

Java Pro 7.15



Spring Security.



ПРЕПОДАВАТЕЛЬ



Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

1

ОСНОВНОЙ БЛОК

Введение

- Протокололся
- Ох, рано
- Тестим на месте



Проблема

Реалии современного мира таковы, что ни одно web-приложение, имеющее дело с данными пользователей, не может обойтись без защиты от кибер-мошенников.

Данные ценные сами по себе, но кроме краже данных мошенники или конкуренты могут нанести вред инфраструктуре приложения, либо воспользоваться её ресурсами для проведения атак на сервисы третьей стороны.

Как защитить своё приложение от действий злоумышленников?



Протокололся

Основные понятия кибер-безопасности

Идентификация - это заявление о том, кем является пользователь. В зависимости от ситуации, это может быть имя, адрес электронной почты, номер учетной записи, и т.д. (получение учетной записи (*identity*) пользователя по *username* или *email*)

Аутентификация - предоставление доказательств, что пользователь на самом деле есть тот, кем он идентифицировался (от слова «*authentic*» - истинный, подлинный).

Авторизация - проверка, что пользователю разрешен доступ к запрашиваемому ресурсу (есть права на доступ).

Протокололся

Основные понятия кибер-безопасности

Например, при попытке попасть в закрытый клуб вас *идентифицируют* (спросят ваше имя и фамилию), *аутентифицируют* (попросят показать паспорт и сверят фотографию) и *авторизуют* (проверят, что фамилия находится в списке гостей), прежде чем пустят внутрь.



Протокололся

Основные понятия кибер-безопасности

Авторизация в приложении обычно реализуется с помощью назначения ролей.

Роль – это некоторый статус пользователя, обладая которым можно выполнять те или иные действия, т.е. запрашивать те или иные эндпоинты бэкенда.

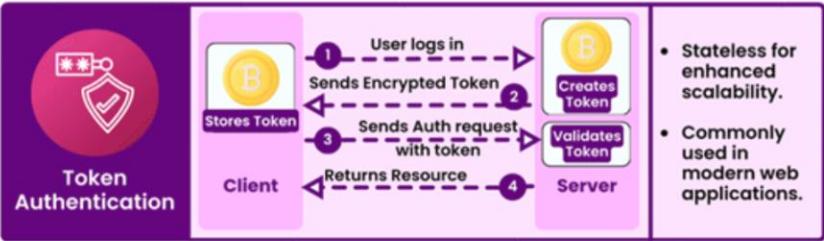
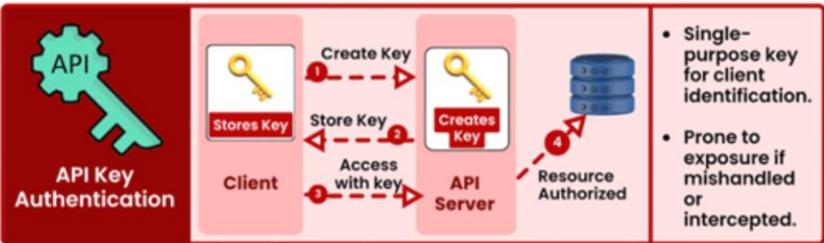
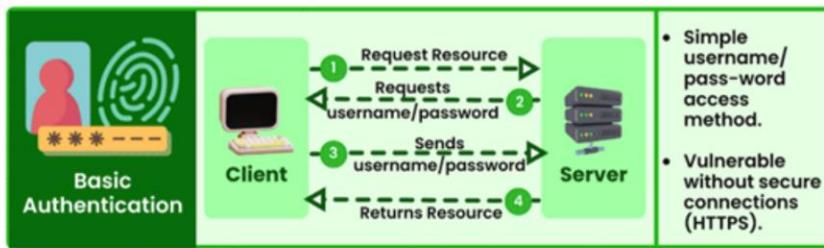
Например, можно выделить такие роли:

Роль	Описание	Доступные действия
unauthorized	незарегистрированный пользователь	Регистрация: /register Аутентификация: /login Восстановление пароля: /recover
user	рядовой зарегистрированный пользователь приложения	Базовый функционал приложения: перечень эндпоинтов приложения
moderator	пользователь с правами модерирования комментариев	Базовый функционал + Модерирование: DELETE /posts/{id}, PUT /user/{id}/ban
admin	пользователь с правами доступа к панели администратора	Базовый функционал + Модерирование + Панель администратора: PATCH /properties и т.д.
owner	владелец приложения	Полный объём прав + Аналитика

Протокололся Способы аутентификации

Существует множество способов аутентификации по *REST API*. Наиболее популярные приведены на схеме.

1 Аутентификация по паролю – метод основанный 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.



Протокололся

Способы аутентификации

1.1 **HTTP authentication** - протокол, описанный в стандартах *HTTP 1.0/1.1*, существует очень давно и до сих пор активно применяется в корпоративной среде. Сервер, при обращении неавторизованного клиента к защищенному ресурсу, отсылает *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.

ACL (*Access Control List*) - список правил, запрещающих или разрешающих использование ресурсов сети: доступа к интернету, телефонии, видеосвязи и т.д.

Протокололся

Способы аутентификации

Существует несколько схем аутентификации, отличающихся по уровню безопасности:

1.1.1 **Basic** - наиболее простая схема, при которой *username* и *password* пользователя передаются в заголовке *Authorization* в незашифрованном виде (в кодировке *base64*). Однако при использовании *HTTPS* (*HTTP over SSL*) протокола, является относительно безопасной.

SSL (*Secure Sockets Layer* - уровень защищённых сокетов) - протокол, который шифрует и защищает данные во время их передачи по интернету.

Для этого он использует специальные криптографические ключи, изменяющие данные до неузнаваемости. В модели *TCP/IP* протокол находится на прикладном уровне, а в *OSI* - между транспортным и прикладным.



Протокололся Способы аутентификации

Для настройки аутентификации **Basic** на компьютере должны быть установлены *SSL*-сертификаты корпоративной сети, в которой Вы работаете. Эти сертификаты обычно предоставляет служба информационной безопасности (ИБ) или команда DevOps.

В *Postman* запросе нужно

перейти на вкладку

Authorization, выбрать

подходящий тип и указать

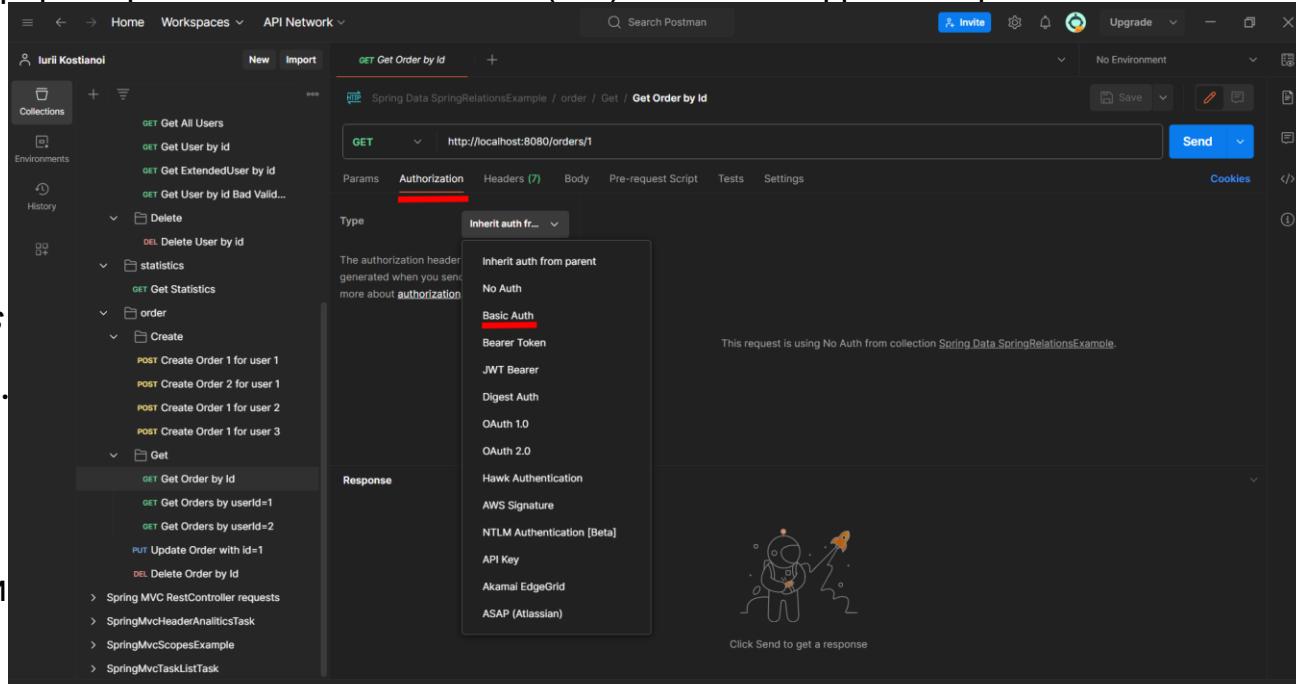
креды (сокр. от *credentials* – данные аутентификации).

Настроить

аутентификацию можно

также на уровне папки или

коллекции.



Протокололся

Способы аутентификации

1.1.2 **Digest** - [challenge-response](#)-схема, при которой сервер посылает уникальное значение *nonce*, а браузер передает *MD5* хэш пароля пользователя, вычисленный с использованием указанного *nonce*. Более безопасная альтернатива *Basic*-схемы при незащищенных соединениях, но подвержена [man-in-the-middle attacks](#) (с заменой схемы на *Basic*). Использование этой схемы не позволяет применить современные хэш-функции для хранения паролей пользователей на сервере.

1.1.3 **NTLM** (известная как *Windows authentication*) - также основана на *challenge-response* подходе, при котором пароль не передается в чистом виде. Эта схема не является стандартом HTTP, но поддерживается большинством браузеров и веб-серверов. Преимущественно используется для аутентификации пользователей *Windows Active Directory* в веб-приложениях. Уязвима к [pass-the-hash](#)-атакам.

Протокололся Способы аутентификации

1.1.4 **Negotiate** - еще одна схема из семейства *Windows authentication*, которая позволяет клиенту выбрать между *NTLM* и *Kerberos* аутентификацией.

Kerberos - более безопасный протокол, основанный на принципе [Single Sign-On](#) (SSO).

Однако он может функционировать, только если и клиент, и сервер находятся в зоне *intranet* и являются частью домена *Windows*.

Стоит отметить, что при использовании *HTTP-аутентификации* у пользователя нет стандартной возможности выйти из веб-приложения, кроме как закрыть все окна браузера.

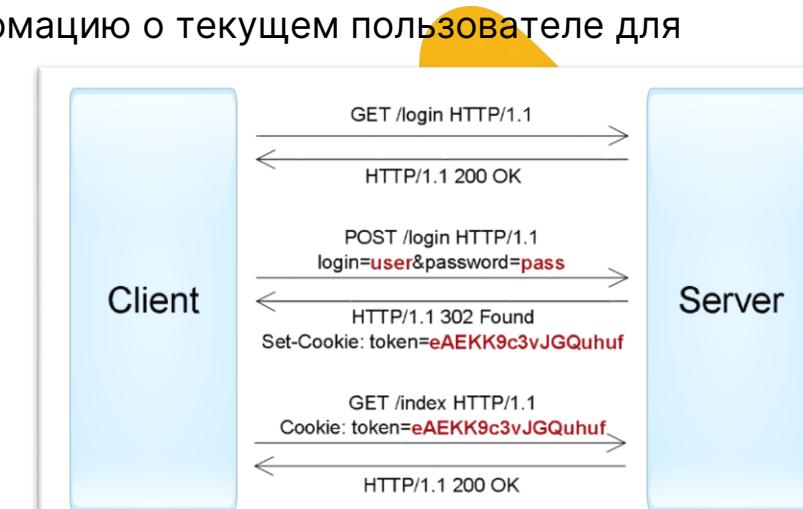


Протокололся

Способы аутентификации

1.2 **Forms authentication** – протокол, работающий по следующему принципу: в веб-приложение включается *HTML*-форма, в которую пользователь должен ввести свои *username/password* и отправить их на сервер через *HTTP POST* для аутентификации. В случае успеха веб-приложение создает *session token*, который обычно помещается в *browser cookies*. При последующих веб-запросах *session token* автоматически передается на сервер и позволяет приложению получить информацию о текущем пользователе для авторизации запроса.

Для этого протокола нет определенного стандарта, поэтому все его реализации специфичны для конкретных систем, а точнее, для модулей аутентификации фреймворков разработки.



Протокололся

Способы аутентификации

Приложение может создать *session token* двумя способами:

- Как идентификатор аутентифицированной сессии пользователя, которая хранится в памяти сервера или в базе данных. Сессия должна содержать всю необходимую информацию о пользователе для возможности авторизации его запросов.
- Как зашифрованный и/или подписанный объект, содержащий данные о пользователе, а также период действия. Этот подход позволяет реализовать *stateless*-архитектуру сервера, однако требует механизма обновления сессионного токена по истечении срока действия.

