



borisyuzhakov

15 окт 2017 в 22:15

Пять простых шагов для понимания JSON Web Tokens (JWT)



5 мин



630К

Информационная безопасность*, Веб-разработка*, Программирование*

Из песочницы



Представляю вам мой довольно вольный перевод статьи *5 Easy Steps to Understanding JSON Web Tokens (JWT)*. В этой статье будет рассказано о том, что из себя представляют *JSON Web Tokens (JWT)* и с чем их едят. То есть какую роль они играют в проверке подлинности пользователя и обеспечении безопасности данных приложения.

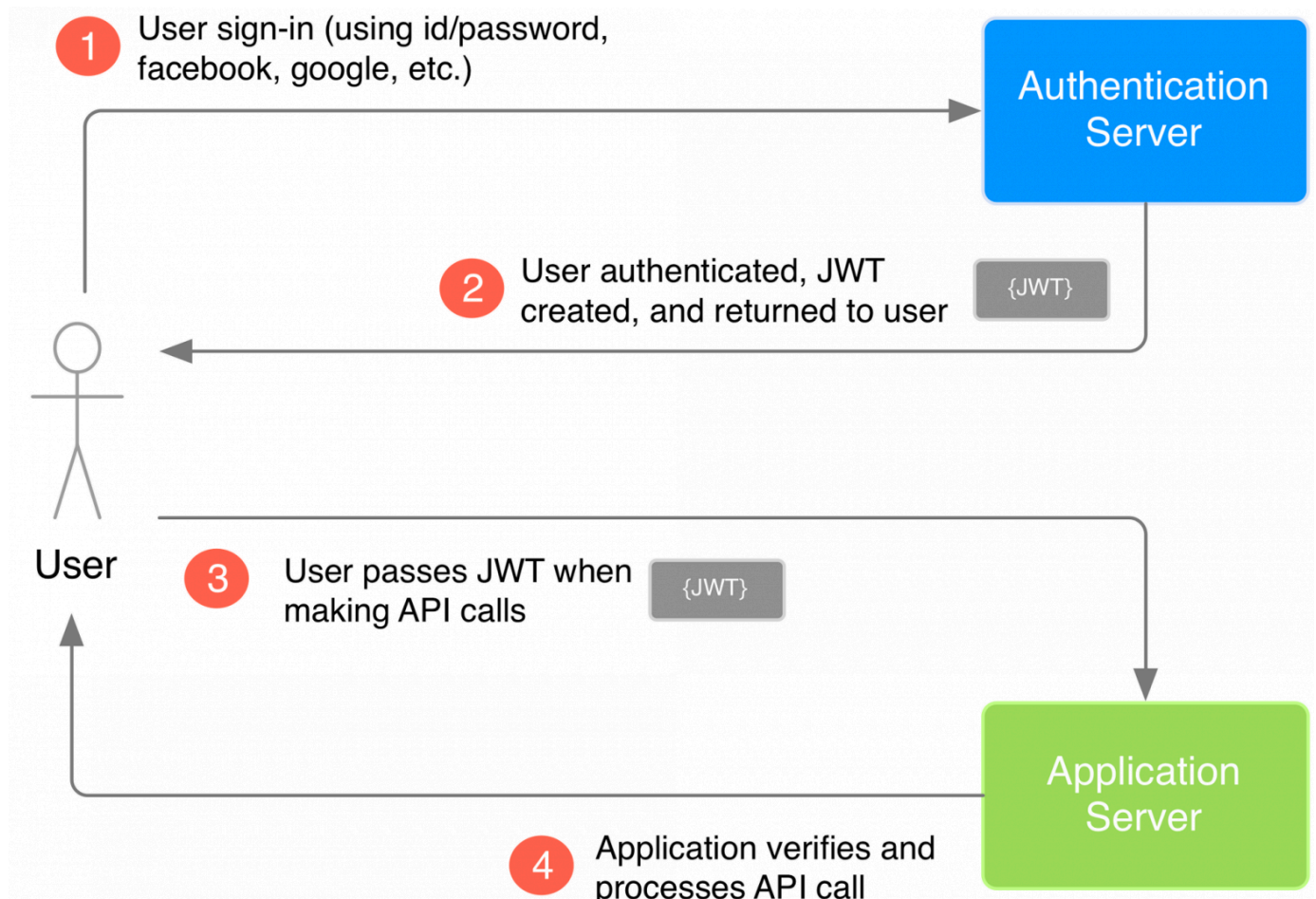
Для начала рассмотрим формальное определение.

