**Authorization in APIs**

**Authorization** is how we determine what things a given user is allowed to do. So, for example, if you imagine an online learning platform, you might have different sets of permissions for different users. You might have some users that are students and can only see data in certain ways. A student user might only be able to see a grade mark, while another user who is a teacher might be able to actually modify the grade. There are many additional ways that you might want to authorize a user. If we continue thinking about an online learning platform, the teacher may only be able to modify grades for courses that they teach, and students should only be able to view their own grades and not the grades of their classmates. **Authorization is how we determine which things you have been given permission (are authorized) to do**

In an API, this can take the form of determining whether you are **authorized to use different endpoints or certain methods on an endpoint**. For instance, some users might not be allowed to use a certain endpoint at all, while in other cases a user may be allowed to use an endpoint to GET data, but not to DELETE or modify the data. There may also be cases where an API might return different information depending on the kind of user that you are. An API call that returns data about a user's grades might return information about the actual grade along with some public feedback if you call it as a student, but it might return some additional private comments if you call it as a teacher.

There might be a case where an API blocked direct access to certain data for a type of user. However, as that same user, I could see the data if I viewed an object that was a hierarchical parent of the object I wasn't supposed to see. There are often many different paths to the same data, and you want to think carefully about what those paths are when you are testing that the authorization is working correctly.

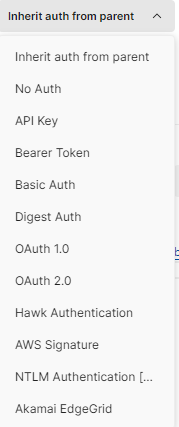
**Authentication in APIs**

**Authentication** is about determining whether you really are that kind of user. In other words, are you who you say you are?

If we take this back to the idea of API authorization and authentication, authentication involves putting in a password or providing a key that proves that you are who you say you are. Once you have done that, authorization determines which data you can see and interact with

**Types of Authorization**

POSTMAN uses Authorization as a term for both Authorization and Authentication



* **Inherit auth from parent option**. This is a powerful option that makes it much easier to manage an API. Usually when an API has authorization enabled, it will be required for every endpoint in the API. This means that any time you want to access a request in the API you will need to specify auth options. However, you aren't trying to test the authorization itself; you merely need to enter it so that you can do some other testing. In this case, it makes a lot of sense to only specify the login information in one place rather than repeating it for each request. Postman enables this by allowing you to specify authorization at the folder or collection level. If you click on the **View more actions** menu beside the collection in which your request is located and choose the **Edit** option, you will see that there is an **Authorization** tab. You can set up authorization options here, and then for any request in that collection, you can choose the **Inherit auth from parent option** to use the values you entered in the collection.

Sometimes in API testing, we want to check that different users are correctly authorized and so we might want to use different credentials for some of the API calls. To do this in Postman, you could **create a folder inside your collection for each user**, and in the options for that folder, you can specify authorization options that can be inherited by the requests in the folder. These options are set on a folder in essentially the same way they are on a collection

* **Basic Auth** option is used for APIs that require you to specify a username and password in order to use the API. It is probably the simplest form of API authorization, but it isn't one that is generally recommended or used in production systems. The reality is that Basic Auth is just a little bit too, well, basic for most uses. It is too easy to hack Basic Auth and so it is rarely used in production.

If you click on the **Headers** tab, you will see that one of the headers Postman has automatically added to this request is the Authorization header. The value for this header starts with the keyword **Basic** and this is how the server knows that you are using the basic authorization for this request. The rest of the value for the authorization header represents a **base64** encoding of the username and password that you put in.

It is important to note here that **base64 encoding is not secure**. If you copy the characters from this and go to https://www.base64decode.org/ and decode it, you will see that it is trivial to get back the username and password from this request. Since base64 encoding is not encryption, it is important to be careful about how you use it. If you are not using a secure (https) connection, this kind of encoding is open to man-in-the-middle attacks and so you want to be careful about using it. Most APIs don't use Basic Auth and instead rely on more complex authorization schemes.

* **Bearer tokens** are a common way to set up API authorization. They have some advantages over Basic Auth, since they can contain within them not only the concept of authentication, but also authorization

It’s required to perform log in request to Log in endpoint. Then the response would have Access Token, which we then can insert into Bearer token authorization.