Перехват *session token* зачастую дает аналогичный уровень доступа, что и знание *username/password*. Поэтому все коммуникации между клиентом и сервером в случае *forms authentication* должны производиться только по защищенному соединению *HTTPS*.

Протокололся

Способы аутентификации

1.3 **Другие протоколы аутентификации по паролю** являются нестандартными, но могут встречаться при разработке клиент-серверных приложений. Существует всего несколько мест, где можно передать *username* и *password* в *HTTP* запросах:

URL query - считается небезопасным вариантом, т. к. строки URL могут запоминаться браузерами, прокси и веб-серверами.

Request body - безопасный вариант, но он применим только для запросов, содержащих тело сообщения (такие как *POST*, *PUT*, *PATCH*).

HTTP header - оптимальный вариант, при этом могут использоваться и стандартный заголовок *Authorization* (например, с *Basic-схемой*), и другие произвольные заголовки.



Протокололся

Способы аутентификации

Аутентификация по паролю считается не очень надежным способом, так как пароль часто можно подобрать, а пользователи склонны использовать простые и одинаковые пароли в разных системах, либо записывать их на бумаге так, что пароль может быть скомпрометирован. Если злоумышленник смог выяснить пароль, то пользователь зачастую об этом не узнает. Кроме того, разработчики приложений могут допустить ряд концептуальных ошибок, упрощающих взлом учетных записей.

Список наиболее часто встречающихся уязвимостей в случае использования аутентификации по паролю можно найти в [приложении А](#).



Протокололся

Способы аутентификации

Аутентификация по паролю считается не очень надежным способом, так как пароль часто можно подобрать, а пользователи склонны использовать простые и одинаковые пароли в разных системах, либо записывать их на бумаге так, что пароль может быть скомпрометирован. Если злоумышленник смог выяснить пароль, то пользователь зачастую об этом не узнает. Кроме того, разработчики приложений могут допустить ряд концептуальных ошибок, упрощающих взлом учетных записей.

Список наиболее часто встречающихся уязвимостей в случае использования аутентификации по паролю можно найти в [приложении А](#).



Протокололся

Способы аутентификации

2 **Аутентификация по сертификатам** – способ аутентификации, подразумевающий владение сертификатом, который позволяет шифровать сообщения.

Сертификат представляет собой набор атрибутов, идентифицирующих владельца, подписанный *certificate authority* (CA). CA выступает в роли посредника, который гарантирует подлинность сертификатов (по аналогии с паспортным столом, выпускающим паспорта).

Также сертификат криптографически связан с закрытым ключом, который хранится у владельца сертификата и позволяет однозначно подтвердить факт владения сертификатом.

У сертификата есть срок действия, что дополнительно позволяет защищать сертификат от подделки.



Протокололся Способы аутентификации

На стороне клиента сертификат вместе с закрытым ключом могут храниться в операционной системе, в браузере, в файле, на отдельном физическом устройстве (smart card, USB token). Обычно закрытый ключ дополнительно защищен паролем или PIN-кодом. В веб-приложениях традиционно используют сертификаты стандарта X.509.

Аутентификация с помощью X.509-сертификата происходит в момент соединения с сервером и является частью протокола *SSL/TLS*. Этот механизм также хорошо поддерживается браузерами, которые позволяют пользователю выбрать и применить сертификат, если веб-сайт допускает такой способ аутентификации.

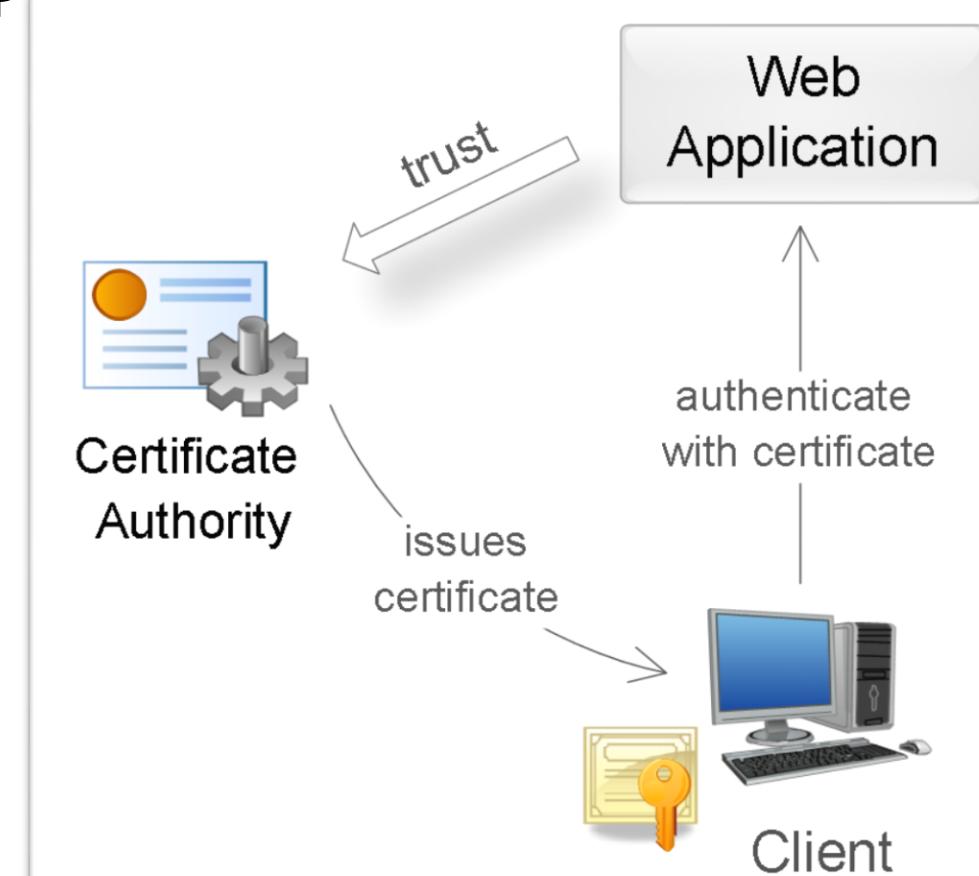


Протокололся

Способы аутентификации

Во время аутентификации сервер выполняет проверку сертификата на основании следующих правил:

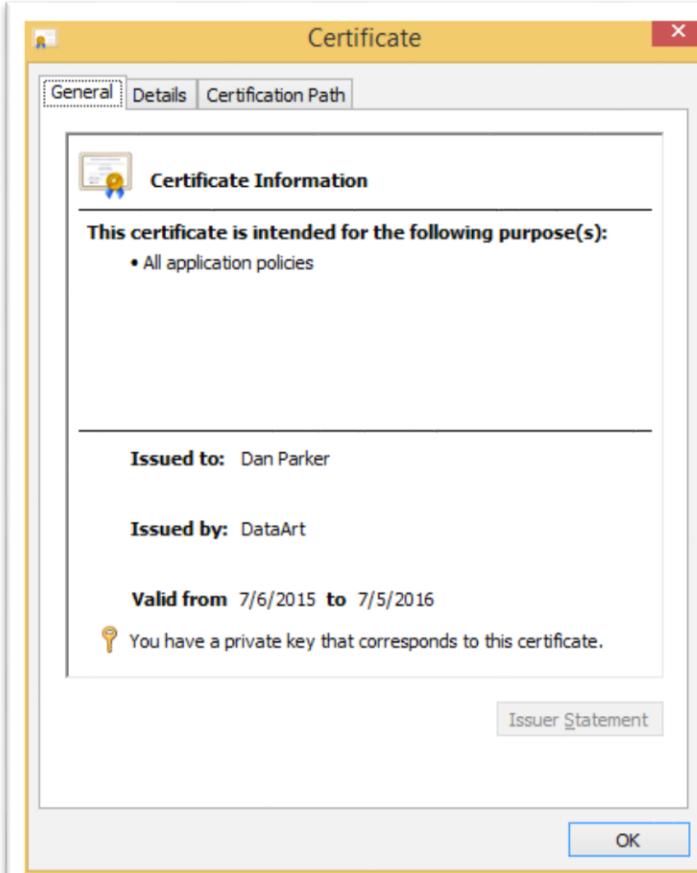
- Сертификат должен быть подписан доверенным *certification authority* (проверка цепочки сертификатов).
- Сертификат должен быть действительным на текущую дату (проверка срока действия).
- Сертификат не должен быть отозван соответствующим *CA* (проверка списков исключения).



Протокололся Способы аутентификации

После успешной аутентификации веб-приложение может выполнить авторизацию запроса на основании таких данных сертификата, как *subject* (имя владельца), *issuer* (эмитент), *serial number* (серийный номер сертификата) или *thumbprint* (отпечаток открытого ключа сертификата).

Аутентификация по сертификату - более надежный способ, чем аутентификация по паролю за счёт создания и использования цифровой подписи, доказывающей факт применения закрытого ключа в конкретной ситуации ([non-repudiation](#)). Однако трудности с распространением и поддержкой сертификатов делает такой способ аутентификации малодоступным в широких кругах.



Протокололся Способы аутентификации

3 **Аутентификация по одноразовым паролям** - обычно применяется дополнительно к аутентификации по паролям для реализации [two-factor authentication](#) (2FA). В этой концепции пользователю необходимо предоставить данные двух типов для входа в систему: что-то, что он знает (например, пароль), и что-то, чем он владеет (например, устройство для генерации одноразовых паролей или номер телефона). Такая аутентификация позволяет подтверждать особо важные действия пользователя (например, перевод денег в банковском приложении).



Протокололся

Способы аутентификации

Наиболее популярные источники для создания одноразовых паролей:

- Аппаратные или программные токены, которые могут генерировать одноразовые пароли на основании секретного ключа, введенного в них, и текущего времени. Секретные ключи пользователей, являющиеся фактором владения, также хранятся на сервере, что позволяет выполнить проверку введенных одноразовых паролей. Пример аппаратной реализаций токенов - RSA SecurID; программной - приложение Google Authenticator, Aladdin SecurLogon и др.

Аппаратный токен RSA SecurID генерирует новый код каждые 30 секунд.



Протокололся

Способы аутентификации

- Случайно генерируемые коды, передаваемые пользователю через SMS, PUSH-уведомления или другой канал связи. В этой ситуации фактор владения - телефон пользователя (точнее - SIM-карта, привязанная к определенному номеру) (любое банковское приложение).



Протокололся

Способы аутентификации

- Распечатка или *scratch card* со списком заранее сформированных одноразовых паролей. Для каждого нового входа в систему требуется ввести новый одноразовый пароль с указанным номером.



Протокололся Способы аутентификации

В веб-приложениях такой механизм аутентификации часто реализуется посредством расширения *forms authentication*: после первичной аутентификации по паролю, создается сессия пользователя, однако в контексте этой сессии пользователь не имеет доступа к приложению до тех пор, пока он не выполнит дополнительную аутентификацию по одноразовому паролю.



Протокололся

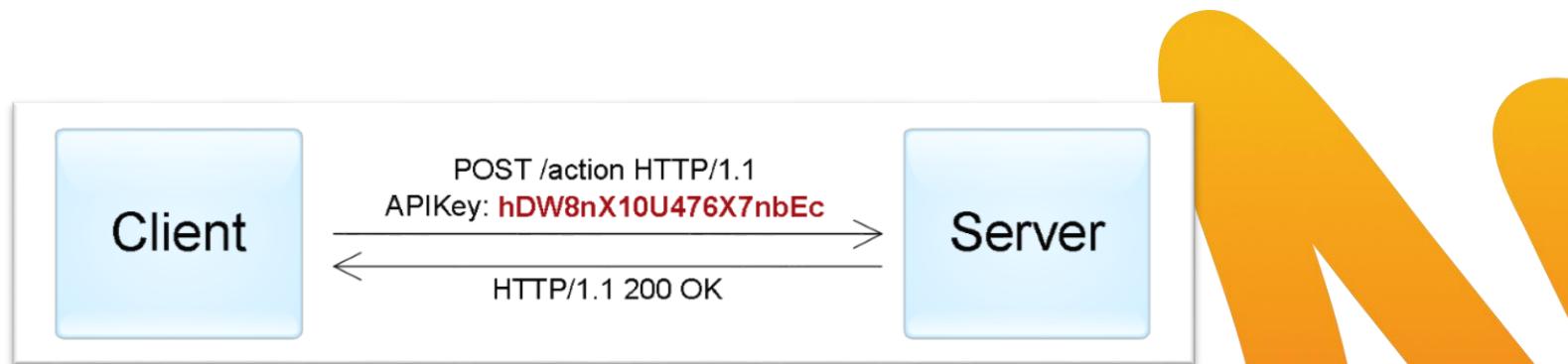
Способы аутентификации

4 Аутентификация по ключам доступа - чаще всего используется для аутентификации устройств, сервисов или других приложений при обращении к веб-сервисам. В качестве секрета применяются ключи доступа (*access key, API key*) - длинные уникальные строки, содержащие произвольный набор символов, по сути заменяющие собой комбинацию *username/password*. Обычно сервер генерирует ключи доступа по запросу пользователей, которые далее сохраняют эти ключи в клиентских приложениях. При создании ключа также возможно ограничить срок действия и уровень доступа, который получит клиентское приложение при аутентификации с помощью этого ключа.

