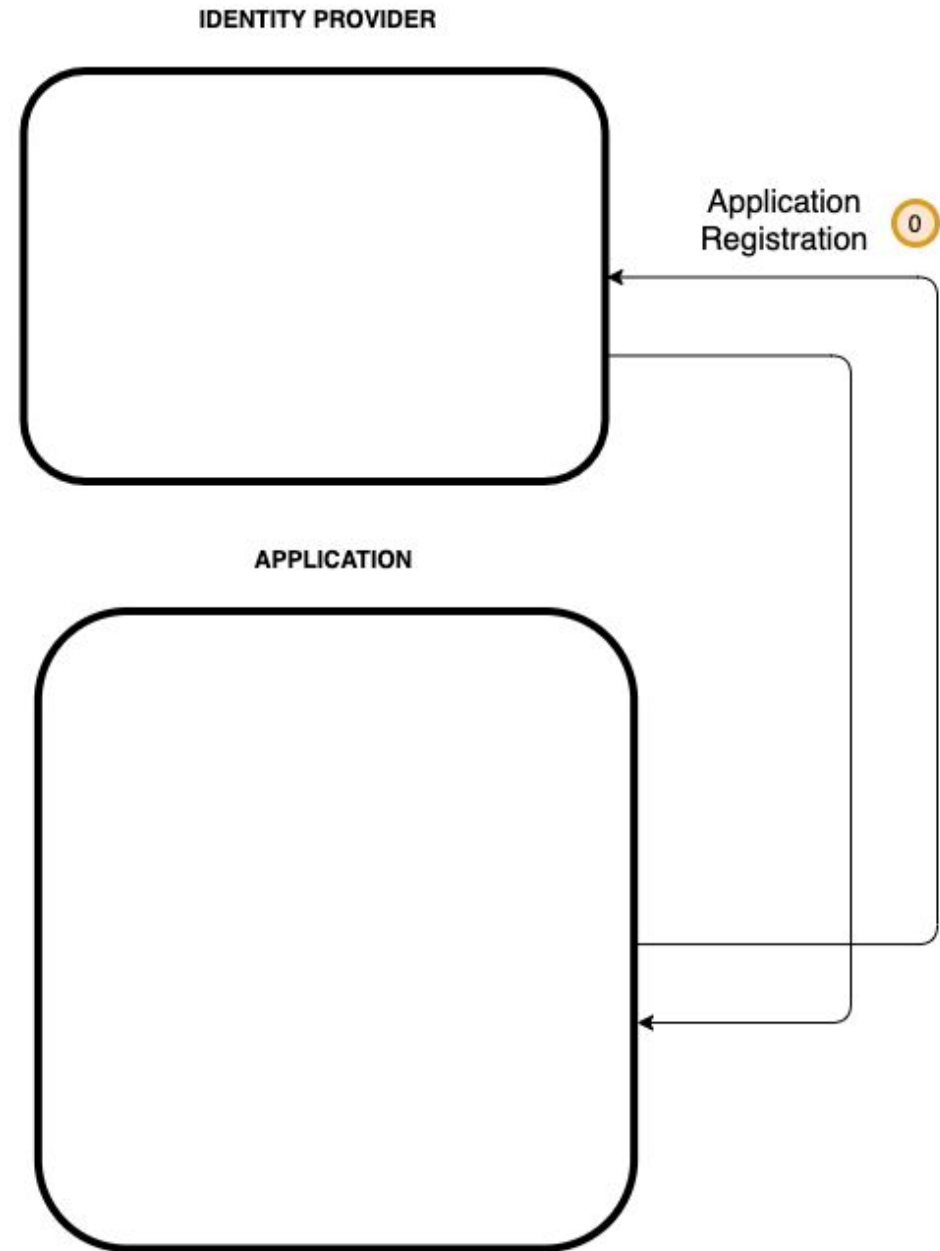
Основная проблема, что OAuth2.0 не стандартизирует способ получения данных о пользователе (Identity). Т.е. OAuth2.0 про аутентификацию и авторизацию.

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

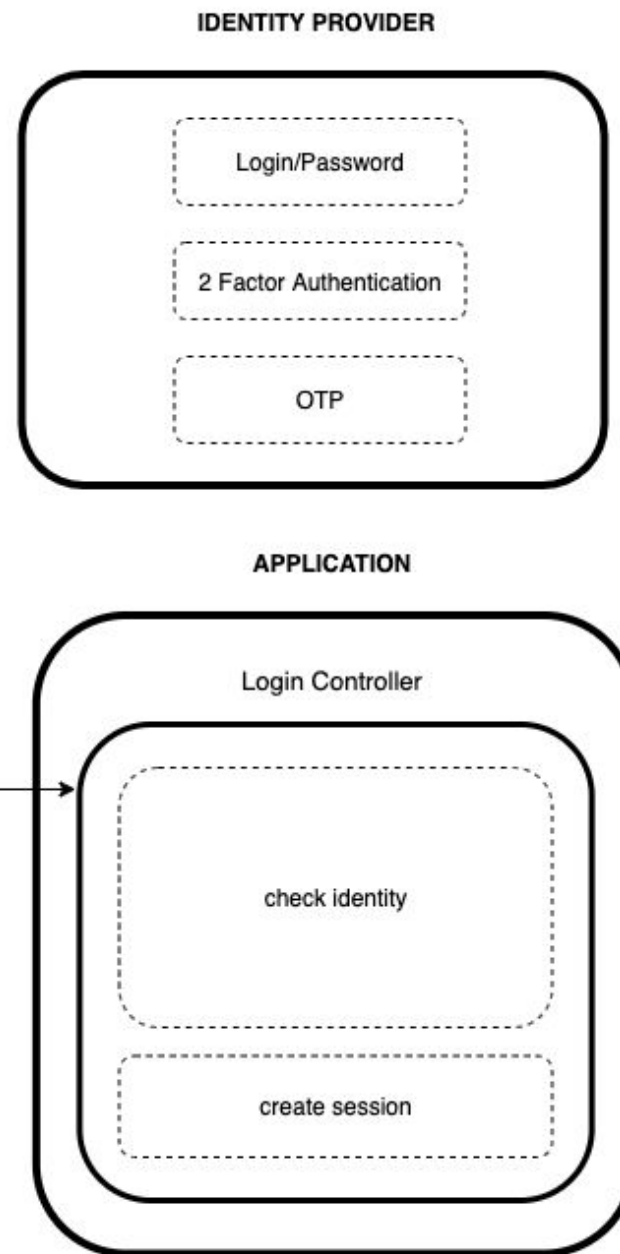


Пользователь делает запрос

К нам приходит пользователь и делает неавторизованный запрос.



1 /auth/?

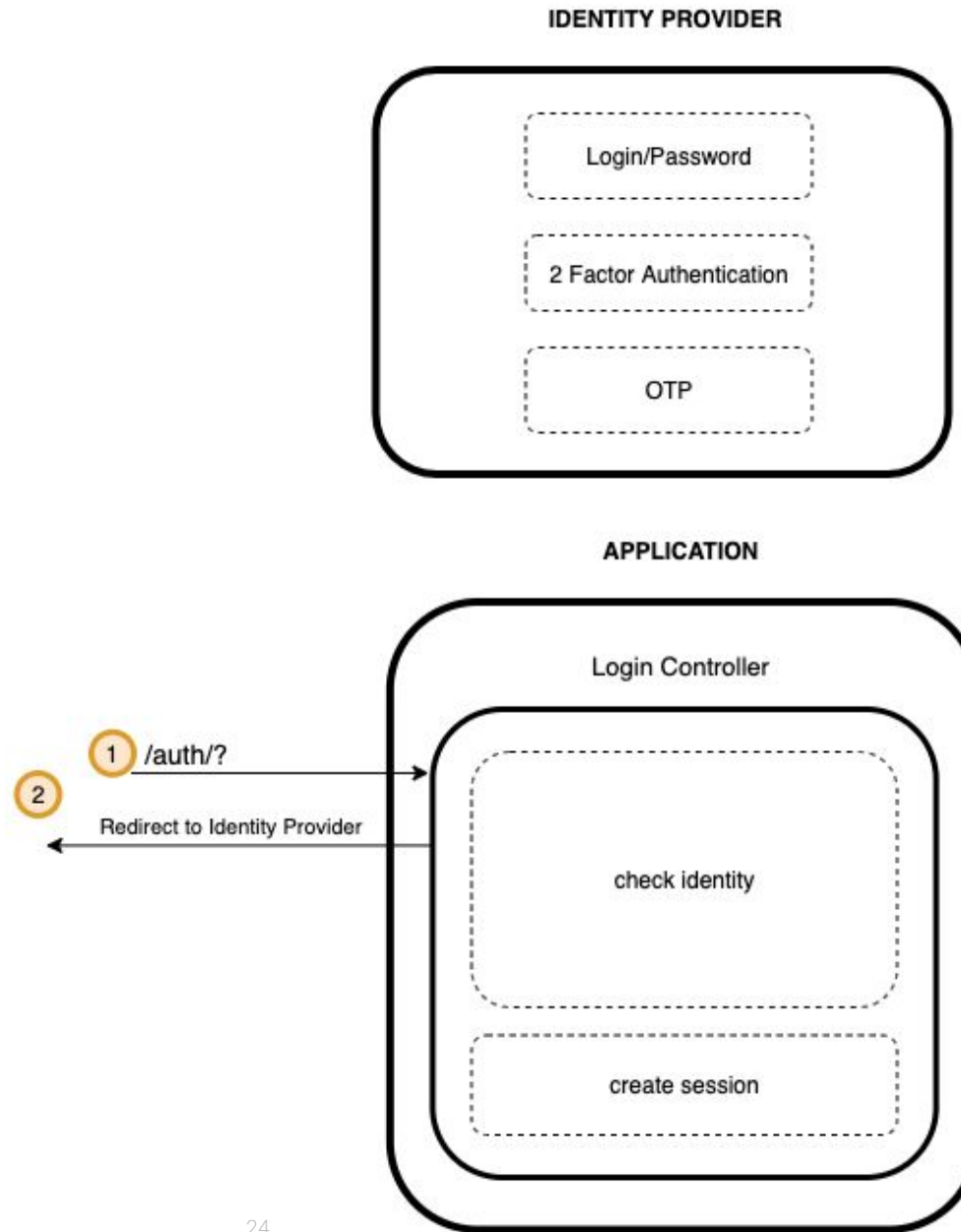


Редиректим пользователя к Identity provider-y

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** – урл, на котором приложение будет ждать ответ от провайдера
- 3) **scope** – уровень доступа, который мы хотим получить. Для идентификации **openid** (можно дополнительно profile/email)