JSON Web Token (JWT) — это JSON объект, который определен в открытом стандарте RFC 7519. Он считается одним из безопасных способов передачи информации между двумя участниками. Для его создания необходимо определить заголовок (header) с общей информацией по токену, полезные данные (payload), такие как id пользователя, его роль и т.д. и подписи (signature).

Кстати, правильно *JWT* произносится как /dʒɒt/

Простыми словами, *JWT* — это лишь строка в следующем формате `header.payload.signature`.

Предположим, что мы хотим зарегистрироваться на сайте. В нашем случае есть три участника — пользователь `user`, сервер приложения `application server` и сервер аутентификации `authentication server`. Сервер аутентификации будет обеспечивать пользователя токеном, с помощью которого он позднее сможет взаимодействовать с приложением.



Приложение использует *JWT* для проверки аутентификации пользователя следующим образом:

1. Сперва пользователь заходит на сервер аутентификации с помощью аутентификационного ключа (это может быть пара *логин/пароль*, либо *Facebook* ключ, либо *Google* ключ, либо ключ от другой учетки).
2. Затем сервер аутентификации создает *JWT* и отправляет его пользователю.
3. Когда пользователь делает запрос к API приложения, он добавляет к нему полученный ранее *JWT*.
4. Когда пользователь делает API запрос, приложение может проверить по переданному с запросом *JWT* является ли пользователь тем, за кого себя выдает. В этой схеме сервер приложения сконфигурирован так, что сможет проверить, является ли входящий *JWT* именно тем, что был создан сервером аутентификации (процесс проверки будет объяснен позже более детально).

Структура JWT

JWT состоит из трех частей: заголовок `header` , полезные данные `payload` и подпись `signature` . Давайте пройдемся по каждой из них.

Шаг 1. Создаем HEADER

Хедер *JWT* содержит информацию о том, как должна вычисляться *JWT* подпись. Хедер — это тоже *JSON* объект, который выглядит следующим образом:

```
header = { "alg": "HS256", "typ": "JWT" }
```

Поле `typ` не говорит нам ничего нового, только то, что это *JSON Web Token*.
Интереснее здесь будет поле `alg` , которое определяет алгоритм хеширования. Он будет использоваться при создании подписи. `HS256` — не что иное, как `HMAC-SHA256` , для его вычисления нужен лишь один секретный ключ (более подробно об этом в шаге 3). Еще может использоваться другой алгоритм `RS256` — в отличие от предыдущего, он является асимметричным и создает два ключа: публичный и приватный. С помощью приватного ключа создается подпись, а с помощью публичного только лишь проверяется подлинность подписи, поэтому нам не нужно беспокоиться о его безопасности.

Шаг 2. Создаем PAYLOAD

Payload — это полезные данные, которые хранятся внутри *JWT*. Эти данные также называют *JWT-claims* (заявки). В примере, который рассматриваем мы, сервер аутентификации создает *JWT* с информацией об *id* пользователя — **userId**.

```
payload = { "userId": "b08f86af-35da-48f2-8fab-cef3904660bd" }
```

Мы положили только одну *заявку* (*claim*) в *payload*. Вы можете положить столько *заявок*, сколько захотите. Существует список стандартных *заявок* для *JWT payload* — вот некоторые из них:

- *iss* (*issuer*) — определяет приложение, из которого отправляется токен.
- *sub* (*subject*) — определяет тему токена.
- *exp* (*expiration time*) — время жизни токена.