Протокололся

Способы аутентификации

С технической точки зрения, здесь не существует единого протокола: ключи могут передаваться в разных частях *HTTP*-запроса. Наиболее оптимальный вариант – использование *HTTP header*. Например *Bearer* для передачи токена в заголовке (*Authorization: Bearer [token]*). Чтобы избежать перехвата ключей, соединение с сервером должно быть обязательно защищено протоколом *SSL/TLS*.



Протокололся

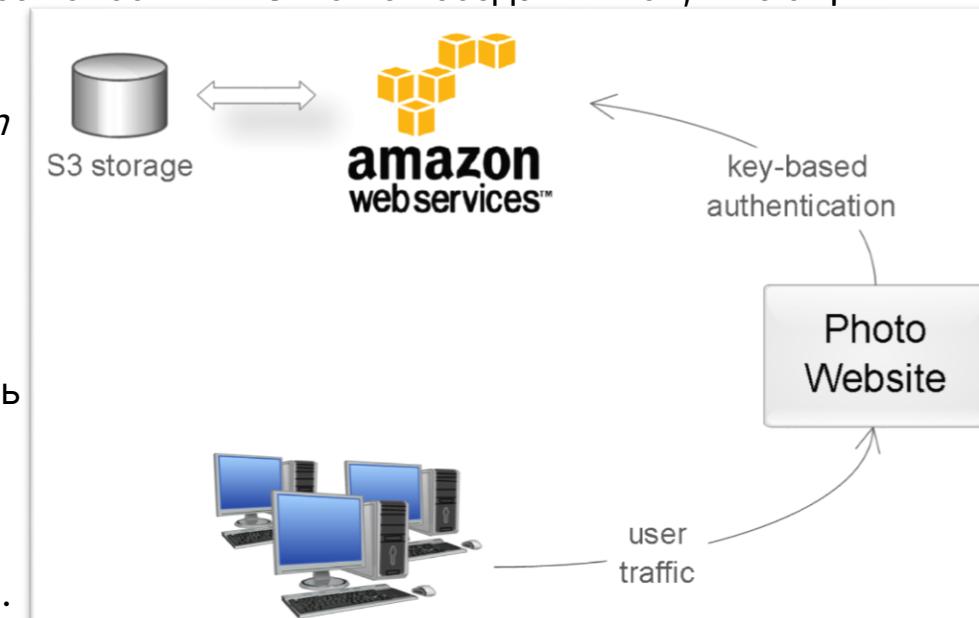
Способы аутентификации

Хороший пример применения аутентификации по ключу - облако *Amazon Web Services*.

Предположим, у пользователя есть веб-приложение, позволяющее загружать и просматривать фотографии, и он хочет использовать сервис *Amazon S3* для хранения файлов. В таком случае, пользователь через консоль *AWS* может создать ключ, имеющий ограниченный доступ к облаку:

только чтение/запись его файлов в *Amazon S3*. Этот ключ в результате можно применить для аутентификации веб-приложения в облаке *AWS*.

Использование ключей позволяет избежать передачи пароля пользователя сторонним приложениям, усложнить подбор ключа (пароль короче) и отвязать ключ от учётки.

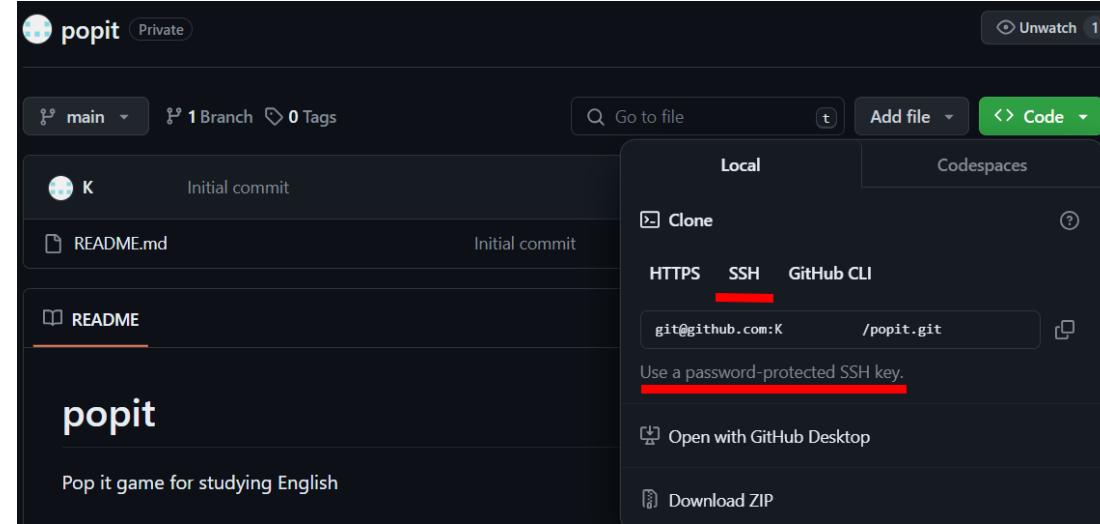


Протокололся

Способы аутентификации

Существуют более сложные схемы аутентификации по ключам для незащищенных соединений. В этом случае, ключ обычно состоит из двух частей: публичной (*public*) и секретной (*private*). Публичная часть используется для идентификации клиента, а секретная часть позволяет сгенерировать подпись. Это позволяет избежать передачи всего ключа в оригинальном виде и защищает от [*replay attacks*](#).

Например, для работы с *GitHub* через *SSH* нужно сгенерировать приватный и публичный ключи.



Протокололся

Способы аутентификации

5 **Аутентификация по токенам** - чаще всего применяется при построении распределенных систем *Single Sign-On* (SSO), где одно приложение (*service provider (SP)* или *relying party*) делегирует функцию аутентификации пользователей другому приложению (*identity provider (IP)* или *authentication service*).

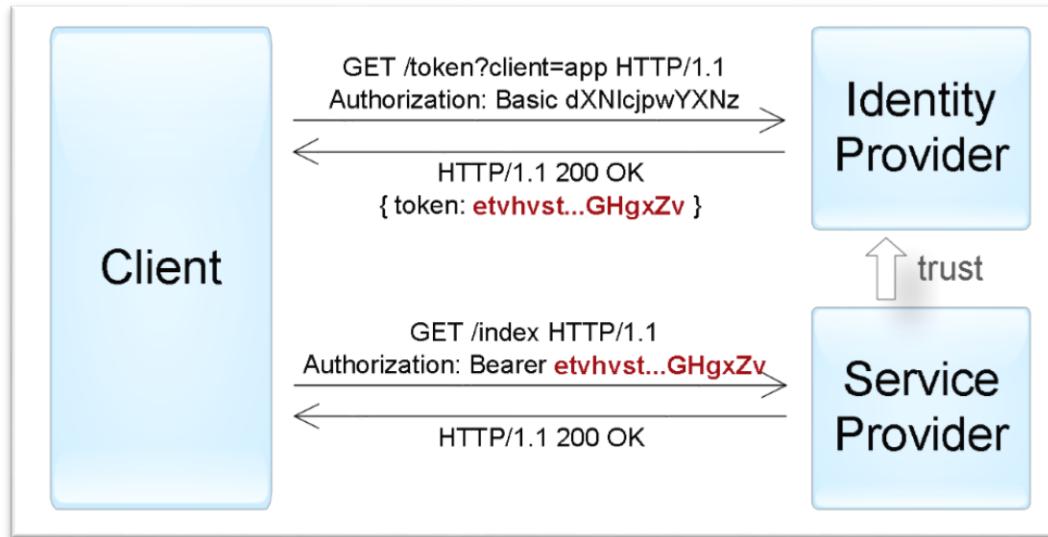
Типичный пример этого способа - вход в приложение через учетную запись в социальных сетях. Здесь социальные сети являются сервисами аутентификации, а приложение доверяет функцию аутентификации пользователей социальным сетям.

Реализация этого способа заключается в том, что *identity provider* предоставляет достоверные сведения о пользователе в виде токена, а *service provider* приложение использует этот токен для идентификации, аутентификации и авторизации пользователя.

Протокололся

Способы аутентификации

1. Клиент аутентифицируется в *identity provider* одним из способов, специфичным для него (пароль, ключ доступа, сертификат, *Kerberos*, итд.).
2. Клиент просит *identity provider* предоставить ему токен для конкретного приложения. *Identity provider* генерирует токен и отправляет его клиенту.
3. Клиент аутентифицируется в приложении при помощи этого токена.



Протокололся Способы аутентификации

Токен – это структура данных, которая содержит информацию о том, кто сгенерировал токен, кто может быть получателем токена, срок действия, набор сведений о самом пользователе (*claims*). Кроме того, токен дополнительно подписывается для предотвращения несанкционированных изменений и гарантий подлинности.



Протокололся

Способы аутентификации

Simple Web Token (SWT) - наиболее простой формат, представляющий собой набор произвольных пар имя-значение в формате кодирования *HTML form*. Токен подписывается с помощью симметричного ключа, таким образом оба *IP*- и *SP*-приложения должны иметь этот ключ для возможности создания/проверки токена.

Пример *SWT* токена (после декодирования):

```
Issuer=http://auth.myservice.com&
Audience=http://myservice.com&
ExpiresOn=1435937883&
UserName=John Smith&
UserRole=Admin&
HMACSHA256=KOUQRPSpy64rvT2KnYyQKtFFXUlgnnesSpE7ADA4o9w
```

Протокололся

Способы аутентификации

JSON Web Token (JWT) - содержит три блока, разделенных точками: заголовок, набор полей (*claims*) и подпись. Первые два блока представлены в *JSON*-формате и дополнительно закодированы в формат *base64*. Набор полей содержит произвольные пары имя/значения, притом стандарт *JWT* определяет несколько зарезервированных имен (*iss*, *aud*, *exp* и другие). Подпись может генерироваться как при помощи симметричных алгоритмов шифрования, так и асимметричных. Кроме того, существует отдельный стандарт, отписывающий формат зашифрованного *JWT*-токена.

Пример подписанного *JWT* токена (после декодирования 1 и 2 блоков).

```
{ «alg»: «HS256», «typ»: «JWT» }.
{ «iss»: «auth.myservice.com», «aud»: «myservice.com», «exp»: «1435937883», «userName»: «John Smith», «userRole»: «Admin» }.
S9Zs/8/uEGGTVVtLggFTizCsMtwOJnRhjaQ2BMUQhcY
```



Протокололся

Способы аутентификации

Security Assertion Markup Language (SAML) - определяет токены (*SAML assertions*) в *XML*-формате, включающем информацию об эмитенте, о субъекте, необходимые условия для проверки токена, набор дополнительных утверждений (*statements*) о пользователе.

Подпись *SAML*-токенов осуществляется при помощи ассиметричной криптографии. Кроме того, в отличие от предыдущих форматов, *SAML*-токены содержат механизм для подтверждения владения токеном, что позволяет предотвратить перехват токенов через *man-in-the-middle*-атаки при использовании незащищенных соединений.

Стандарт *SAML* описывает также способы взаимодействия и протоколы между *IP* и *SP* для обмена данными аутентификации и авторизации посредством токенов.

Протокололся

Способы аутентификации

5.1 **WS-Trust** и **WS-Federation** – стандарты, входящие в группу стандартов *WS-**, описывающих *SOAP/XML* веб-сервисы. Аналогичны SAML, поэтому достаточно сложные, из-за чего используются преимущественно в корпоративных сценариях.

WS-Trust описывает интерфейс сервиса авторизации, именуемого *Secure Token Service (STS)*. Этот сервис работает по протоколу *SOAP* и поддерживает создание, обновление и аннулирование токенов. При этом стандарт допускает использование токенов различного формата, однако на практике в основном используются *SAML-токены*.

WS-Federation касается механизмов взаимодействия сервисов между компаниями, в частности, протоколов обмена токенов. При этом *WS-Federation* расширяет функции и интерфейс сервиса *STS*, описанного в стандарте *WS-Trust*.

Протокололся Способы аутентификации

5.2 OAuth и OpenID Connect –

OAuth (*Open Authorization*) не описывает протокол аутентификации пользователя. Вместо этого он определяет механизм получения доступа одного приложения к другому от имени пользователя. Однако существуют схемы, позволяющие осуществить аутентификацию пользователя на базе этого стандарта.

Сейчас *OAuth 2.0* очень популярен и используется повсеместно для предоставления делегированного доступа и третье-сторонней аутентификации пользователей.