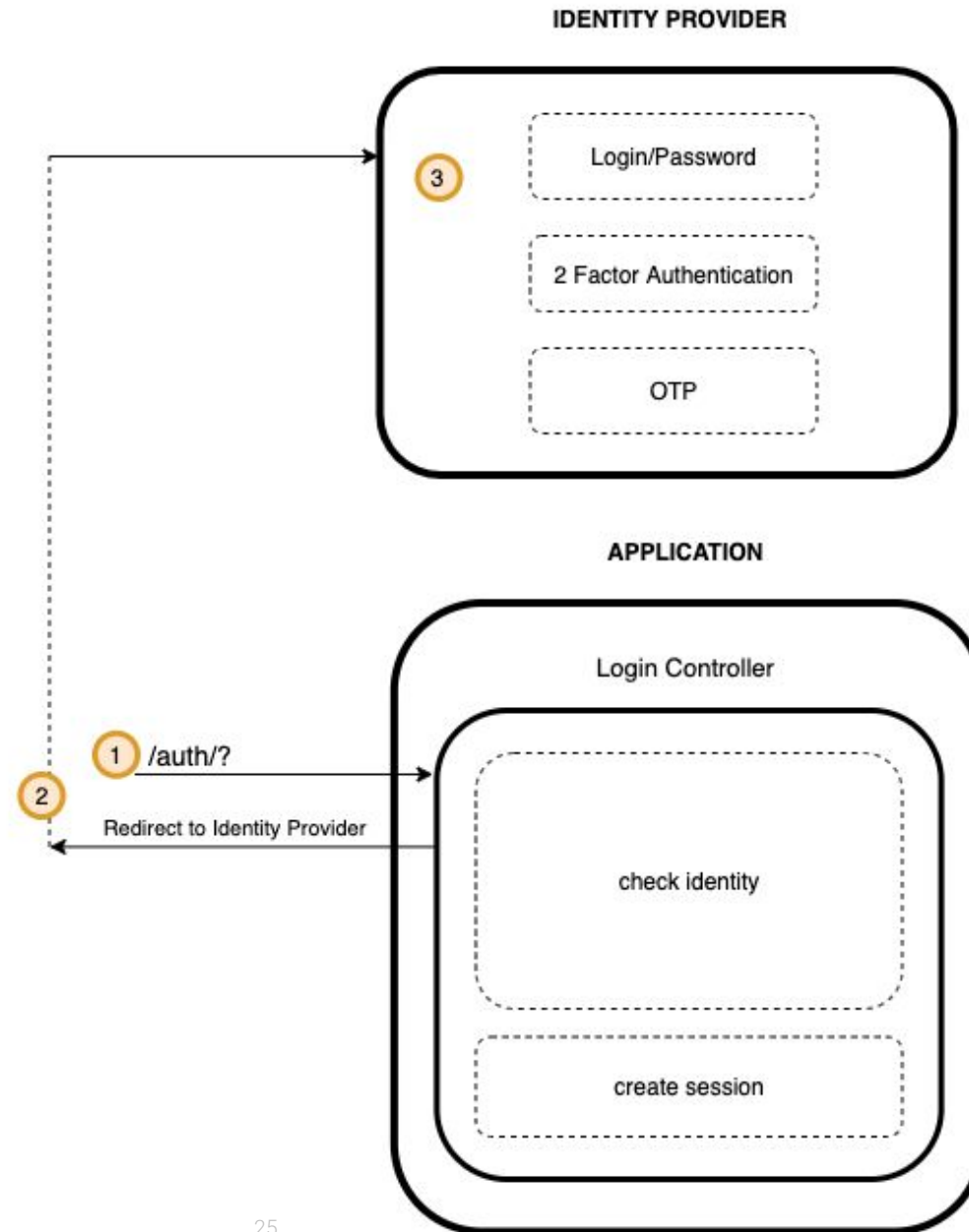


Identity провайдер делает свою работу

Identity провайдер делает свою работу – он идентифицирует пользователя. Способ, которым он это делает – остается на его усмотрение.

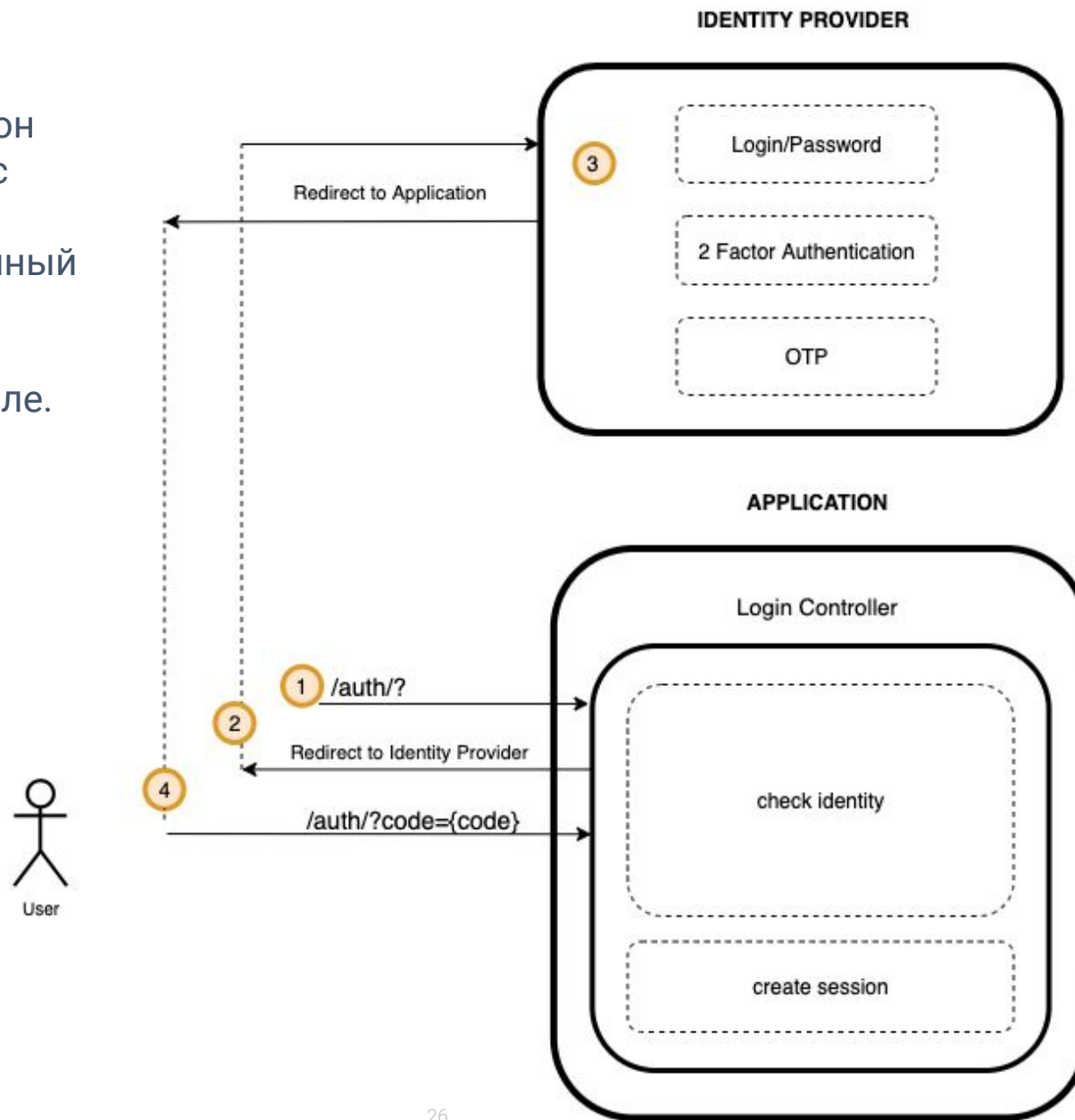
Может быть провайдер увидит, что у пользователя сейчас активна сессия, и пользователь даже не заметит, как он побывал на сайте Провайдера.

Или наоборот, IdP решит, что надо не только спросить логин и пароль, но и еще код из смс.



Identity provider редиректит 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.

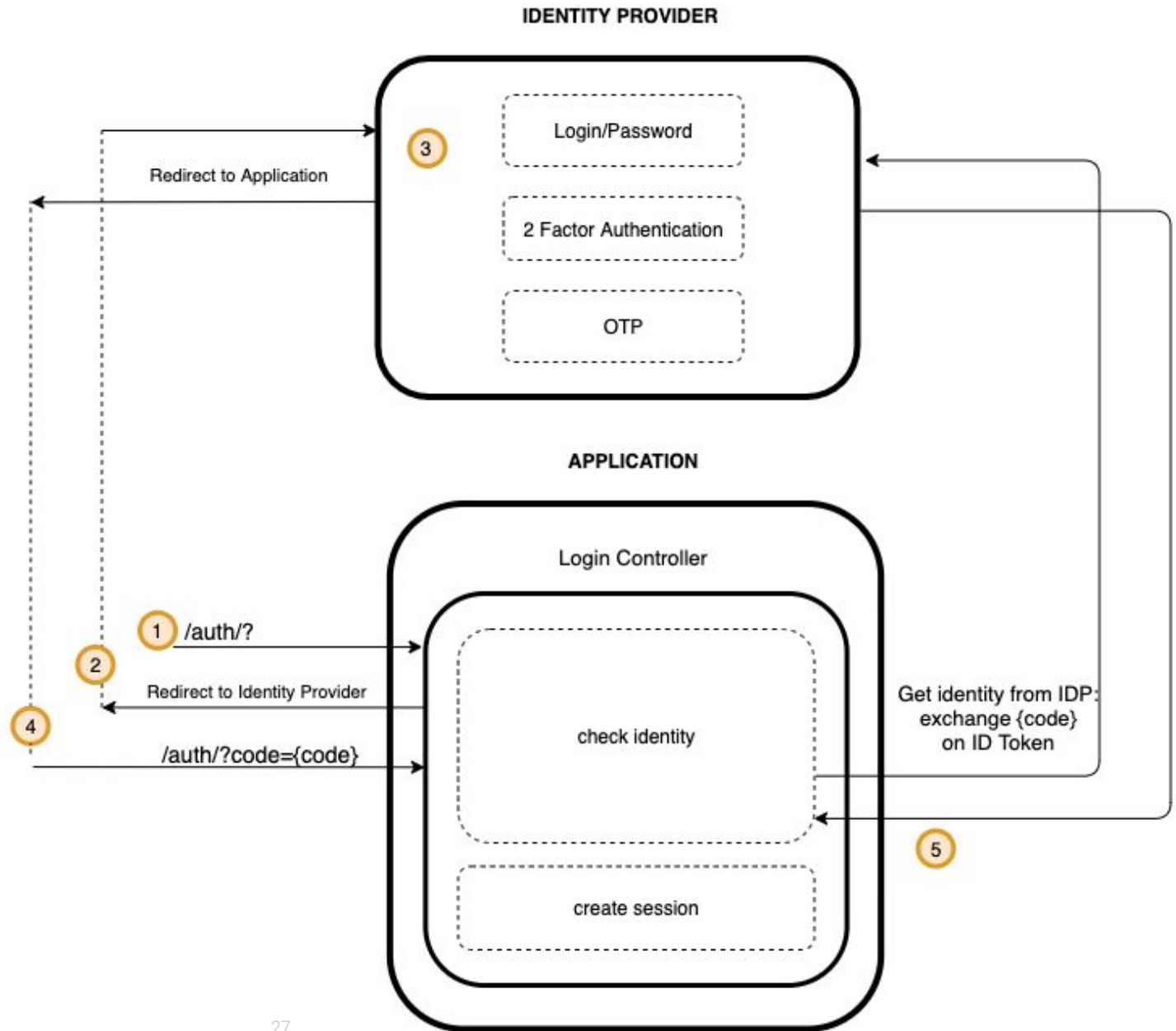


Узнаем у провайдера identity пользователя

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.