Эти поля могут быть полезными при создании *JWT*, но они не являются обязательными. Если хотите знать весь список доступных полей для *JWT*, можете заглянуть в [Wiki](#). Но стоит помнить, что чем больше передается информации, тем больший получится в итоге сам *JWT*. Обычно с этим не бывает проблем, но все-таки это может негативно сказаться на производительности и вызвать задержки во взаимодействии с сервером.

Шаг 3. Создаем SIGNATURE

Подпись вычисляется с использованием следующего псевдо-кода:

```
const SECRET_KEY = 'cAtwa1kkEy'  
const unsignedToken = base64urlEncode(header) + '.' + base64urlEncode(payload)  
const signature = HMAC-SHA256(unsignedToken, SECRET_KEY)
```

Алгоритм **base64url** кодирует хедер и payload, созданные на 1 и 2 шаге. Алгоритм соединяет закодированные строки через точку. Затем полученная строка хешируется алгоритмом, заданным в хедере на основе нашего секретного ключа.

```
// header eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9  
// payload eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ  
// signature -xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

Шаг 4. Теперь объединим все три JWT компонента вместе

Теперь, когда у нас есть все три составляющих, мы можем создать наш *JWT*. Это довольно просто, мы соединяем все полученные элементы в строку через точку.

```
const token = encodeBase64Url(header) + '.' + encodeBase64Url(payload) + '.' +  
// JWT Token  
// eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM
```

Вы можете попробовать создать свой собственный *JWT* на сайте jwt.io.

Вернемся к нашему примеру. Теперь сервер аутентификации может слать пользователю *JWT*.

Как JWT защищает наши данные?

Очень важно понимать, что использование *JWT* **НЕ** скрывает и не маскирует данные автоматически. Причина, почему *JWT* используются — это проверка, что отправленные данные были действительно отправлены авторизованным источником. Как было продемонстрировано выше, данные внутри *JWT* закодированы и подписаны, обратите внимание, это не одно и то же, что зашифрованы. Цель кодирования данных — преобразование структуры. Подписанные данные позволяют получателю данных проверить аутентификацию источника данных. Таким образом закодирование и подпись данных не защищает их. С другой стороны, главная цель шифрования — это защита данных от неавторизованного доступа. Для более детального объяснения различия между кодированием и шифрованием, а также о том, как работает хеширование, смотрите эту статью. Поскольку *JWT* только лишь закодирована и подписана, и поскольку *JWT* не зашифрована, *JWT* не гарантирует никакой безопасности для чувствительных (*sensitive*) данных.

Шаг 5. Проверка JWT

В нашем простом примере из 3 участников мы используем *JWT*, который подписан с помощью `HS256` алгоритма и только сервер аутентификации и сервер приложения знают секретный ключ. Сервер приложения получает секретный ключ от сервера аутентификации во время установки аутентификационных процессов. Поскольку приложение знает секретный ключ, когда пользователь делает API-запрос с приложенным к нему токеном, приложение может выполнить тот же алгоритм подписывания к *JWT*, что в шаге 3. Приложение может потом проверить эту подпись, сравнивая ее со своей собственной, вычисленной хешированием. Если подписи совпадают, значит *JWT* валидный, т.е. пришел от проверенного источника. Если подписи не совпадают, значит что-то пошло не так — возможно, это является признаком потенциальной атаки. Таким образом, проверяя *JWT*, приложение добавляет доверительный слой (*a layer of trust*) между собой и пользователем.

В заключение

Мы прошли по тому, что такое *JWT*, как они создаются и как валидируются, каким образом они могут быть использованы для установления доверительных отношений между пользователем и приложением. Но это лишь кусочек пазла большой темы авторизации и обеспечения защиты вашего приложения. Мы рассмотрели лишь основы, но без них невозможно двигаться дальше.

Что дальше?

Подумаем о безопасности и добавим `Refresh Token`. Смотрите следующую мою статью на эту тему.