Протокололся

Способы аутентификации

Чтобы лучше *OAuth*, рассмотрим пример веб-приложения, которое помогает пользователям планировать путешествия. Как часть функциональности оно умеет анализировать почту пользователей на наличие писем с подтверждениями бронирований и автоматически включать их в планируемый маршрут. Возникает вопрос, как это веб-приложение может безопасно получить доступ к почте пользователей, например, к Gmail?

- *Попросить пользователя указать данные своей учетной записи?* - плохой вариант.
- *Попросить пользователя создать ключ доступа?* - возможно, но весьма сложно.

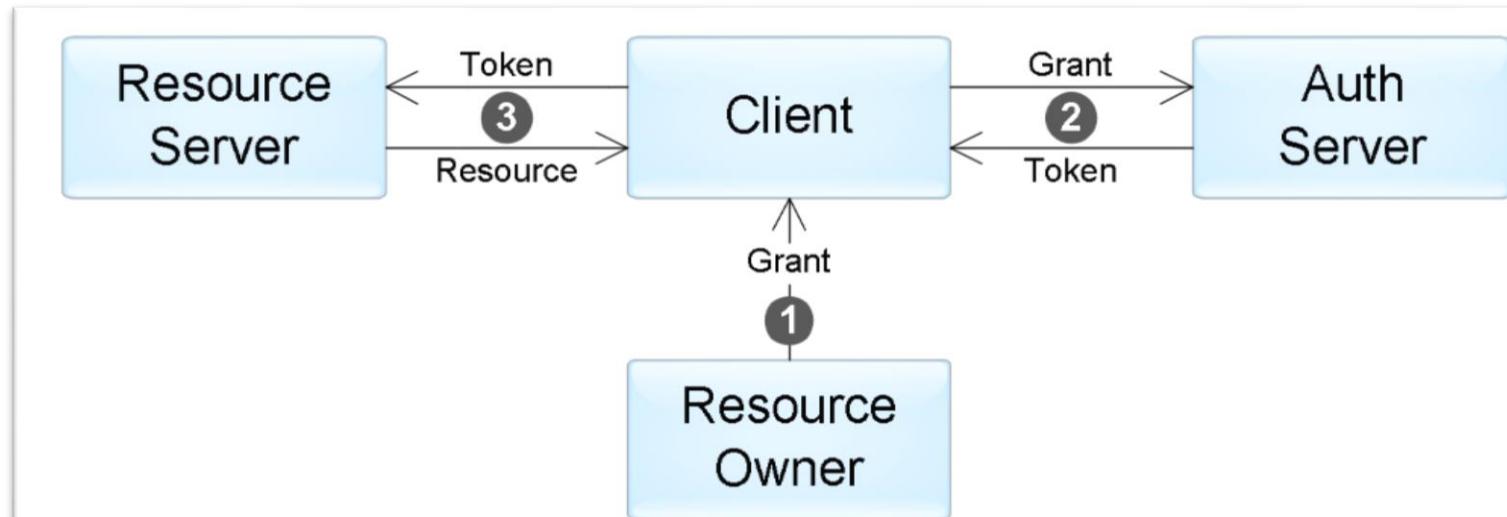
Как раз эту проблему и позволяет решить стандарт *OAuth*: он описывает, как приложение путешествий (*client*) может получить доступ к почте пользователя (*resource server*) с разрешения пользователя (*resource owner*).



Протокололся

Способы аутентификации

1. Пользователь (*resource owner*) дает разрешение приложению (*client*) на доступ к определенному ресурсу в виде *гранта*.
2. Приложение обращается к серверу авторизации и получает токен доступа к ресурсу в обмен на свой грант.
3. Приложение использует этот токен для получения требуемых данных от сервера ресурсов.



Протокололся

Виды грантов

1. *Authorization Code* - этот грант пользователь может получить от сервера авторизации после успешной аутентификации и подтверждения согласия на предоставление доступа. Такой способ наиболее часто используется в веб-приложениях.
2. *Implicit* - применяется, когда у приложения нет возможности безопасно получить токен от сервера авторизации (например, *JavaScript*-приложение в браузере). В этом случае грант представляет собой токен, полученный от сервера авторизации, а шаг 2 (см. предыдущий слайд) исключается из сценария выше.
3. *Resource Owner Password Credentials* - грант представляет собой пару *username/password* пользователя. Может применяться, если приложение является «интерфейсом» для сервера ресурсов (например, приложение - мобильный клиент для *Gmail*).
4. *Client Credentials* - в этом случае нет никакого пользователя, а приложение получает доступ к своим ресурсам при помощи своих ключей доступа (исключается шаг 1).

Протокололся

Способы аутентификации

OAuth не определяет формат токена, который получает приложение: в сценариях, адресуемых стандартом, приложению нет необходимости анализировать токен, т. к. он лишь используется для получения доступа к ресурсам. Поэтому ни токен, ни грант сами по себе не могут быть использованы для аутентификации пользователя. Однако если приложению необходимо получить достоверную информацию о пользователе, существуют несколько способов это сделать:

API сервера ресурсов обычно включает операцию, предоставляющую информацию о самом пользователе (например, `/me` в *Facebook API*). Приложение может выполнять эту операцию каждый раз после получения токена для идентификации клиента.

Использовать стандарт **OpenID Connect**, разработанный как слой учетных данных поверх *OAuth*. В соответствии с этим стандартом, сервер авторизации предоставляет дополнительный *identity token* на шаге 2. Этот токен в формате *JWT* будет содержать набор определенных полей (*claims*) с информацией о пользователе.

Проблема

При таком разнообразии и сложности реализации различных протоколов довольно просто запутаться в организации информационной безопасности приложения.

Что для нас подготовил Spring, чтобы мы легко могли реализовать аутентификацию и авторизацию?

Не знаешь как
реализовать security
микросервисов?

Жаль



Ох, рано

Spring Security

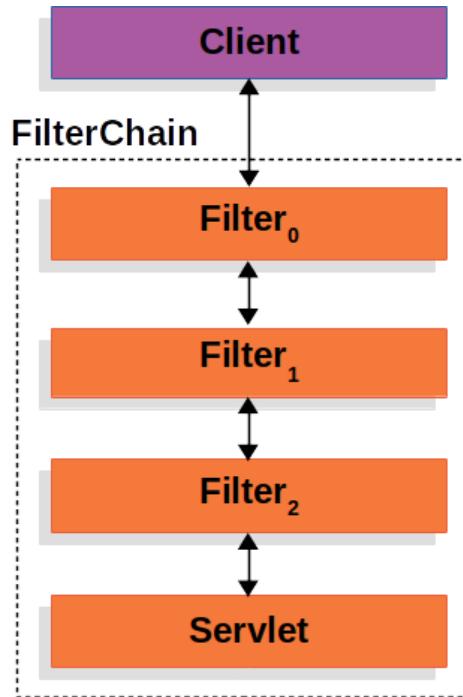
Spring Security – ещё один компонент *Spring Framework*, который помогает обеспечивать безопасность web-приложений (как монолитных, так и распределённых). В *Spring Security* заложены механизмы для обеспечения аутентификации и авторизации пользователей и сторонних сервисов в Вашем приложении.



Ох, рано

Spring Security

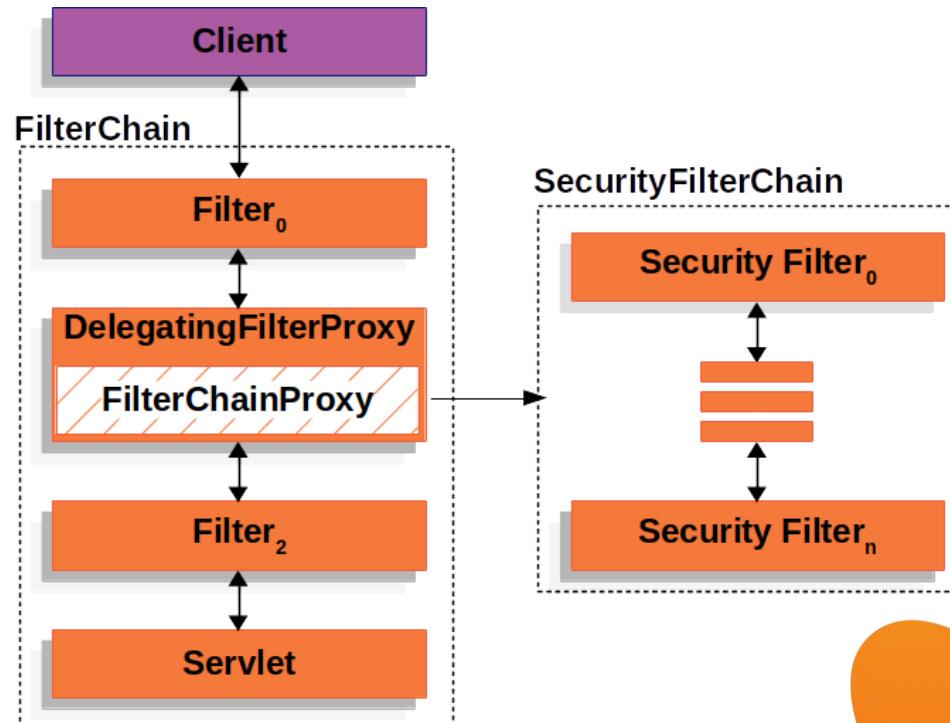
В основе *Spring Security* лежит использование фильтров web-приложения.



Ох, рано

Spring Security

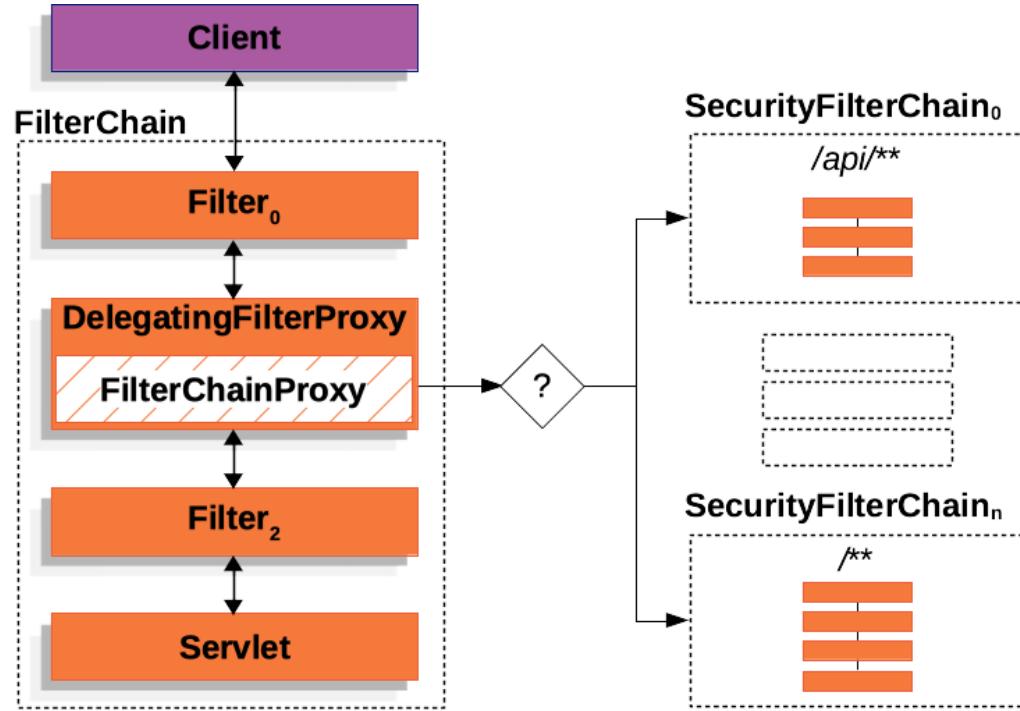
Spring Security встраивается в цепочку фильтров приложения, добавляя свою цепочку фильтров. Если пользователь не прошёл аутентификацию, то его запрос не достигнет сервлета.