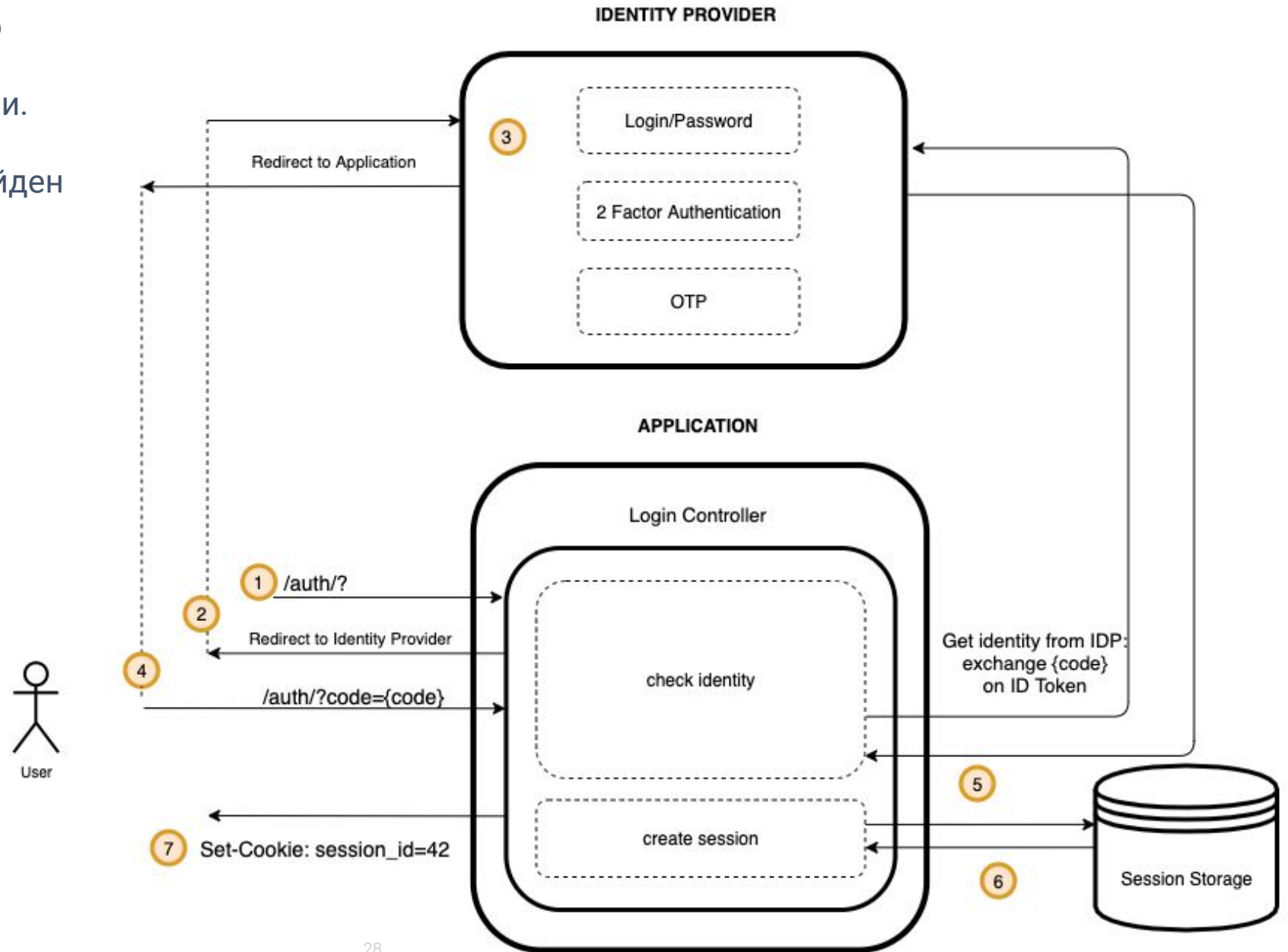
- Обмениваем одноразовый авторизационный токен (квиток) на access_token, refresh_token, id_token
- Access_token используется для того, чтобы делать запросы от лица пользователя
- ID_token это в специальном виде закодированная и подписанная информация о пользователе.



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, соответствующий ему, создается сессия.



Идентификация через внешнего провайдера

- Часто удобна, и повышает конверсию для сервисов
- Иногда невозможна, например, в случае корпоративных сервисов
- OIDC/OAuth2 все делают “немного” по-своему, что приводит к отсутствию универсальных библиотек. Правда сейчас это решается с помощью openid-configuration

Клиентские библиотеки

Есть официальный набор библиотек для работы с OIDC

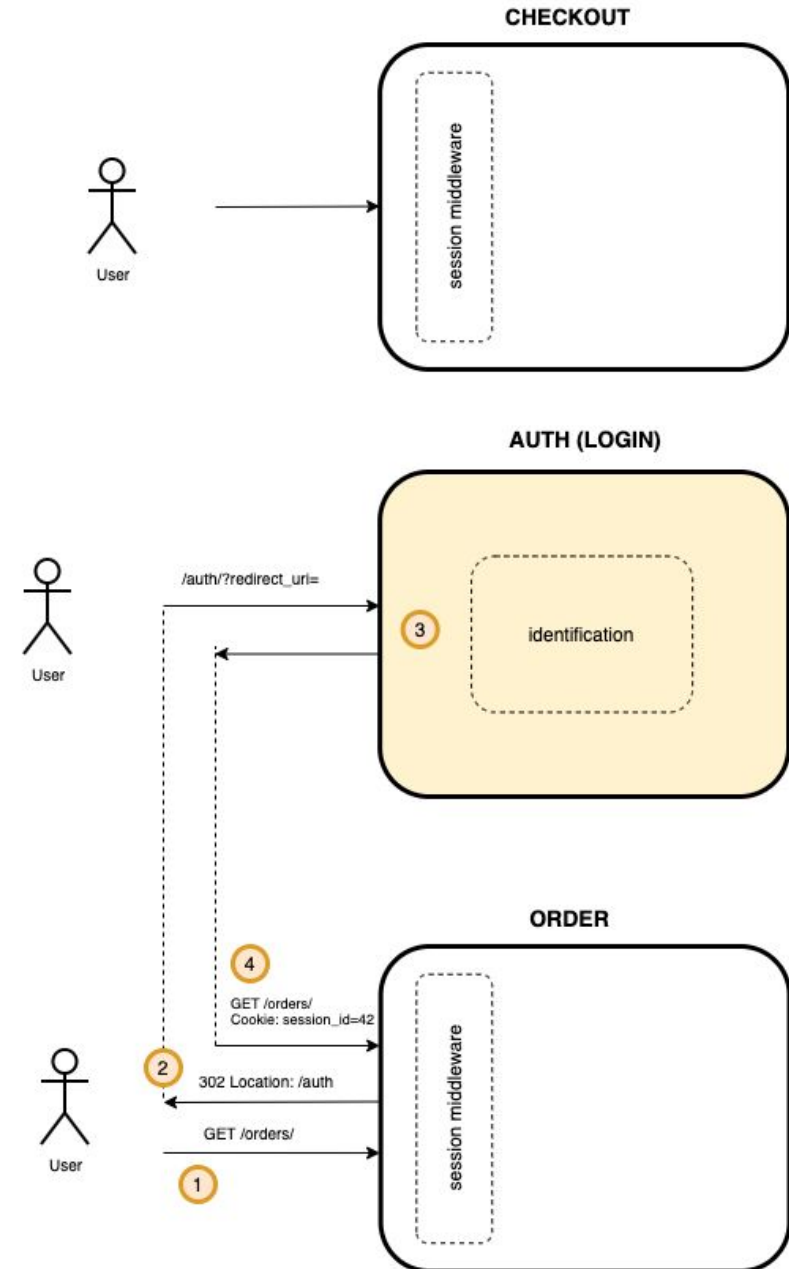
<https://openid.net/developers/certified/>

03

Паттерны аутентификации в микросервисах

Единый сервис идентификации

- В микросервисной архитектуре, чтобы в каждом сервисе не дублировать код по логину пользователя, его обычно выделяют в отдельный сервис.
- Когда сервис понял, что пользователь неавторизован, делается редирект пользователя на сервис «аутентификации».
- В этом сервисе пользователь идентифицируется, например, с помощью внешних IdP или с помощью логина-пароля
- Создается сессия.
- Сервис его возвращает на исходную страничку с помощью редиректа.

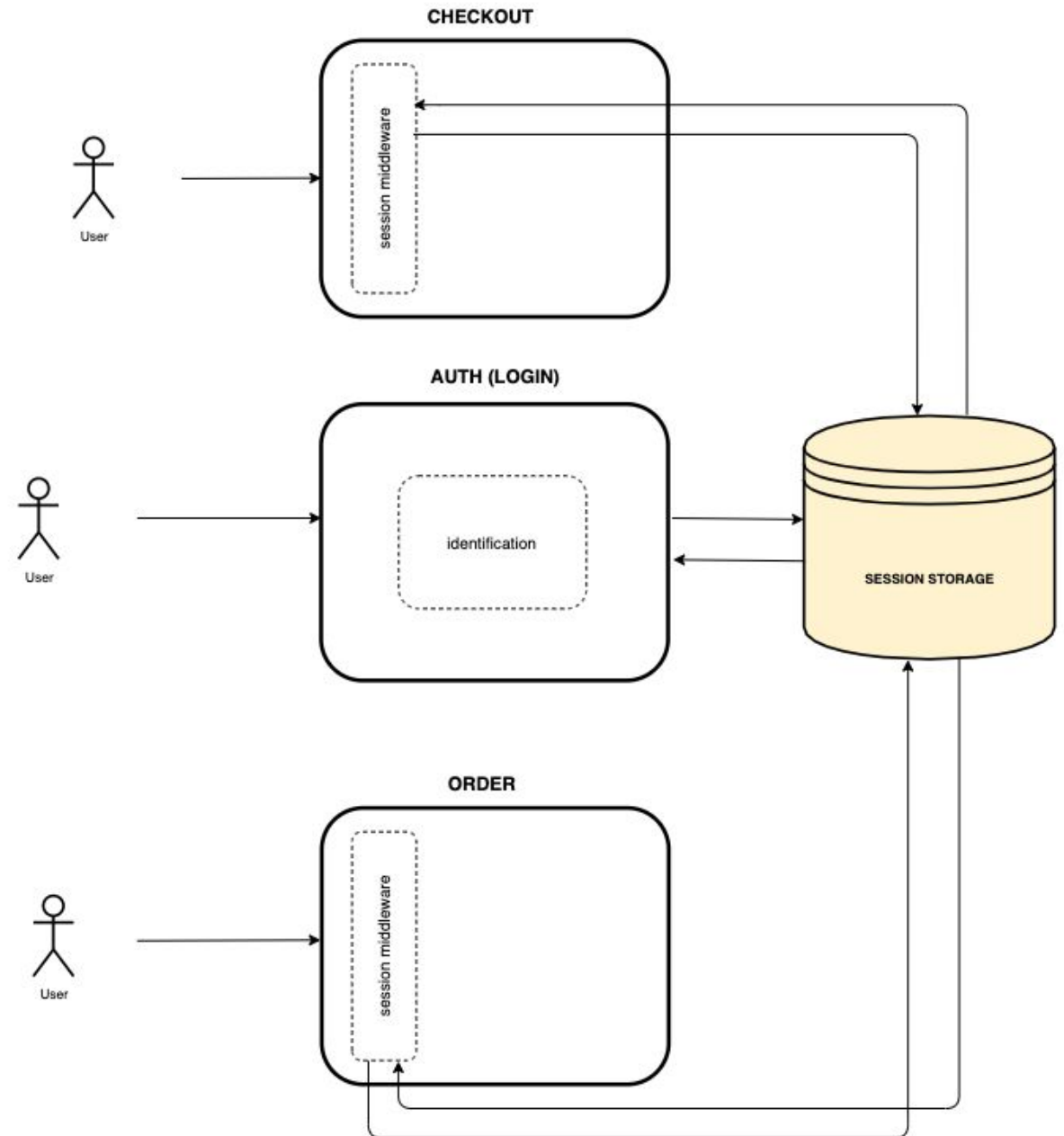


Единое хранилище для сессий

- Чтобы сессии были едиными, их нужно как-то централизовать, например, можно хранить данные о сессиях в отдельном хранилище.
- Каждый сервис обращается к этому хранилищу, когда хочет проверить валидность сессии, записать данные, создать.