This provides several advantages over basic authentication since in addition to authenticating that someone is who they say they are, with this token you can also determine what actions they are authorized to use. You can also change or revoke what things the bearer of the token can do and so if the token ever is compromised, it is easier to mitigate the damage. Due to this, these kinds of authorization tokens are very common

If you look at the Headers tab for that request, you will see that Postman has added an Authorization header for you. You can also see that the value for this token starts with Bearer, which lets the server know that you are using an API token to authorize your request. Bearer tokens are an example of a larger class of authorization methods known as API keys. Bearer API keys have become so common that Postman includes a separate way to set them up, but there are other ways to use **API keys as well.**

* **API keys** can be used in a variety of different ways. We can add API key to the **header**, **query params** or **body**

API keys can also be added to the Authorization header, and this is usually done with bearer tokens, which Postman has separated into its own category of authorization since it is so common.

Aside from using API keys as bearer tokens, one of the more common ways to pass them through the API, is still in the headers. Rather than passing it in with the authorization header though, it will often be passed as its own header. The exact way to do this will vary from API to API and you will need to look at the documentation to figure out exactly how it works. A common header used for this is the x-api-key header, although others can also be used.

* Cloud computing is fully mainstream, and many companies now use cloud computing platforms or are trying to move towards them. In many ways, Amazon has led the charge when it comes to cloud computing and the **Amazon Web Services (AWS)**

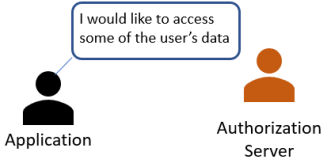
A lot of cloud computing systems and resources can be interacted with via APIs, which need to be authorized. Postman provides an AWS Signature authorization option to help with authorizing AWS APIs

In many ways, the AWS Signature option is just another way to specify an API key. You specify the **AccessKey** and the **SecretKey**, which are a lot like the **key** and the **value** for the API keys option. You can also specify whether the data gets passed in as a header or a query

There are a few advanced options that let you set things such as the **AWS Region**, the **Service Name**, and the **Session Token**. Postman uses these options to automatically add the headers and parameters that you need to your AWS calls

* **OAuth2.0** The first thing to know is that the **Oauth** specification, while of course being about authorization, is primarily about the delegation of that authorization

Imagine you want to play a game that needs access to some of your user data from Facebook. You ask the authorization server for access to the data that it needs

****

The authorization server will prompt the user to see if the application should be allowed to have the access it wants. If you approve that access, the authorization server will then give the application a token that will give it access to the data it has requested access to

****

An authorization server gives the application a token that it can then use to access the data that it needs from the resource server

OAuth workflow involves a couple of steps and includes some complications, but it has a few benefits as well. It allows a third-party application to access some of your data without that application needing to be trusted with your password. It also has a lot of the same benefits as a bearer token in that you can change or revoke access if anything goes wrong in the future.

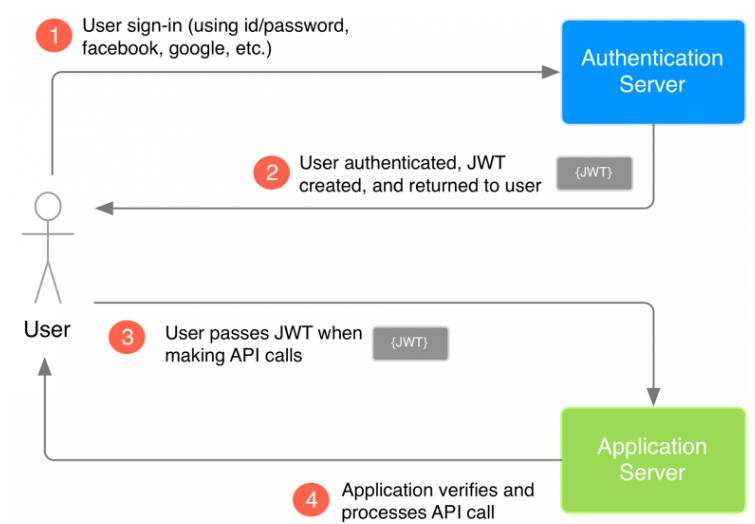
<https://www.youtube.com/watch?v=vQldMjSJ6-w&ab_channel=JavaScript.Ninja>

* **JWT** - 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).

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

Необходимость создания JWT вышла из того, что сервер аутентификации и сервер API находятся не на 1 сервере, и если передавать аутентификацию через Cookie, то данные аутентификации выставятся автоматически для сервера аутентификации, а нам нужно передавать данные на другой сервер

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.



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

**Структура JWT**

JWT состоит из трех частей: заголовок **header**, полезные данные **payload** и подпись **signature**

**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 ассиметричным 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. 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 использование 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) + '.' + encodeBase64Url(signature)

// JWT Token

// eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM

**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 тоже, 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