Ох, рано

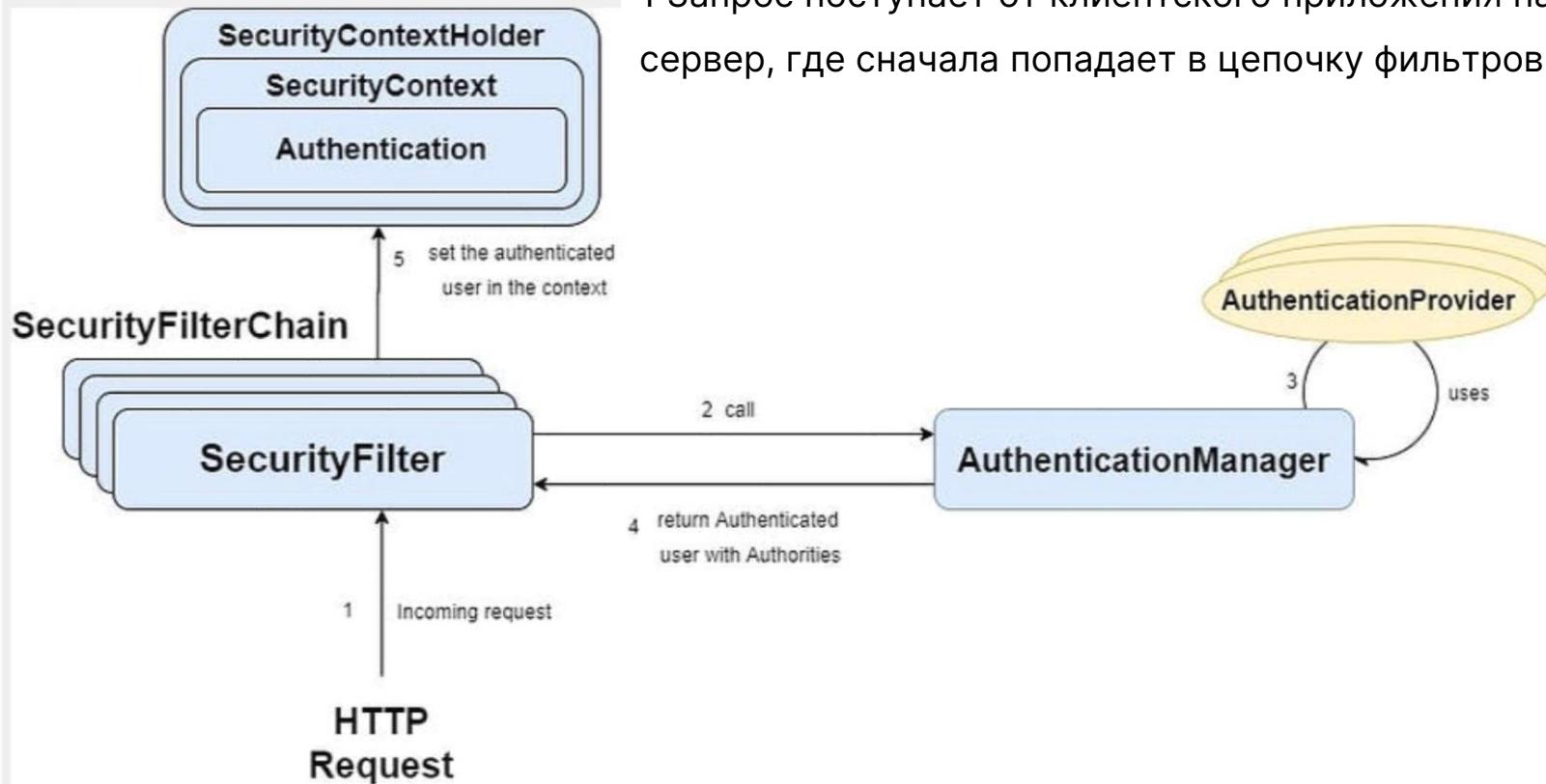
Spring Security

Spring Security позволяет также выбирать встраиваемую цепочку фильтров на основании заданных условий.



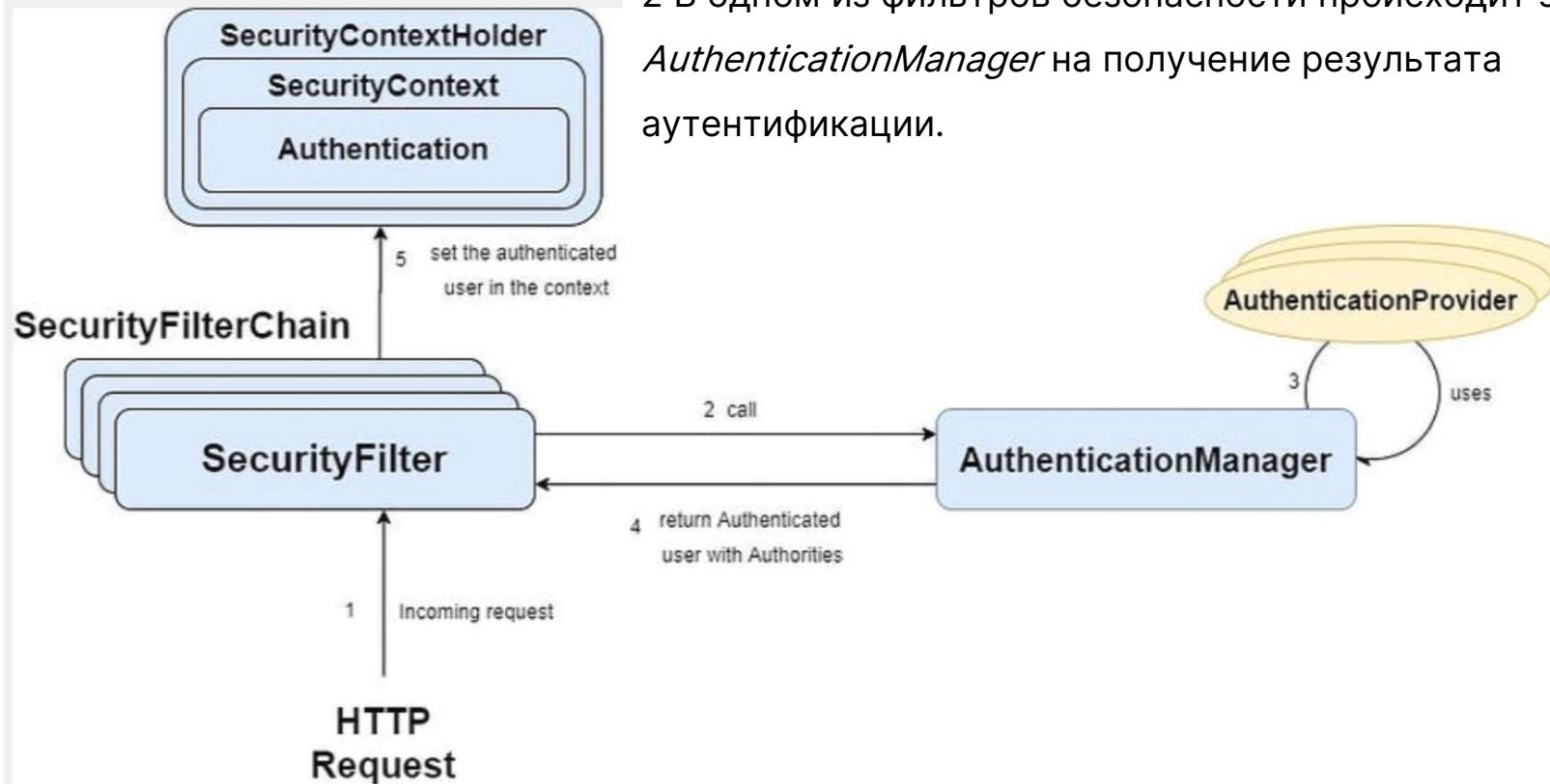
Ох, рано

Аутентификация в Spring Security



Ох, рано

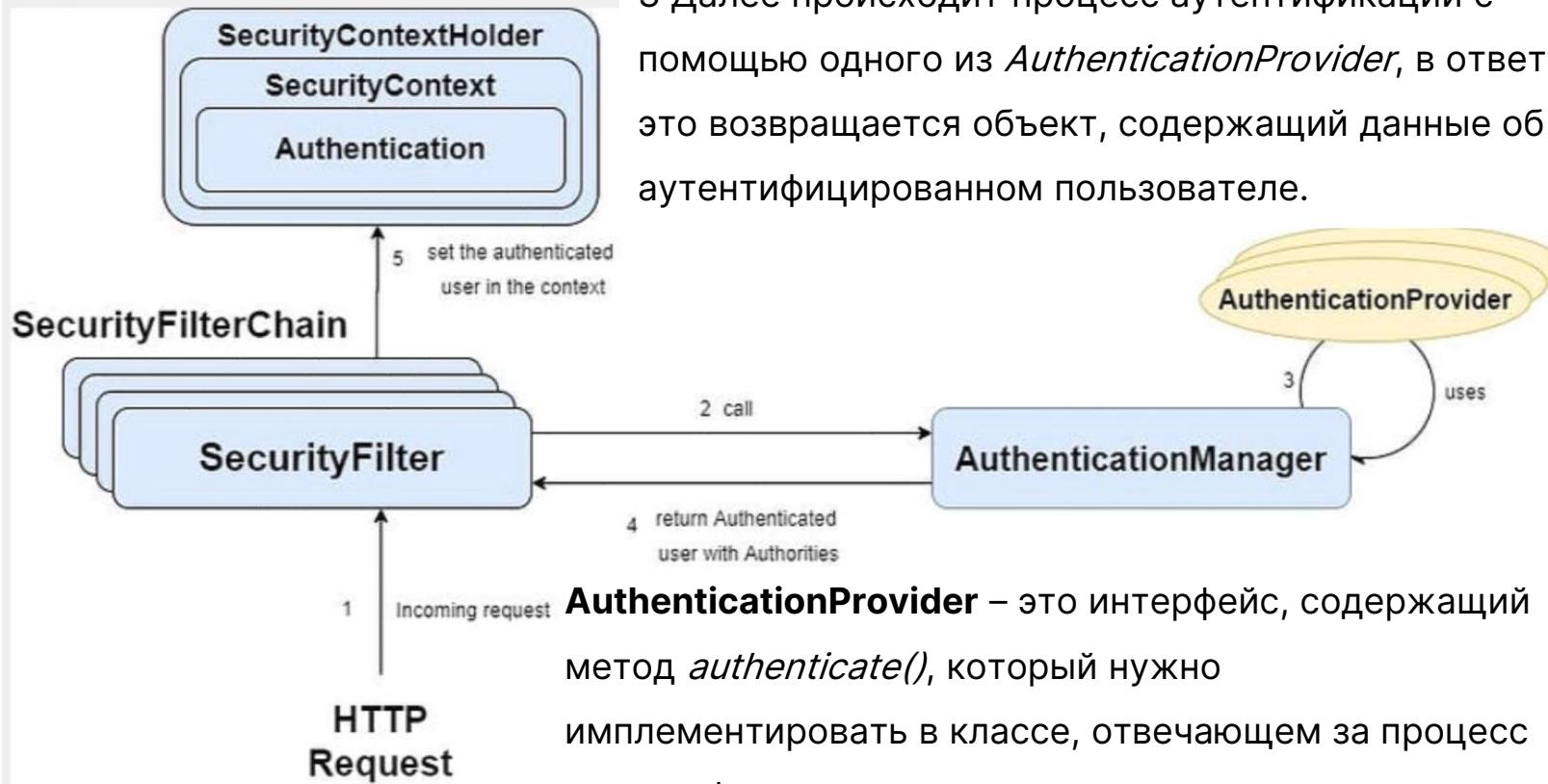
Аутентификация в Spring Security



2 В одном из фильтров безопасности происходит запрос *AuthenticationManager* на получение результата аутентификации.