Минусы подхода:

- Сменить хранилище в таком случае будет тяжело
- Нужно очень аккуратно на разных языках программирования и фреймворках реализовать доступ к этому хранилищу. Ошибка в одном сервисе приведет к порче «глобальных» данных

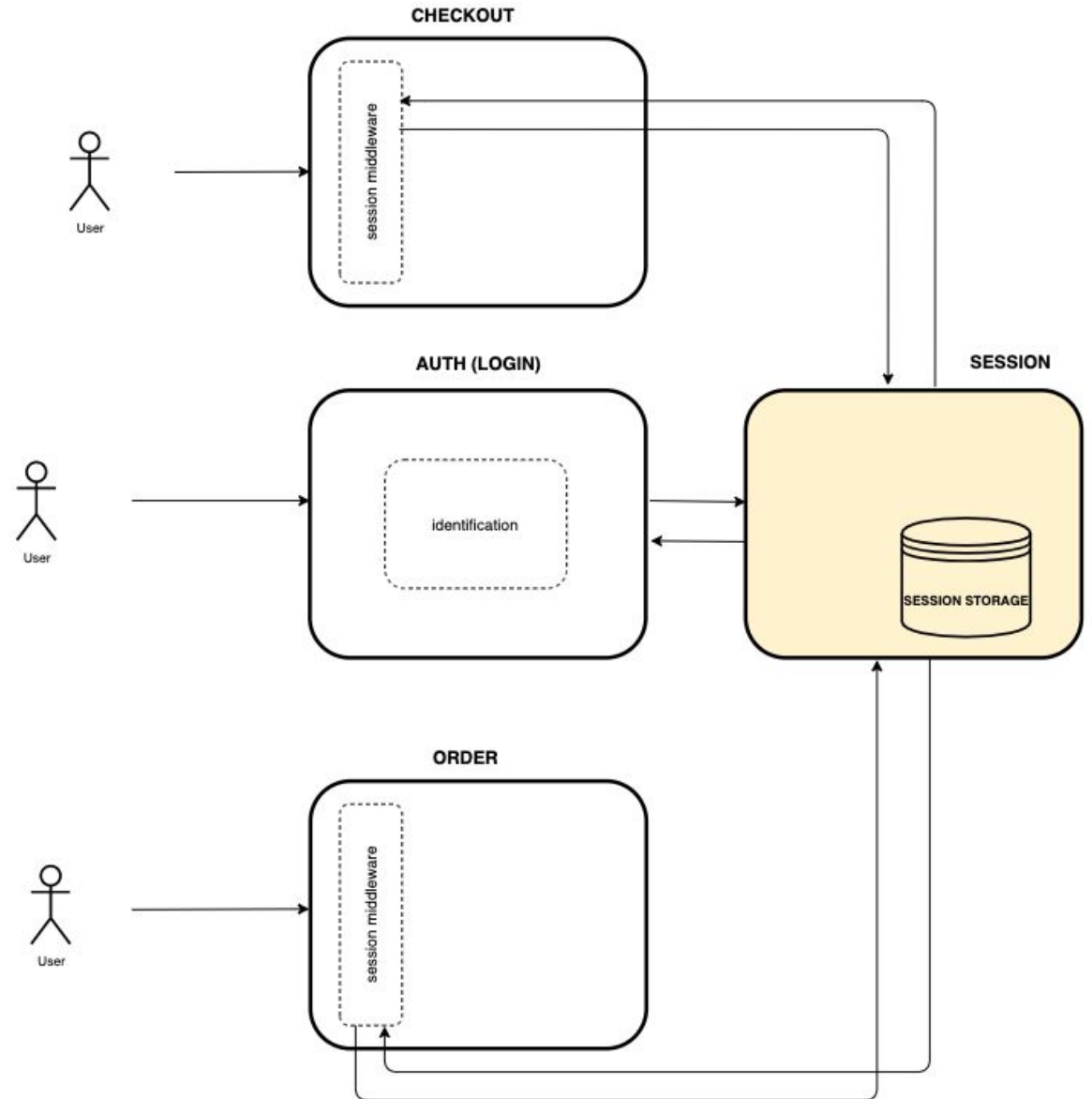


Единый сервис для сессий

Чтобы иметь возможность менять реализацию хранения сессий, иногда используют отдельный сервис управления сессиями.

Минусы подхода:

- Сервис является очень нагруженным
- Нужно будет в каждом сервисе делать отдельную библиотеку для доступа к этому сервису



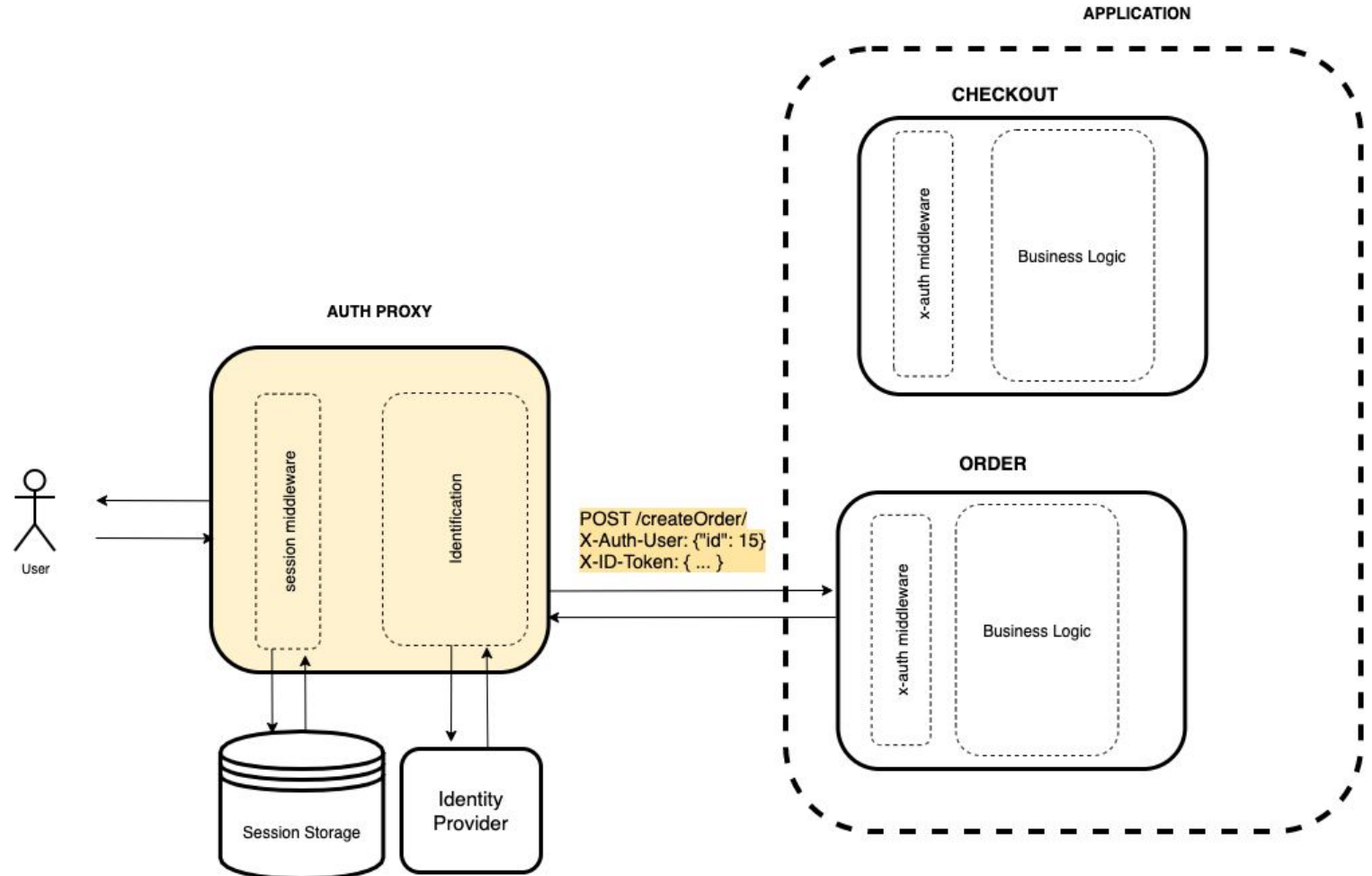
Auth прокси

Помимо использования библиотек, можно использовать вариант с прокси, который занимается аутентификацией.

В таком варианте все клиентские запросы проходят через Auth Proxy, который идентифицирует клиента, создает и проверяет сессии.

В рамках реализации AuthProxy может использовать внешнего Identity Provider-а, может и не использовать.

Сервисам проксируется запрос с уже обогащенной информацией о пользователе.

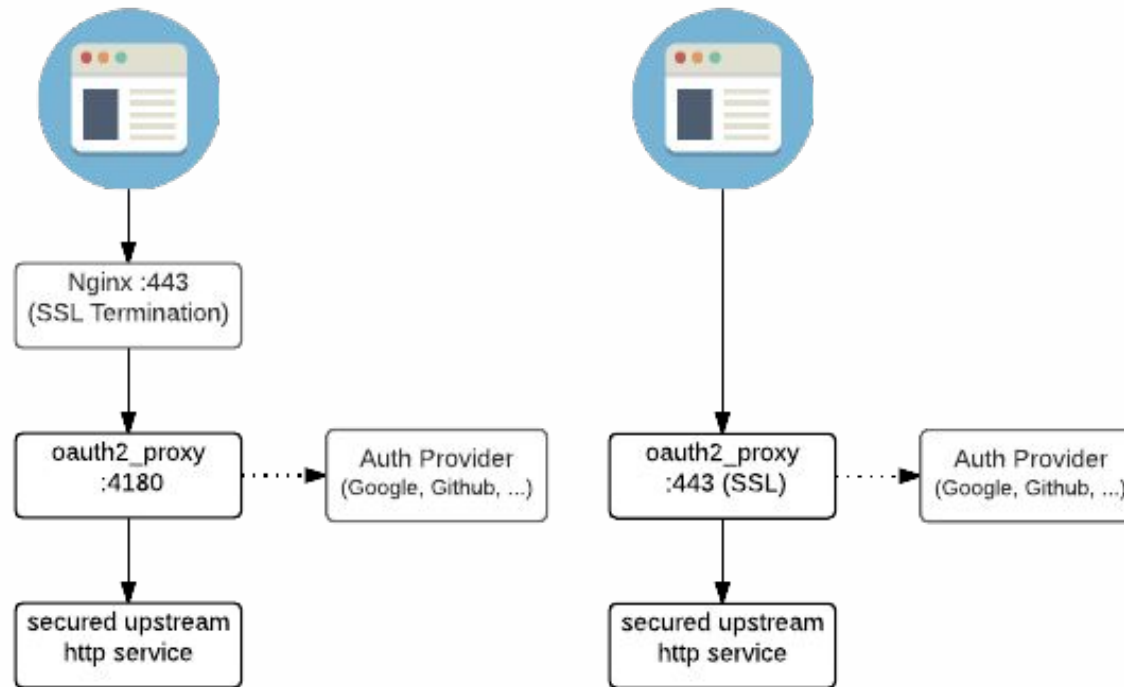


Oauth2-proxy

Классический вариант – oauth2-proxy (<https://oauth2-proxy.github.io/oauth2-proxy/>)

На основе этого прокси есть еще такие решения:

- <https://github.com/pomerium/pomerium> (<https://www.pomerium.io/>)
- <https://github.com/openshift/oauth-proxy>

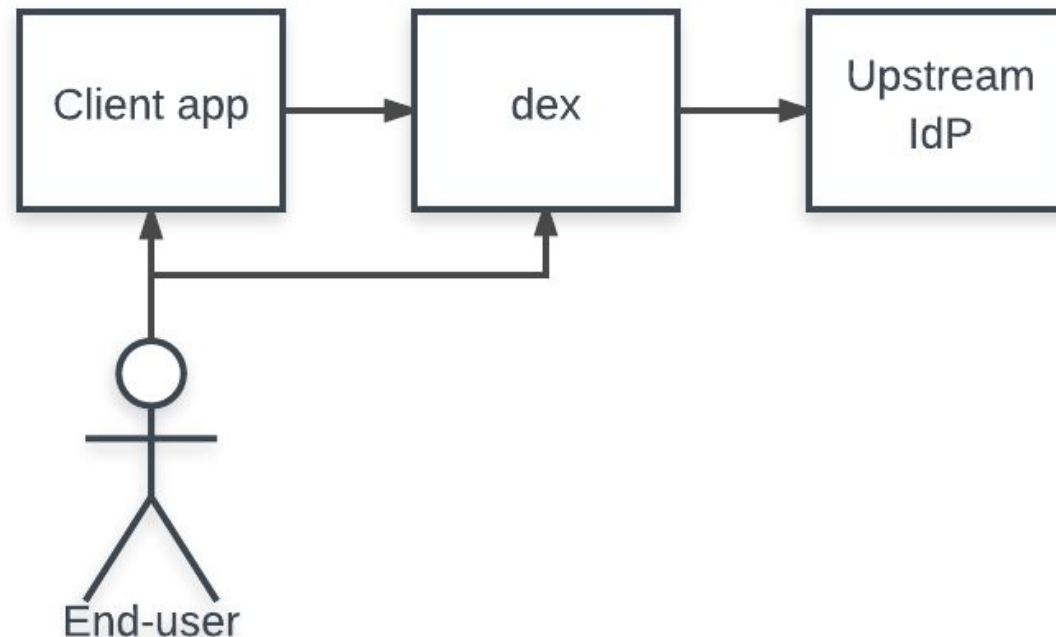


OIDC коннектор

<https://github.com/dexidp/dex>

DEX – пример реализации Identity провайдера, который позволяет через себя подключить других Identity Provider-ов, которые не обязательно являются OIDC, например, LDAP, SAML и т.д.

Приложению достаточно клиентской библиотеки к dex, а dex сам уже общается с Identity Provider-ами



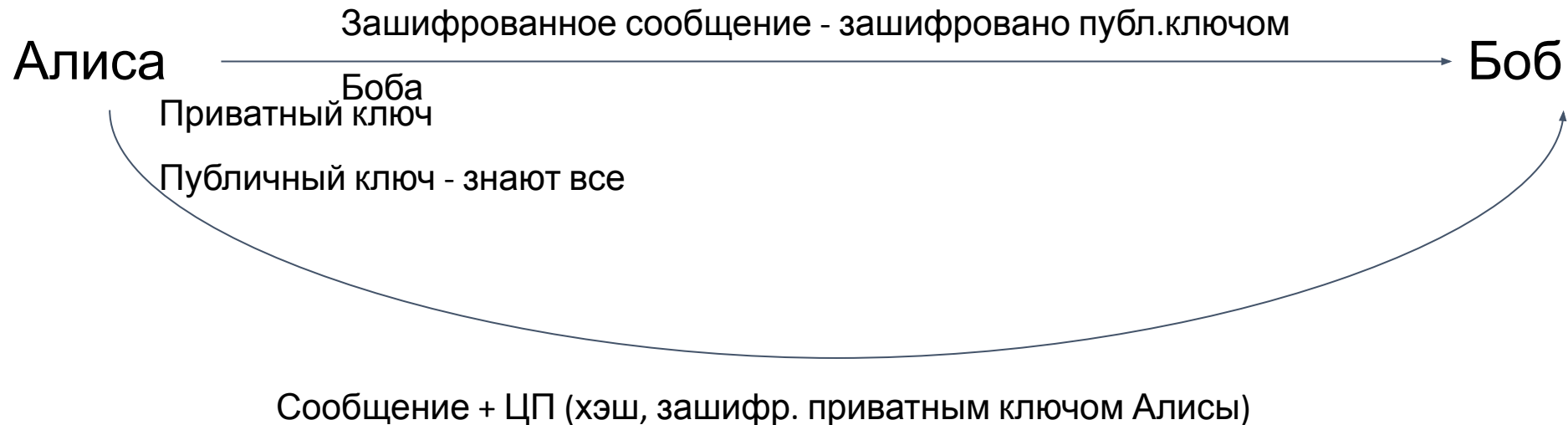
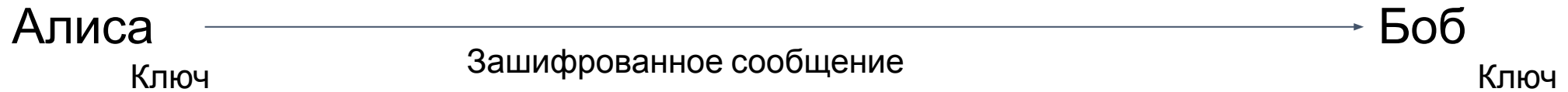
Особенности cookie-based централизованных подходов

- Сессионные куки доступны только на определенных доменах. Если микросервисы находятся на разных доменах, то xhr-запросы будут блокироваться
- Единое хранилище для сессий обычно становится критичным, как по производительности, так и по отказоустойчивости сервисом
- Поддержка сессий практически полностью реализуется на бекенде, для фронтенда практически ничего делать не надо

04

Token-based сессии. JWT

Несимметричное шифрование и цифровая подпись



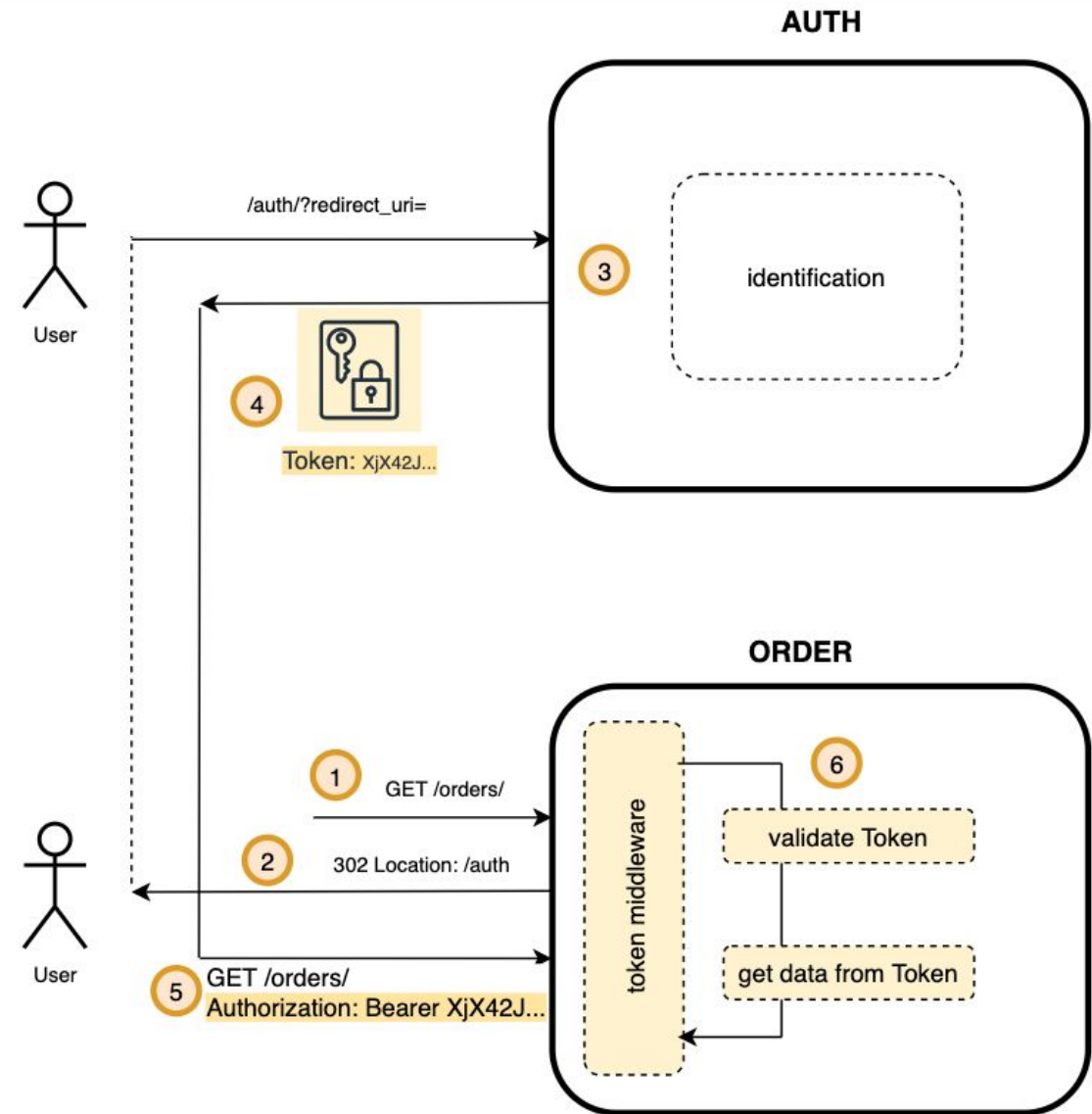
Token-based сессии

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

Чтобы каждый не придумывал свой формат токенов, возник стандарт JWT.

JWT-токен – это просто закодированный в base64url с данными + заголовок с метаданной и подпись.

JWT = B64(Header).B64(Payload).SIGN

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MzkwMjQ.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)</pre> <input type="checkbox"/> secret base64 encoded	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

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

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)</pre> <input type="checkbox"/> secret base64 encoded	Signature is created by signing encoded header and payload with signature using header algorithm

JWT библиотеки

В подавляющем большинстве случаев самостоятельно писать библиотеки для кодирования и декодирования JWT токенов не придется (и не стоит)

Здесь - <https://jwt.io/#libraries-io> – можно посмотреть набор официальных библиотек.



Работа с JWT на клиентской стороне

Когда пользователь логинится, ему передается JWT Token, который сохраняется на клиенте

- JWT токены позволяют реализовать механизм SSO – Single Sign On – если пользователь залогинился в одном сервисе, то в остальных логиниться не надо.
- В отличие от архитектур, где сессионные данные хранятся в едином хранилище, JWT не позволяет достаточно просто реализовать Single Sign Off. Если пользователь разлогинился в одном сервисе, то в другом он все-равно останется залогинен.