Ох, рано

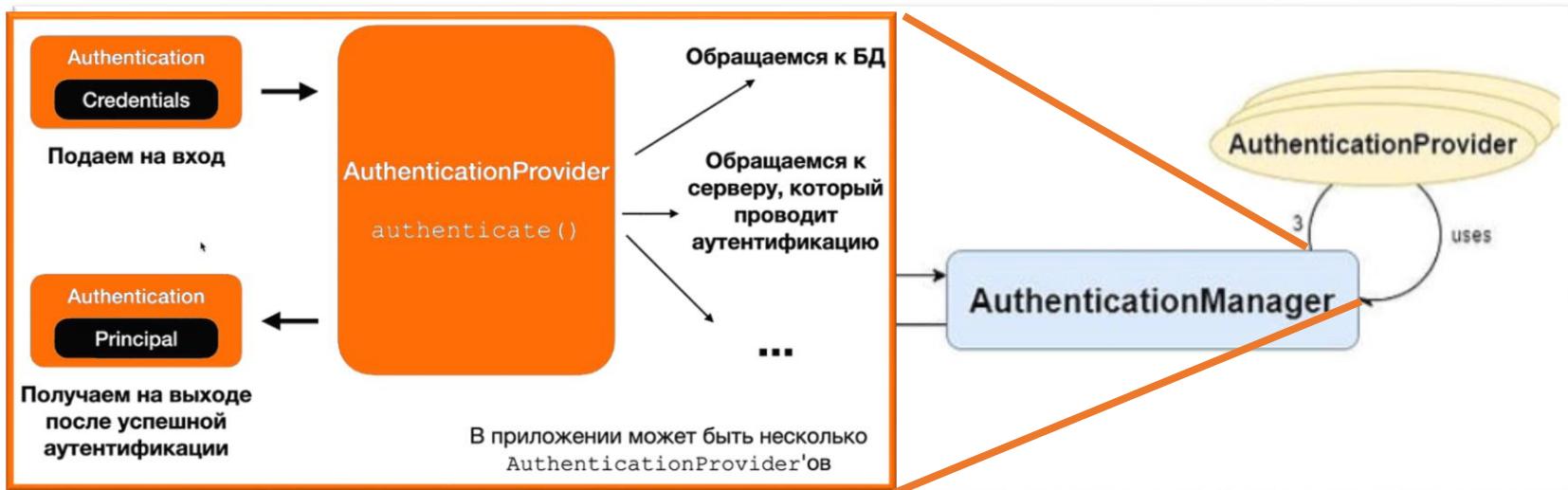
Аутентификация в Spring Security



Ох, рано

Аутентификация в Spring Security

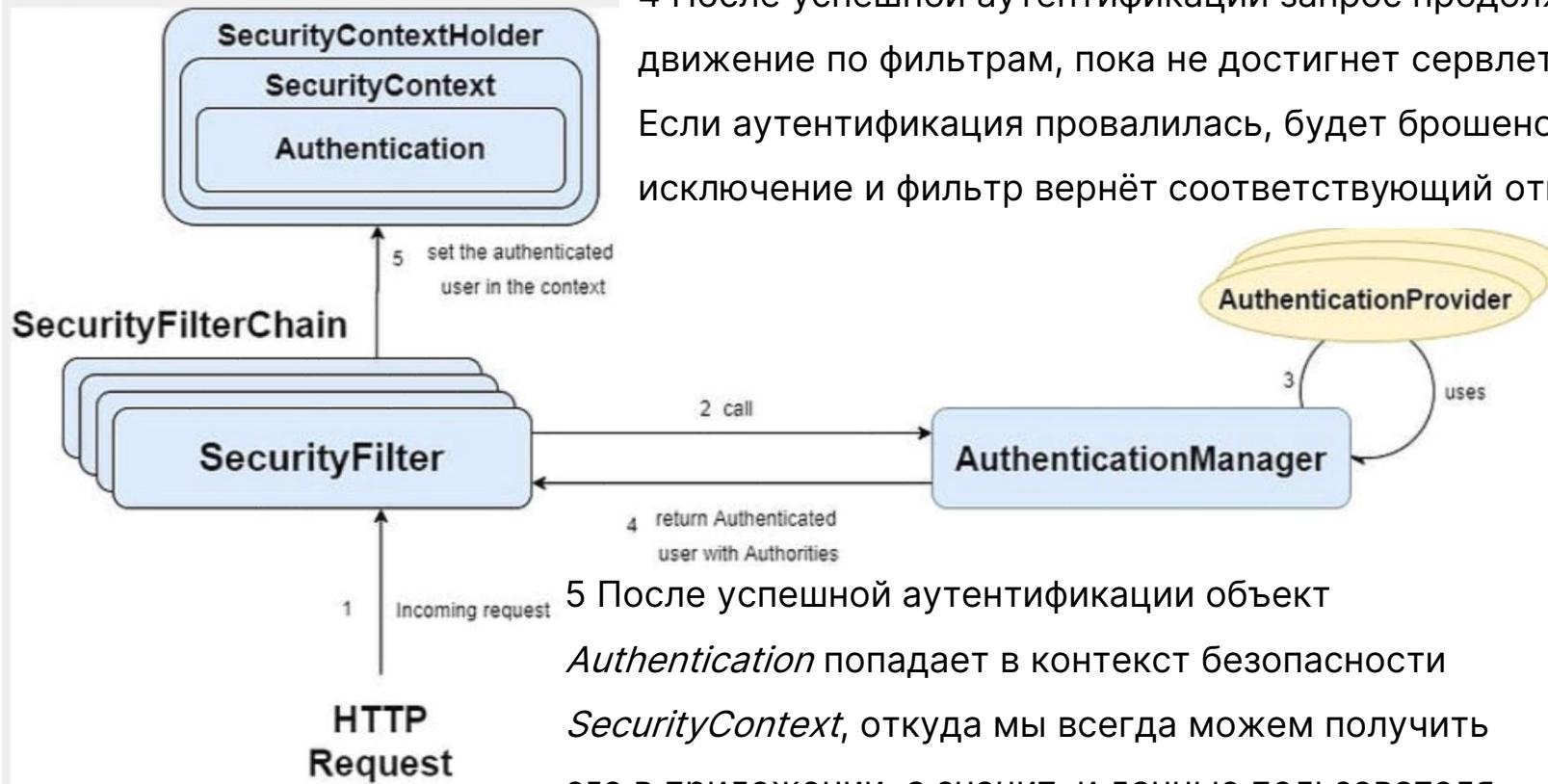
AuthenticationProvider получает объект *Authentication*, содержащий необходимые *credentials* (обычно пароль и логин) и возвращает объект *Authentication*, содержащий объект *Principal* – обертку, в которой будут лежать те самые данные пользователя (*user*).



Сам же процесс аутентификации по паролю и логину обычно происходит с помощью сверки данных с записями в БД, либо с помощью вызова стороннего сервера аутентификации.

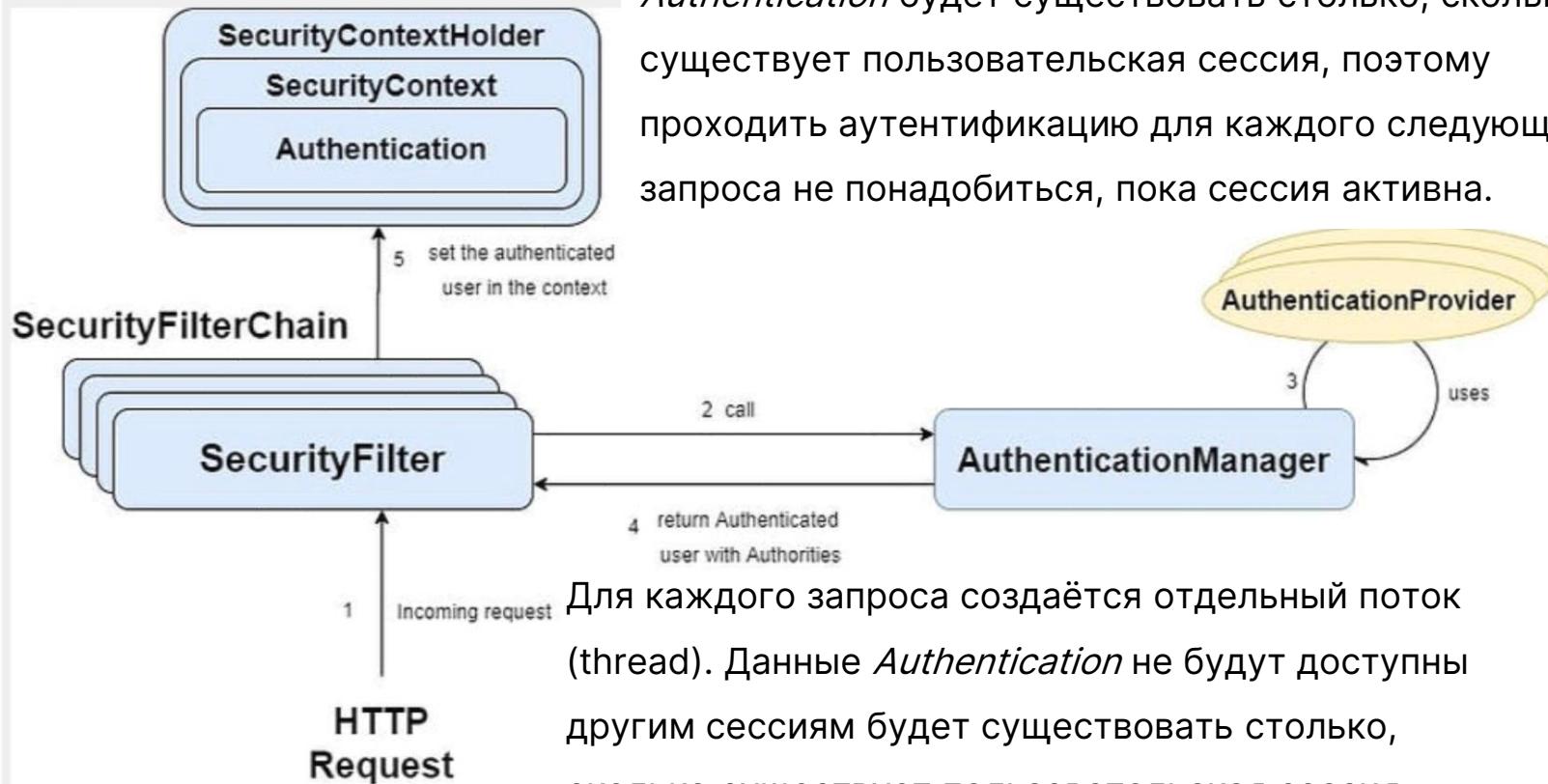
Ох, рано

Аутентификация в Spring Security



Ох, рано

Аутентификация в Spring Security



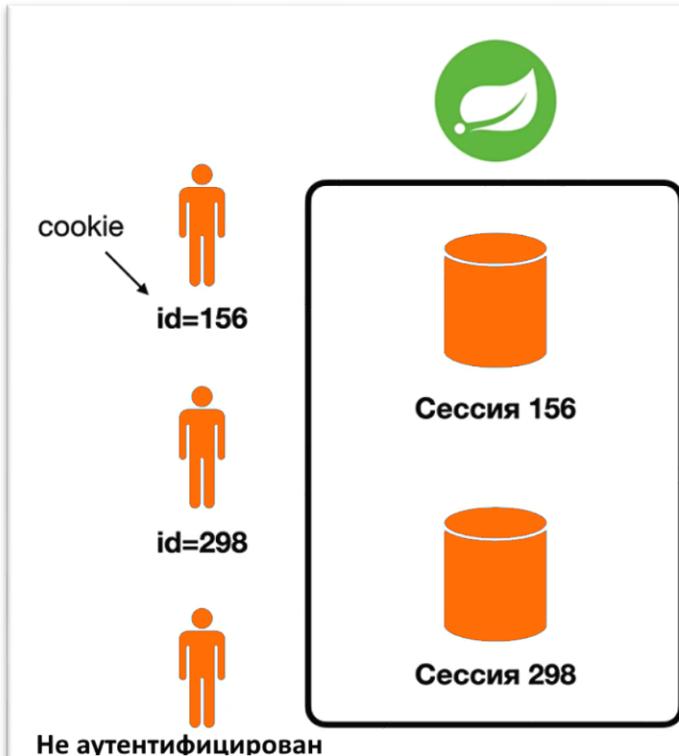
Ох, рано

Сессии и cookies

Для того, чтобы пользовательскому приложению не приходилось каждый раз передавать пароль и логин при открытии сессии ей назначается *cookie* с именем JSESSIONID.

Сервер формирует пару *cookie* (ключ-значение) и в ответе на запрос клиента добавляет заголовок *Set-Cookie* со значением (например, *Set-Cookie: token=123abc*)

Клиенты могут по-разному обрабатывать куки. Если клиент – это браузер, то получив куки, он сохраняет их в *Cookie Storage*. В каждый последующий запрос к серверу браузер будет добавлять сохранённые значения в заголовке *Cookie*, если не подошла дата истечения срока годности и параметры *path* и *domain* соответствуют *URL* ресурса. Все куки передаются одной строкой, в которой разделяются точкой с запятой.



Ох, рано

Сессии и cookies

Таким образом, по наличию актуальной *cookie* в запросе приложение понимает, какой пользователь направил запрос и что этот пользователь аутентифицирован. При отсутствии cookie приложение попросит пройти аутентификацию ещё раз. Если пользователь выходит из приложения (делает *log out* или закрывает вкладку браузера), то *cookie* удаляются, а сервер закрывает сессию.

Подобный функционал используется не только для аутентификации. Например, хранение данных в корзине заказов при переходе между вкладками одного сайта или контроль действий, которые можно сделать один раз (например, проголосовать в онлайн-конкурсе) могут быть реализованы тем же способом.

WEBSITE: We use cookies to improve performance.

ME: Same.



Ох, рано

Сессии и cookies

Пример установки cookies:

```
@GetMapping(value = "/set-cookie")
public ResponseEntity<?> setCookie(HttpServletRequest response) throws IOException {
    Cookie cookie = new Cookie("data", "Come_to_the_dark_side"); // создаем объект Cookie,
    // в конструкторе указываем значения для name и value
    cookie.setPath("/"); // устанавливаем путь
    cookie.setMaxAge(86400); // устанавливаем время жизни куки
    response.addCookie(cookie); // добавляем Cookie в запрос
    response.setContentType("text/plain"); // устанавливаем тип ответа
    return ResponseEntity.ok();
}
```

Пример чтения cookies:

```
@GetMapping(value = "/get-cookie")
public ResponseEntity<?> readCookie(@CookieValue(value = "data") String data) {
    return ResponseEntity.ok().body(data);
}
```

Ох, рано

Сессии и cookies

Для управления сессией в поле контроллера или непосредственно в метод контроллера внедрите бин *HttpSession*. Для добавления/получения атрибутов сессии используются методы *setSessionAttribute()* и *getSessionAttribute()* соответственно. Прочие методы [здесь](#)

Можно переносить атрибуты модели в атрибуты сессии, если указать аннотацию *@SessionAttributes("att_name")* над классом контроллера. Тогда любой атрибут с именем *"att_name"*, попавший в *Model*, будет сохраняться в течение всей сессии.