<https://www.youtube.com/watch?v=DPrhem174Ws>

<https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/>

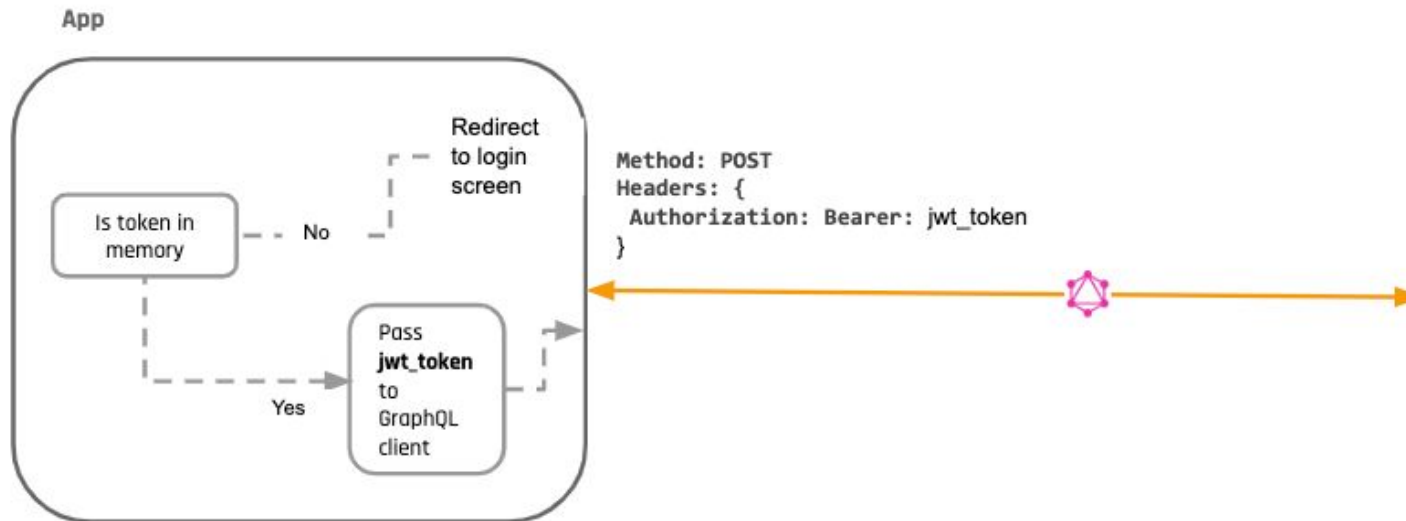
Где хранить JWT токен

- JWT токен не стоит хранить в LocalStorage, потому что возможна XSS атака.
- Если JWT токен хранить в Cookie, стоит помнить про CSRF атаки и использовать SameSite или CSRF-токен
- Хранение JWT токена в памяти сопряжено с неудобствами шаринга этого токена между разными вкладками, и с возможностью XSS атак.
- Поскольку payload в JWT просто закодирован, а не зашифрован, хранить sensitive информацию в них нельзя

Работа с JWT на клиентской стороне

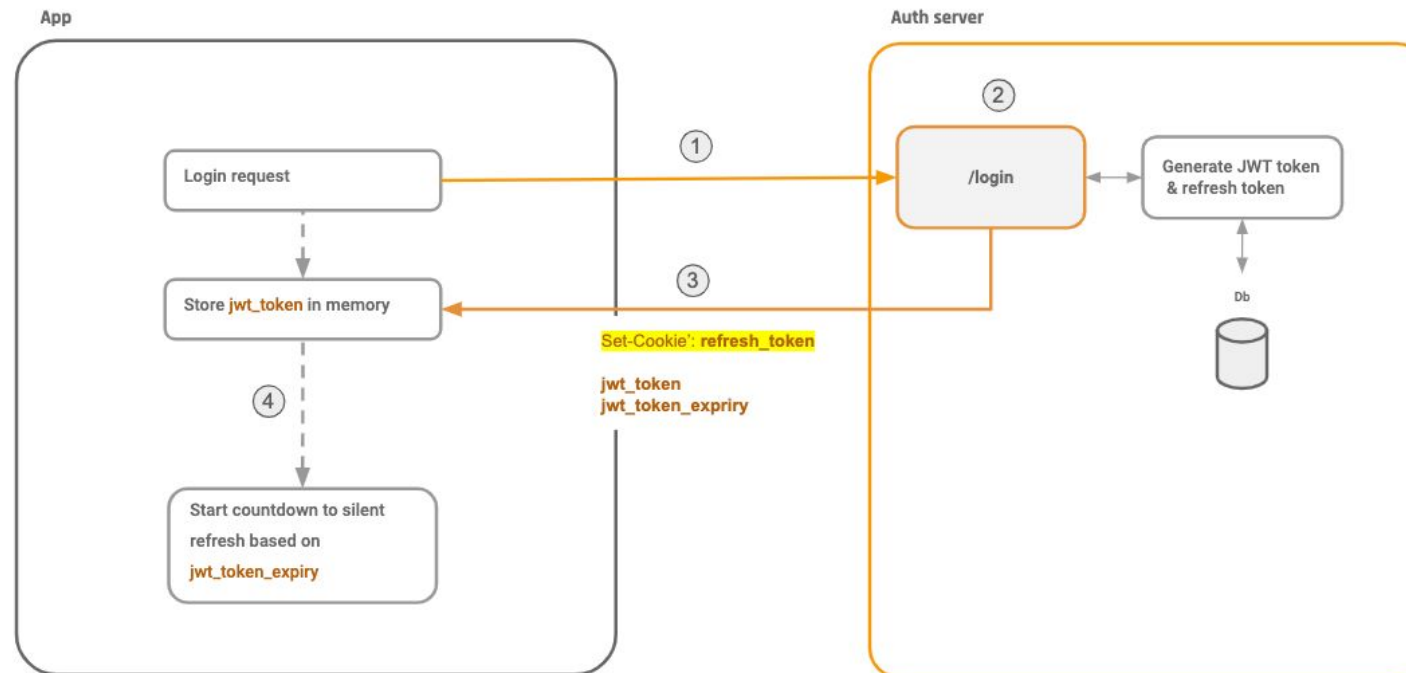
С каждым запросом на сервер отправляется заголовок
Authorization: Bearer jwt_token

Если на клиенте JWT нет этого заголовка, пользователя редиректят на логин.



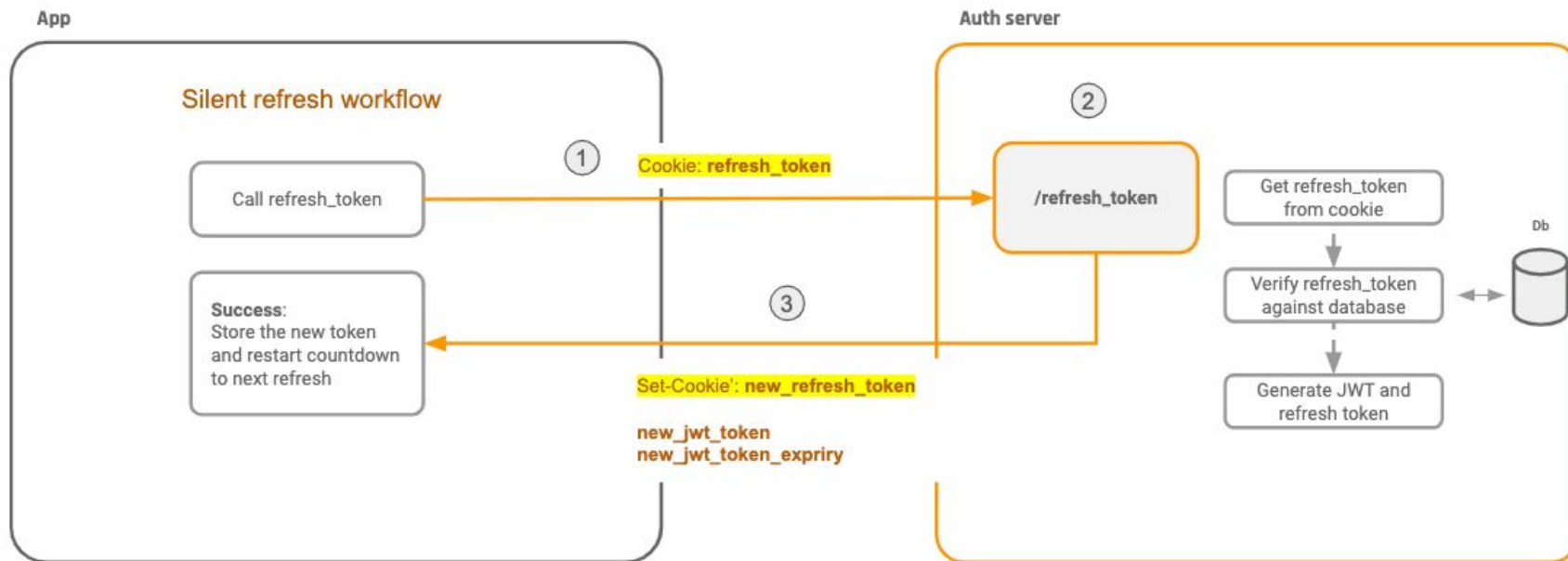
Работа с JWT на клиентской стороне

- Скомпрометированные JWT-токены нельзя инвалидировать без наличия централизованного хранилища. Поэтому время жизни JWT токена должно быть коротким (например, 15 минут).
- Чтобы пользователя не разлогинивало каждые 15 минут, необходимо реализовать механизм скрытого пополнения JWT-токена. Для этого нужен еще один токен – RefreshToken
- Хранить этот токен можно в HttpOnly, Secure Cookie, которая устанавливается при логине



Работа с JWT на клиентской стороне

- Если запрос на сервер завершился 401, то необходимо сходить перевыписать ключ с помощью RefreshToken-а, и переотправить запрос.



Основные особенности token based сессий

- Может упростить backend разработку, если критично отсутствие единого хранилища для сессий.
- Не обеспечивает простого решения для single sign off, поэтому если его наличие критично, может не подойти.
- Существенно усложняет браузерную клиентскую разработку. Без специальных сервисов или библиотек сделать безопасно token-based сессии тяжело.

<https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/>

<https://scotch.io/bar-talk/why-jwts-suck-as-session-tokens>

<https://medium.com/@sherryhsu/session-vs-token-based-authentication-11a6c5ac45e4>

05

Service 2 Service аутентификация

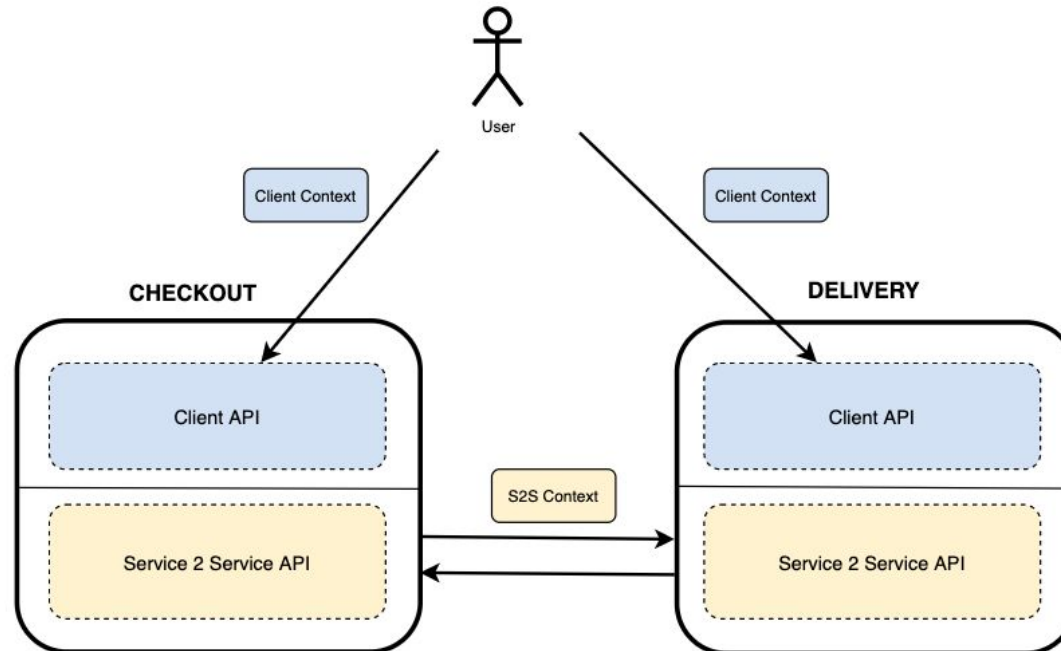
Межсервисное взаимодействие

Зачастую у сервисов есть 2 типа взаимодействий:

- Обработка клиентских запросов
- Обработка сервисных запросов от доверенных сервисов внутри приложения

Крайне желательно не смешивать эти два взаимодействия.

- Не стоит при запросе к другому сервису передавать клиентский токен или симулировать клиентский запрос (с помощью подделки кук и т.д).
- Сделать отдельный префикс API для s2s endpoint-ов, например, srv/



Межсервисное взаимодействие

Даже если сервисы находятся в одной доверенной сети, все-равно имеет смысл ограничивать или как минимум отслеживать кто в какой сервис ходит.

Это бывает полезно в ситуациях

- 1) когда нужно переводить API на новую версию, чтобы понимать, кто еще пользуется старой
- 2) Когда виден всплеск по RPS, но не понятно, какой из сервисов его вызвал

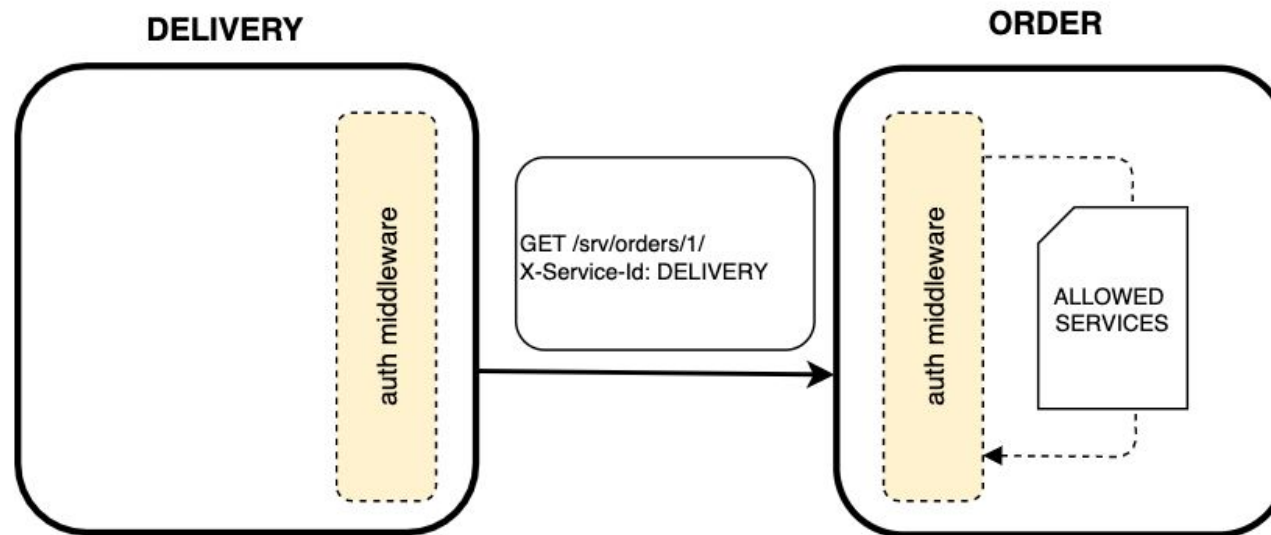
Контекст запроса и Identity сервиса, который вызывает API лучше передавать.

<https://hackernoon.com/service-to-service-authentication-for-microservice-apis-ccf4ab8073e6>

Аутентификация по service id

Самым простым вариантом является идентификация сервиса по Service-id. Сервис при обращении в другой в заголовках в метаданных пересылает свой идентификатор. Сервис, который принимает запросы у себя в конфигурации (или в каком-то другом хранилище) проверяет это Service-id.

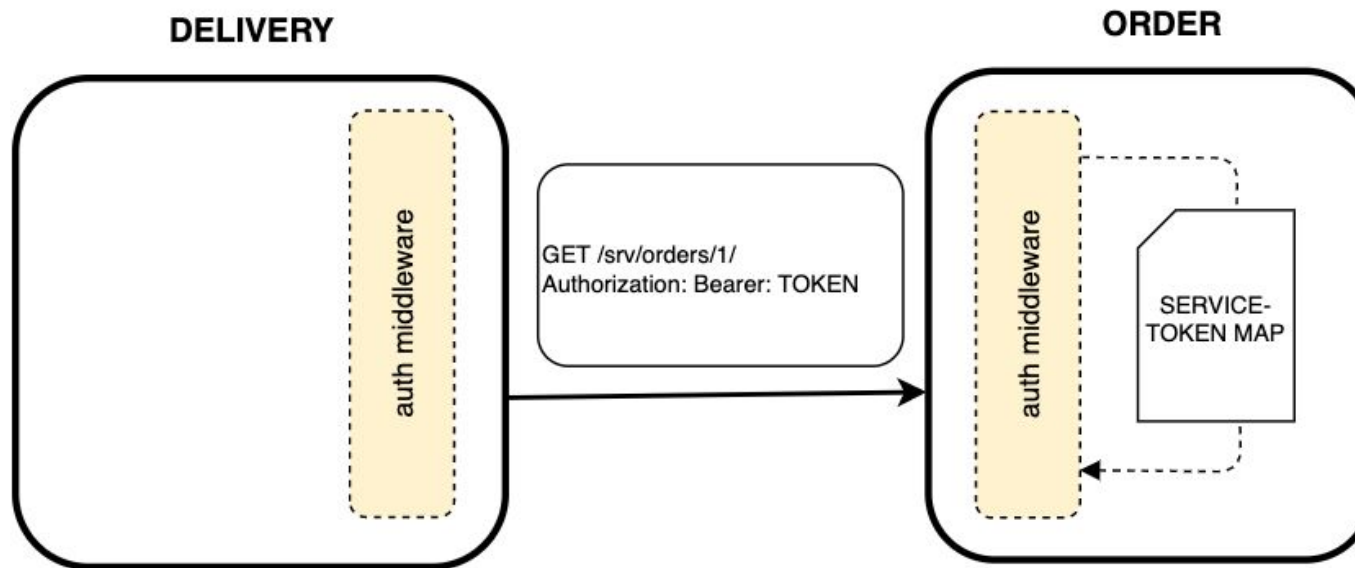
Очевидный недостаток, что случайно или умышленно service-id может быть подделан и не соответствовать действительности.



Аутентификация по токenu

Чуть посложнее вариантом является идентификация сервиса по Токену, который нельзя легко подделать. Токен может быть как просто случайным, так и быть в формате JWT.

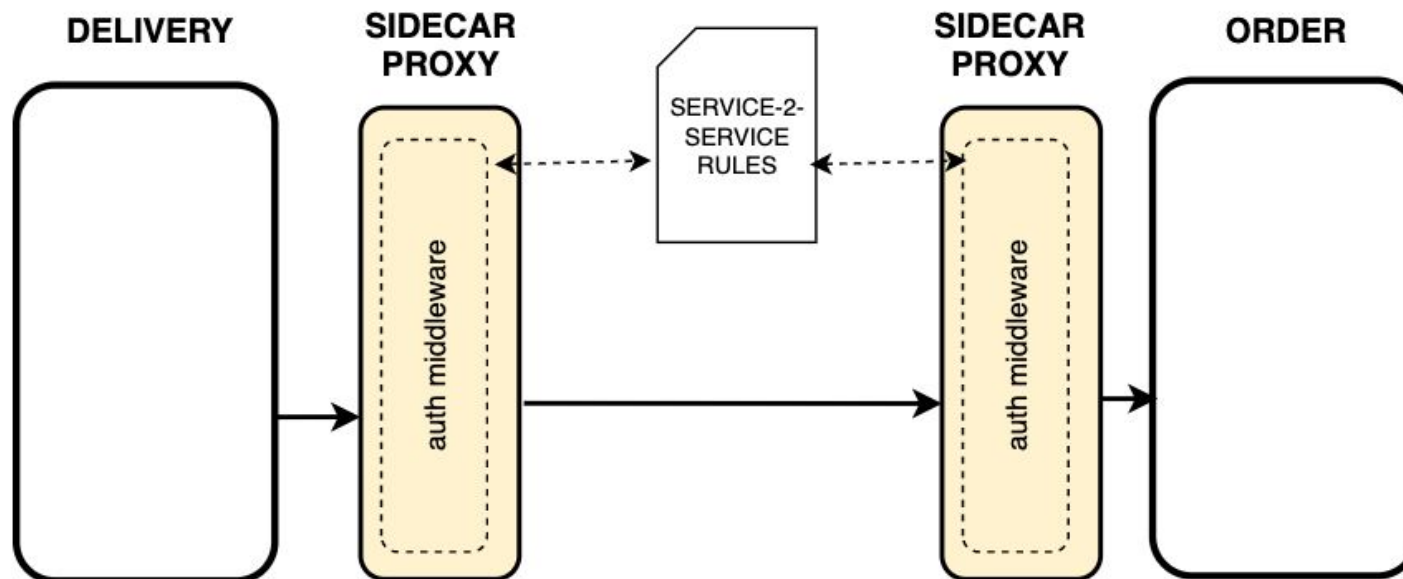
Хранение и управление токенами можно вынести в отдельный сервис. Но зачастую такая операция не приносит существенных преимуществ, если сервисов не очень много (>5000).



Аутентификация в рамках сервис-меша

При наличии сервис-меша запросы идут не напрямую из сервиса в сервис, а через прослойку проксей сервис-меша.

За счет него можно сделать централизованное управление настройками доступа сервисов друг к другу, и при этом инструментировать код конечных сервисов не придется. Что является преимуществом по сравнению с предыдущими вариантами



06

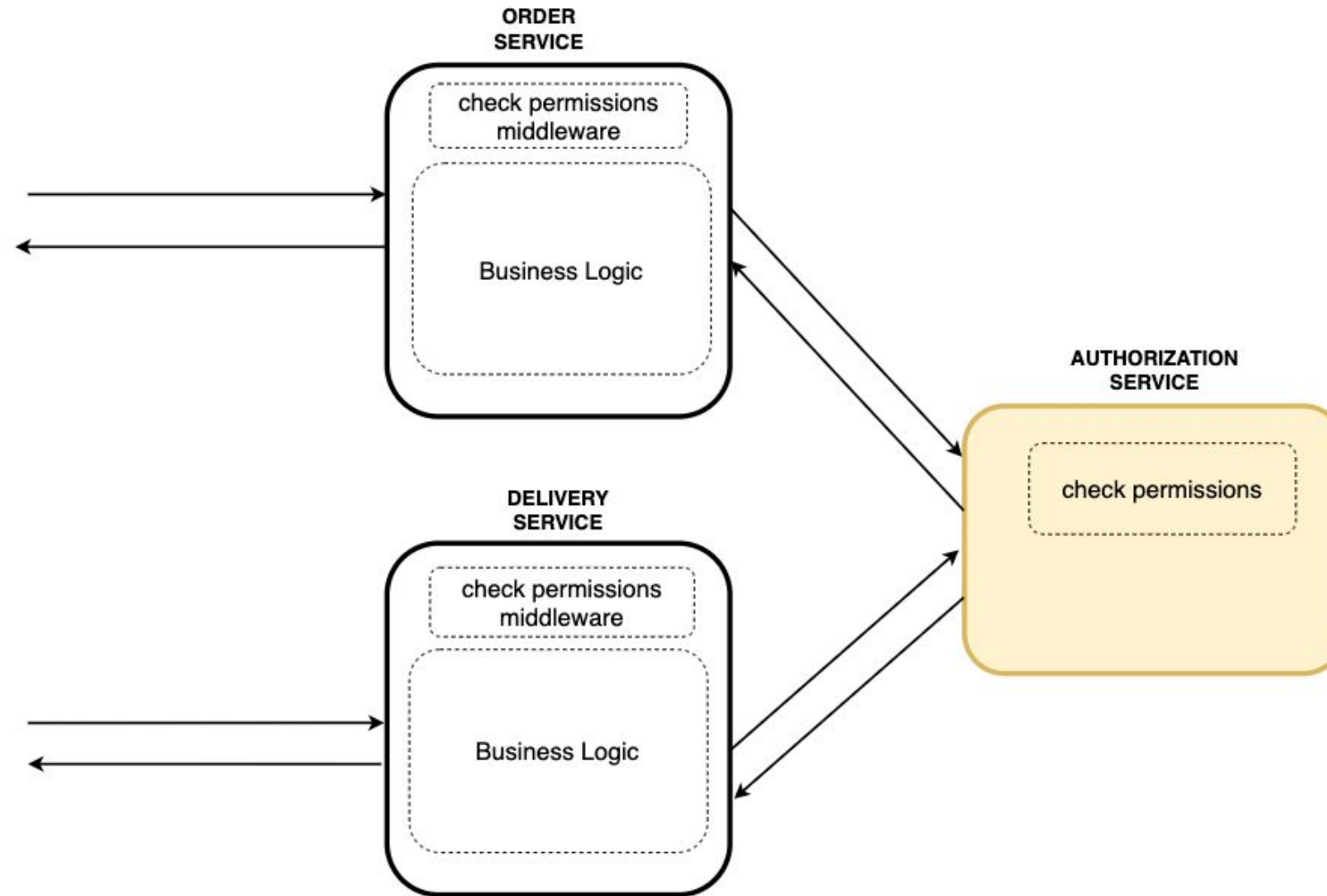
Авторизация

Авторизация

Авторизация – проверка прав. Может ли пользователь совершить то или иное действие.