Ещё один подход – создать бин со *Scope == SESSION* и использовать в контроллере его.

О сохранении сессии в БД читайте [здесь](#).



Задание

- 1 Откройте браузер и консоль браузера (обычно F12). Пройдите аутентификацию на каком-либо сайте. В ответе в консоли найдите заголовок Cookie или Set-Cookie. Попробуйте перейти на другую страницу сайта или обновить текущую страницу (проходить аутентификацию повторно не придётся). Очистите куки браузера и обновите страницу сайта.
- 2 Создайте приложение, которое при обращении к странице будет показывать, сколько раз пользователь обращался к этой странице (счётчик). Для проверки зайдите с двух разных браузеров, чтобы проверить работу сессий.

Ох, рано

Подключение Spring Security

Для подключения *Spring Security* в *Spring Boot* проект достаточно указать в pom.xml стартер *spring-boot-starter-security*. И всё будет работать «из коробки»: доступ ко всем эндпоинтам будет только через страницу авторизации. Однако, даже стартовая страница нашего приложения тогда не будет доступна без аутентификации. Чтобы пройти аутентификацию используйте логин user и пароль, сгенерированный Spring (см. в лог запуска приложения).

```
Using generated security password: 80078187-a100-435e-9210-fe031d9e77bc
```

```
This generated password is for development use only. Your security config
```



Ох, рано

Подключение Spring Security

Чтобы связать сущность пользователя с данными его аутентификации (например, *Basic*-аутентификацией), в сущности пользователя нужно создать соответствующие поля (*username* и *password*). Для удобства их можно создать как Embedded-поле.

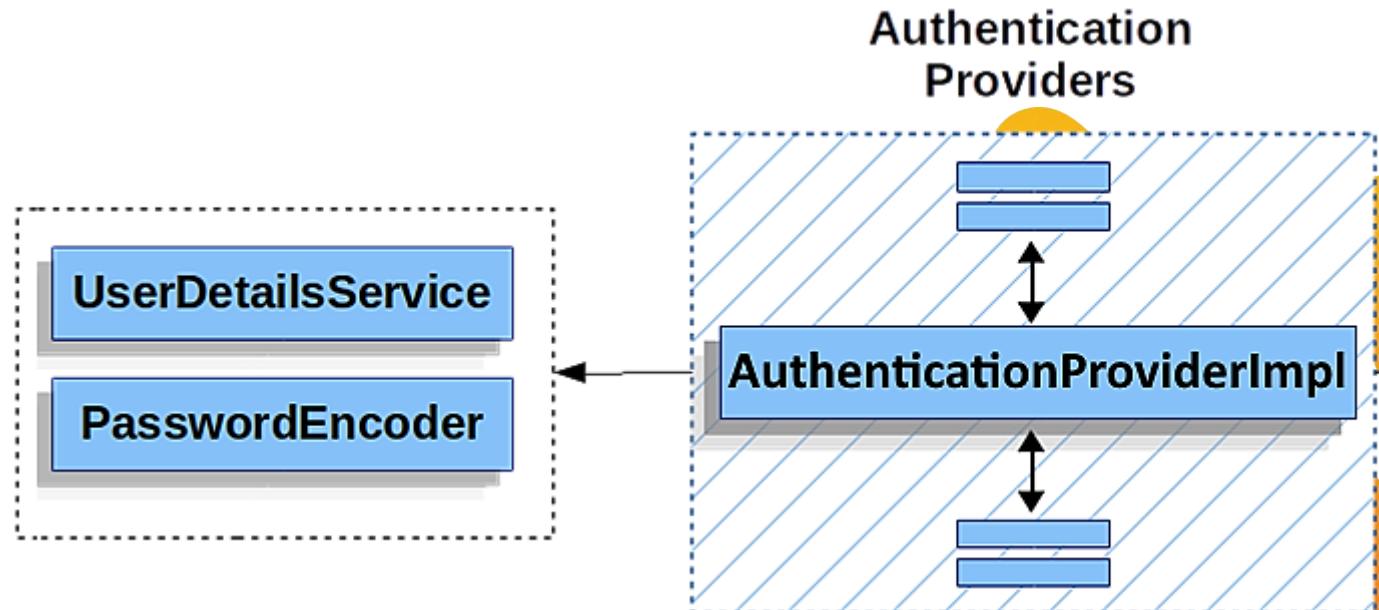
```
@Entity  
@Table(name = "users")  
public class User {  
  
    // другие поля, геттеры, сеттеры  
  
    @Embedded  
    private Credentials credentials;  
}
```

```
@Embeddable  
public class Credentials {  
    @Column(name = "username")  
    private String username;  
    @Column(name = "password")  
    private String password;  
    // геттеры, сеттеры, конструктор  
}
```

Ох, рано

Подключение Spring Security

AuthenticationProvider получает данные о пользователе из имплементации интерфейса **UserDetailsService**. Создавая свой сервис для работы с клиентами следует имплементировать этот интерфейс.

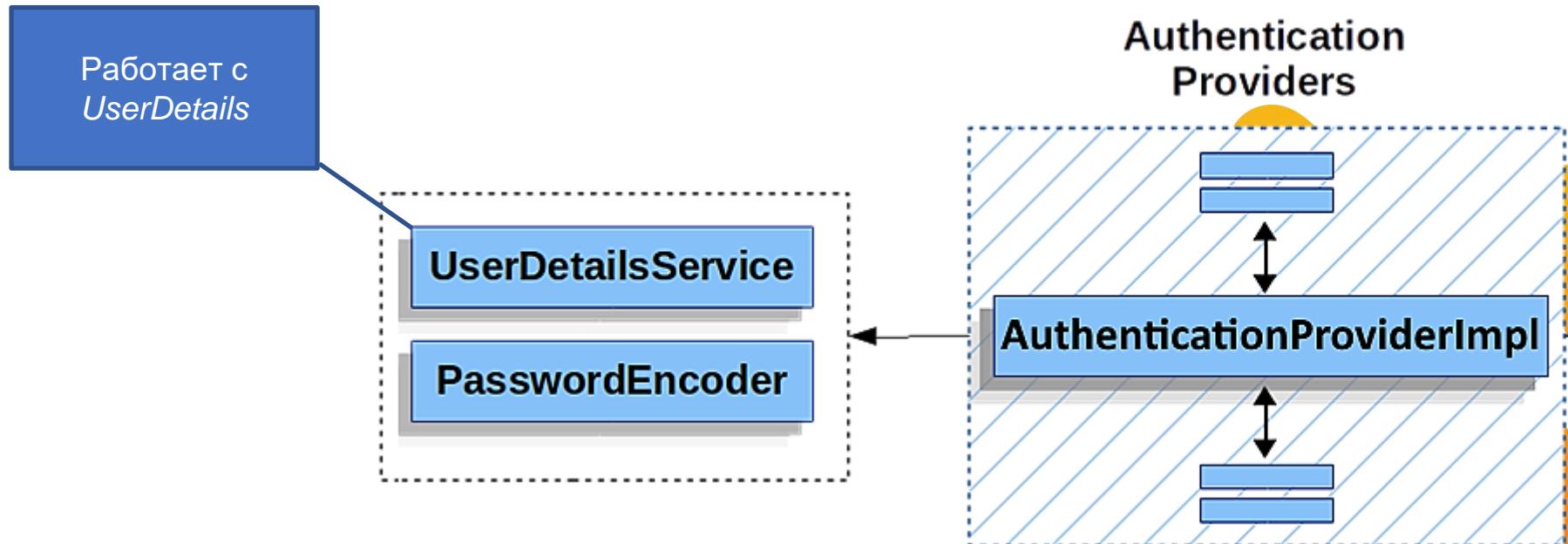


Ох, рано

Подключение Spring Security

UserDetailsService работает с сущностями, имплементирующими интерфейс *UserDetails*.

Чтобы не перегружать сущность *User* обязанностями обычно создают класс-обёртку для *User*, которая как раз и реализует *UserDetails*.



Ох, рано

Пример реализации AuthenticationProvider, UserDetailsService и UserDetails



SpringSecurityExample.zip

Ох, рано

SecurityFilterChain

В версиях Spring до 6 (*Spring Boot* до 3) в конфигурационном классе нужно было наследовать класс *WebSecurityConfigurerAdapter*, в переопределённых методах которого выполнялась настройка. Прежде всего в методе *configure()*.

В новых версиях достаточно создать конфигурационный бин *SecurityFilterChain*.

```
@Configuration @EnableWebSecurity
public class SecurityConfig {
    @Bean // В таком виде бин создаётся по умолчанию, если нет класса конфигурации
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http.authorizeHttpRequests(authz -> authz.anyRequest().authenticated()) // какие эндпоинты
            защищать: anyRequest() означает все запросы, а authenticated() означает доступны только
            аутентифицированным пользователям
            .formLogin(withDefaults()) // настройки формы аутентификации
            .httpBasic(withDefaults()) // настройки вида аутентификации
            .authenticationProvider(authProvider); // используем свой провайдер при необходимости
            .build();
    }
}
```

Ох, рано

SecurityFilterChain

SecurityFilterChain позволяет настроить эндпоинты *login* и *logout* без их явного создания в каком-либо контроллере. *Spring Security* задаёт поведение формы аутентификации по умолчанию:

- при неправильной аутентификации перенаправляет повторно на страницу аутентификации с указанием причины ошибки аутентификации.
- в случае успешной аутентификации перенаправляет на изначально запрашиваемый ресурс

```
.formLogin(withDefaults() // настройки формы аутентификации
.logout(logout -> logout.logoutUrl("/auth/logout").logoutSuccessUrl("/auth/login")) // действия для выхода
```

Для *logout* можно указать URL для эндпоинта и куда перенаправлять после *logout*.

Ох, рано

SecurityFilterChain

Для использования собственной формы *login* потребуется сама форма, а также настройка *SecurityFilterChain*.

```
@Bean
public SecurityFilterChain formLoginFilterChain(HttpSecurity http, AuthenticationManager manager) {
    return http
        // настройка правил аутентификации
        .authorizeHttpRequests(authorize ->
            authorize.requestMatchers("/auth/login", "/auth/registration", "/error").permitAll() // какие url открыть
                .anyRequest().authenticated() // остальные открыть только для аутентифицированных пользователей
        // Настройка своей формы аутентификации
        .formLogin(form -> form.loginPage("/auth/login") // страница аутентификации
            .loginProcessingUrl("/process_login") // сюда будет выполнен post-запрос из формы аутентификации.
            .defaultSuccessUrl("/view/profile", true) // перенаправляем сюда в случае успешной аутентификации
            .failureUrl("/auth/login?error") // если аутентификация не удалась, выдаём ту же страницу с параметром error
        )
        .logout(logout -> logout.logoutUrl("/auth/logout").logoutSuccessUrl("/auth/login")) // действия для выхода
    .build();
}
```

Ох, рано

Своя форма login

Для защиты от [csrf](#) используется настройка `csrf()` при создании бина `SecurityFilterChain`. По умолчанию она включена. Это требует от нашей `login`-формы содержать скрытое поле с `csrf.token`.

```
<!-- Names of fields username and password should be like in the example below -->
<form name="f" method="post" action="/process_login">
    <!-- When you added the field below Thymeleaf will add the same field in all your forms -->
    <!-- After that you can do logout only with post request => you need form -->
    <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}"/>
    <label for="username">Username</label>
    <input type="text" name="username" id="username" />
    <br>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" />
    <br>
    <input type="submit" value="Login"/>
    <div th:if="${param.error}" style="color: red">Invalid username or password</div>
</form>
```

Ох, рано

Создание DaoAuthenticationProvider



[SpringSecurityWithRolesExample.zip](#)