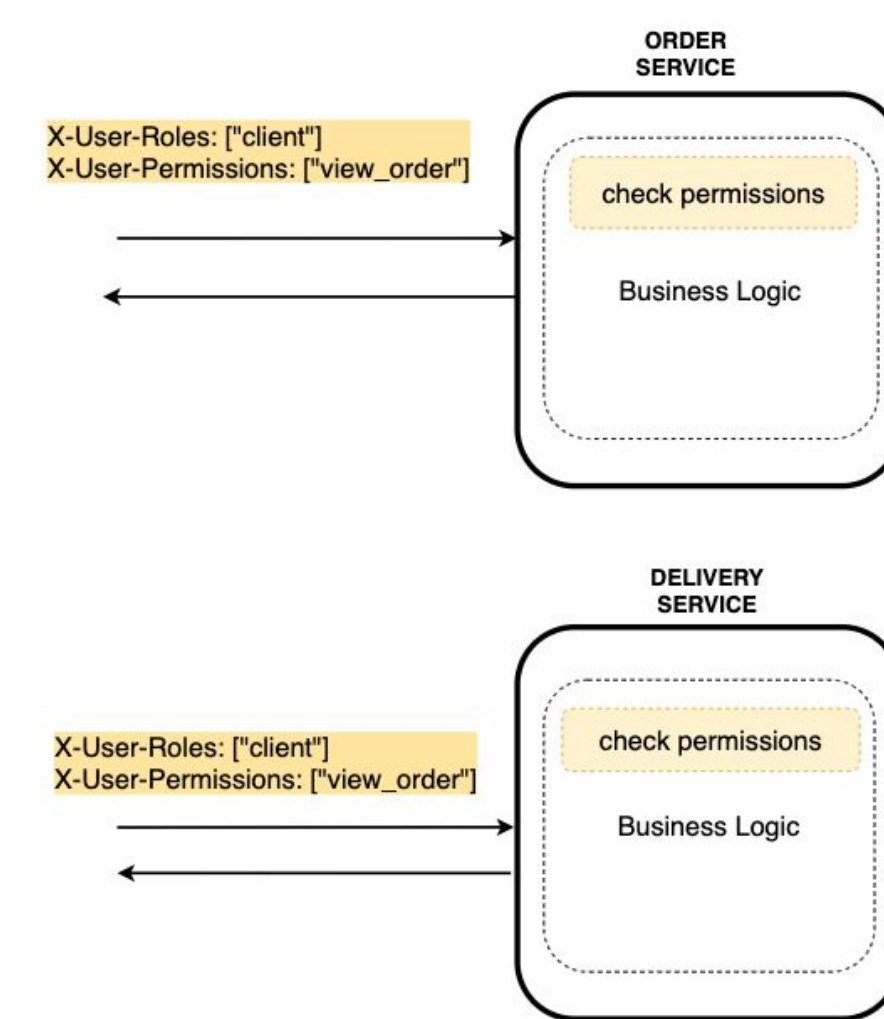
Отдельный сервис для авторизации

Отдельный сервис авторизации (access management)



Проверка прав внутри сервиса

Проверка прав внутри сервиса.



Авторизация

Существуют разные схемы (методы) проверки доступа (access control).

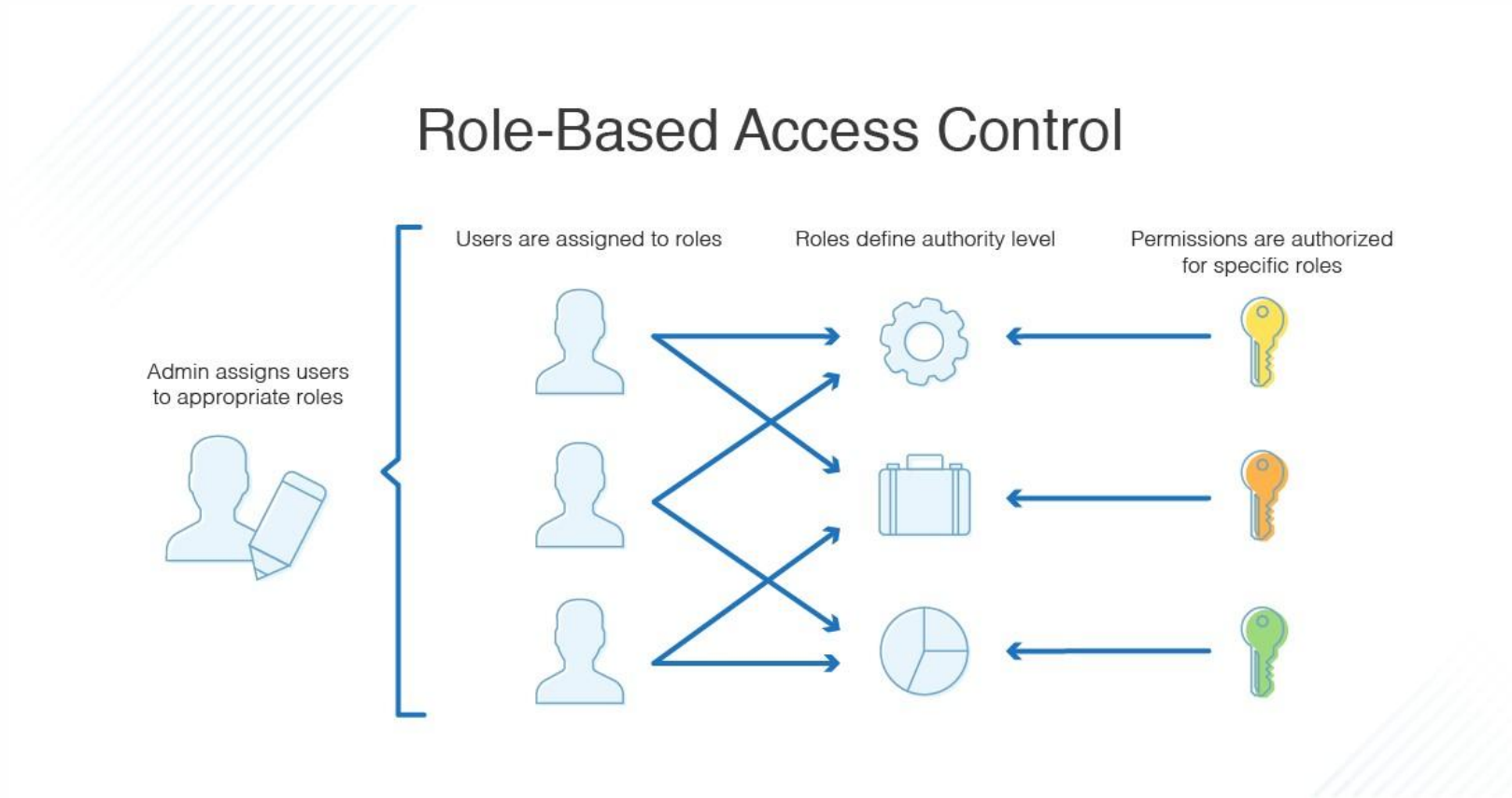
Чаще всего выделяют:

- **RBAC** – role based access control
- **ABAC** – attribute based access control

И другие: DAC, MAC, ...

RBAC

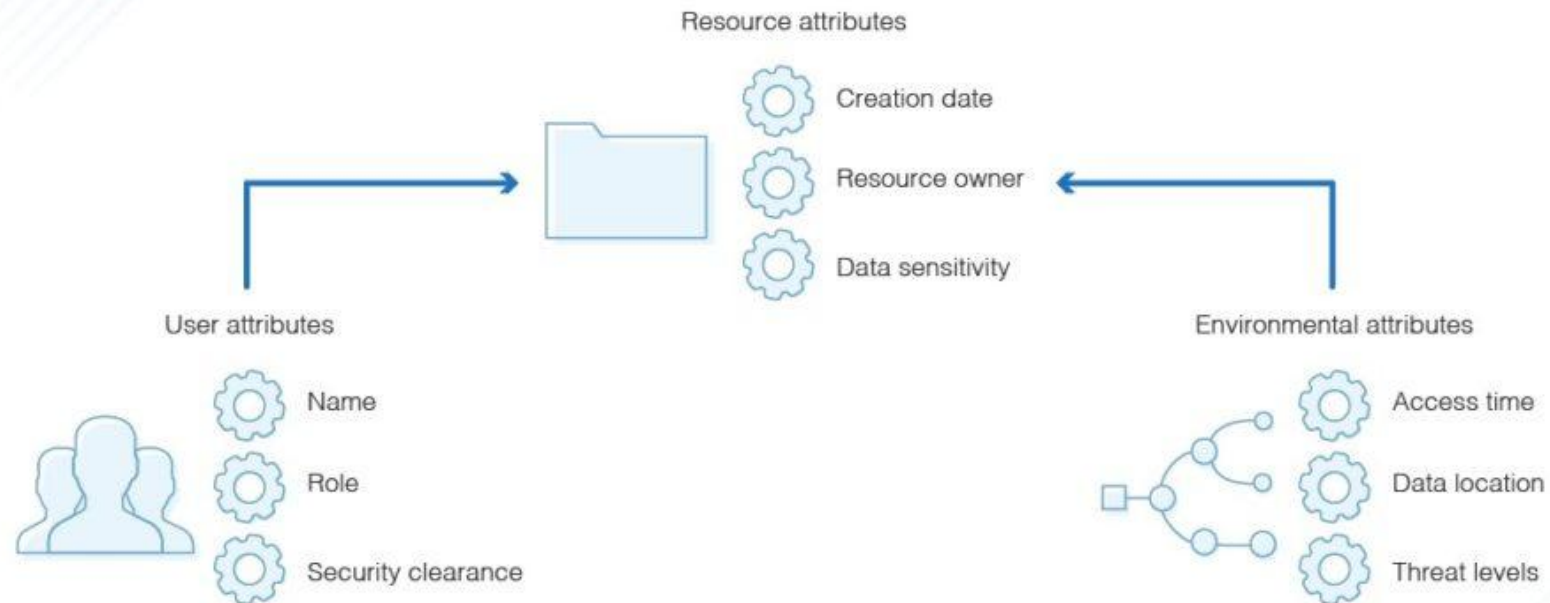
RBAC – role based access control.



ABAC

ABAC –attribute-based access control.

Attribute-Based Access Control



Опрос

<https://otus.ru/polls/31951/>

**Спасибо
за внимание!**