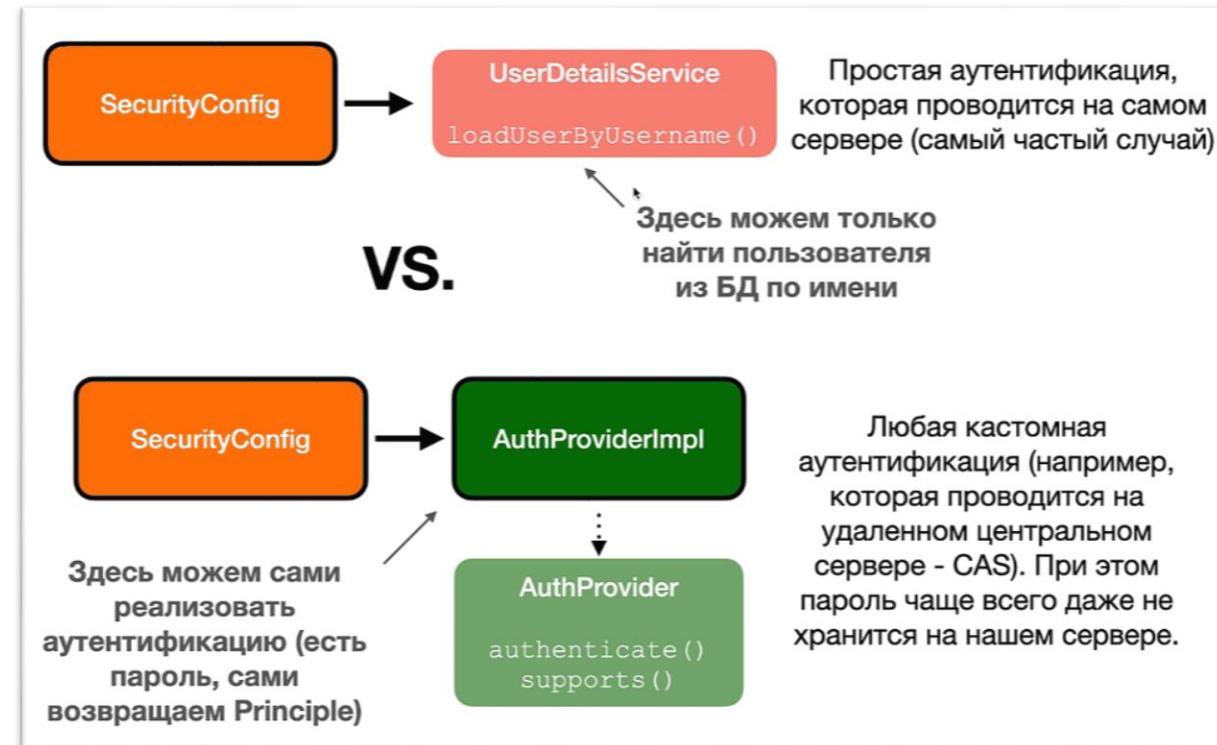


Ох, рано

Собственный AuthProvider

Basic-аутентификация в простейшем виде (с проверкой в БД приложения) не требует создания собственной имплементации *AuthProvider*.

Можно воспользоваться готовой имплементацией – классом *DaoAuthenticationProvider*



Ох, рано

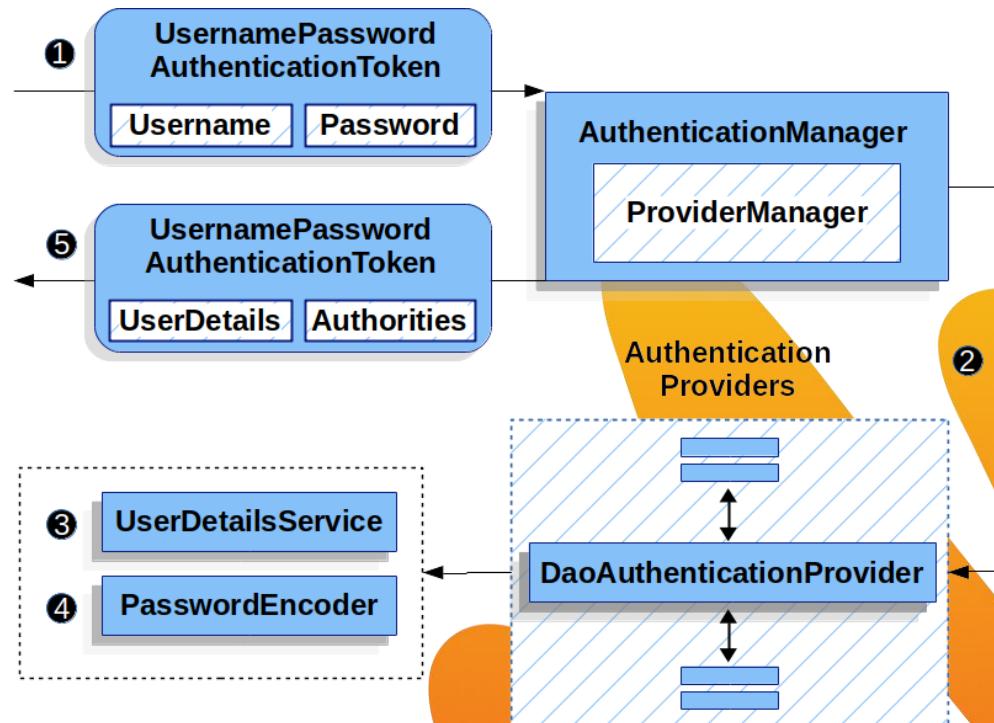
DaoAuthenticationProvider

DaoAuthenticationProvider – это реализация *AuthenticationProvider*, которая использует бины *UserDetailsService* и *PasswordEncoder* для аутентификации имени пользователя и пароля.

1. Фильтр передает

UsernamePasswordAuthenticationToken в *AuthenticationManager*, который реализован классом *ProviderManager*

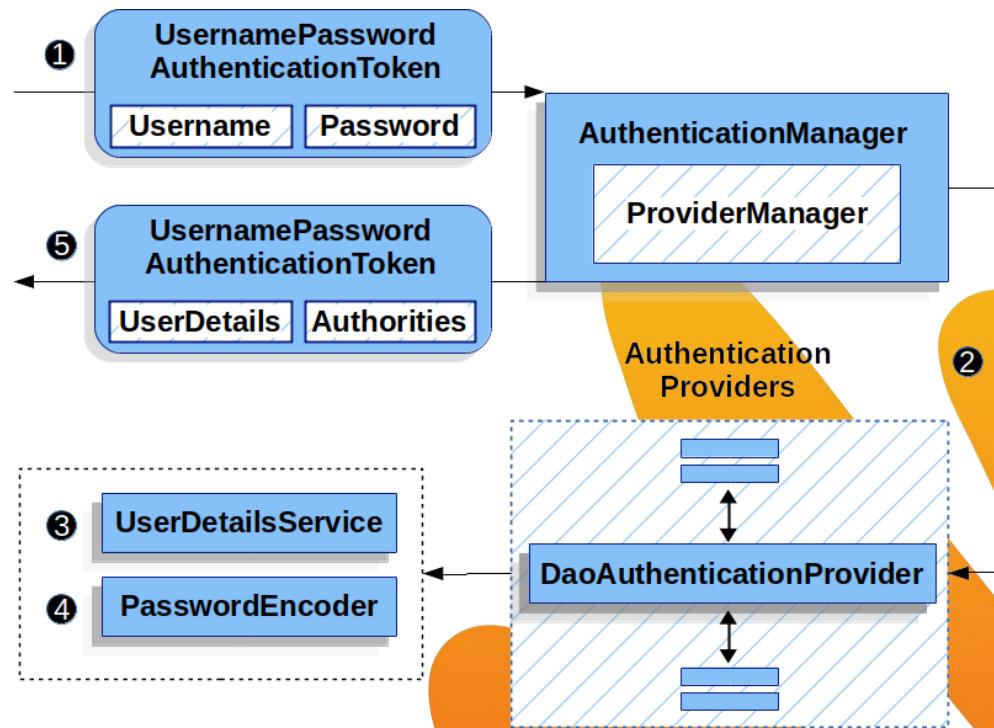
2. *ProviderManager* настроен на использование объекта типа *AuthenticationProvider*, имплементацией которого является *DaoAuthenticationProvider*



Ох, рано

DaoAuthenticationProvider

3. *DaoAuthenticationProvider* просматривает данные пользователя из *UserDetailsService*.
4. *DaoAuthenticationProvider* использует *PasswordEncoder* для декодирования и проверки пароля на пользовательских данных, полученных на предыдущем шаге.
5. При успешной аутентификации возвращаемая объект имеет тип *UsernamePasswordAuthenticationToken* и содержит *Principal* (данные пользователя), который является одной из реализаций *UserDetails*. *UsernamePasswordAuthenticationToken* устанавливается в *SecurityContextHolder*.



Ох, рано

DaoAuthenticationProvider

Чтобы подготовить *DaoAuthenticationProvider* нужно создать бин *UserDetailsService* или имплементировать собственный сервис, следующий этому интерфейсу.

Также нужно создать бин *PasswordEncoder* для декодирования и проверки пароля, хранимого в БД. В простейшем случае пароль может вообще не кодироваться.

```
@Bean
public AuthenticationManager authenticationManager(PasswordEncoder passwordEncoder) {
    DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
    authenticationProvider.setUserDetailsService(userDetailsService); // установили
    authenticationProvider.setPasswordEncoder(passwordEncoder);

    return new ProviderManager(authenticationProvider);
}

@Bean
public PasswordEncoder passwordEncoder() {
//    return new BCryptPasswordEncoder();
    return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance(); // без шифрования
}
```

Ох, рано

Создание DaoAuthenticationProvider



SpringSecurityWithRolesExample.zip



Ох, рано

Роли и разрешения

В ИБ выделяют разные подходы для разделения прав доступа к ресурсам. Наиболее популярные из них – использование ролей (*roles*) и разрешений (*authorities*).

Роли – это статусы, назначаемые пользователю. Например, *ROLE_USER*, *ROLE_MODERATOR*, *ROLE_ADMIN* и т.д. Каждой роли соответствует свой набор **разрешений** (получать данные, создавать/изменять/удалять сущности и т.д.), т.е. доступных ресурсов (эндпоинтов). Разрешения составляют перечень доступных пользователю действий.

Ох, рано

Роли и разрешения

Обычно одной роли соответствует набор разрешений, но для *Spring Security* большого отличия нет. В большей степени *Spring Security* присуще использование разрешений. При использовании методов *hasAnyAuthority()* или *hasAuthority()* необходимо указывать полное название разрешения. При использовании методов *hasAnyRole()* или *hasRole()* в конфигурации указываются роли без префикса *ROLE_*.

Если не указать *HttpMethod*, то правило безопасности будет применено ко всем методам.

```
@Bean
public SecurityFilterChain formLoginFilterChain(HttpSecurity http, AuthenticationManager manager) {
    return http.authorizeHttpRequests(authorize ->
        authorize.requestMatchers("/auth/login", "/auth/registration", "/error").permitAll()
            .requestMatchers(HttpMethod.GET, "/users/**").hasAnyAuthority(ROLE_READ, ROLE_WRITE)
            .requestMatchers(HttpMethod.POST, "/users/**", "view/register").hasRole(ADMIN)
            .anyRequest().authenticated() // для др. методов любой аутент-ый пользователь
    // ...
    .build();
}
```

Ох, рано

Роли и разрешения

Настройки ролей можно выполнить не только в классе конфигурации.

Для этого в классе конфигурации нужно указать аннотацию **@EnableMethodSecurity**. Это позволит использовать аннотации для контроля ролей:

@Secured - это устаревший параметр для авторизации вызовов методов. **@PreAuthorize** заменяет его и рекомендуется вместо него.

@RolesAllowed, **@PermitAll**, **@DenyAll** – аннотации стандарта *JSR-250*, которые тоже поддерживаются в *Spring Security*. Они помогают гибко настраивать роли. Для использования этих аннотаций требуется настроить аннотацию **@EnableMethodSecurity(jsr250Enabled = true)**. Это приведёт к созданию соответствующих перехватчиков (*interceptor*) фреймворком.

@PreAuthorize обладает большей выразительностью, чем аннотации *JSR-250* и поэтому рекомендуется в официальной документации.

Задание

Добавьте безопасность в приложение о меню.



2

Домашнее задание

Домашнее задание

Беречь себя и быть в безопасности



Полезные ссылки

- Статья об Oauth 2.0

<https://habr.com/ru/companies/vk/articles/115163/>

- Документация Spring Security

<https://docs.spring.io/spring-security/reference/servlet/architecture.html>

ЗАКЛЮЧЕНИЕ



Приложение А

Список уязвимостей аутентификации по паролю

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

Приложение А

Список уязвимостей аутентификации по паролю

- Веб-приложение использует уязвимую функцию восстановления пароля, которую можно использовать для получения несанкционированного доступа к другим учетным записям.
- Веб-приложение не требует повторной аутентификации пользователя для важных действий: смена пароля, изменения адреса доставки товаров и т. п.
- Веб-приложение создает session tokens таким образом, что они могут быть подобраны или предсказаны для других пользователей.
- Веб-приложение допускает передачу session tokens по незащищенному HTTP-соединению, либо в строке URL.
- Веб-приложение уязвимо для session fixation-атак (т. е. не заменяет session token при переходе анонимной сессии пользователя в аутентифицированную).

Приложение А

Список уязвимостей аутентификации по паролю

- Веб-приложение не устанавливает флаги HttpOnly и Secure для browser cookies, содержащих session tokens.
- Веб-приложение не уничтожает сессии пользователя после короткого периода неактивности либо не предоставляет функцию выхода из аутентифицированной сессии.