官网: <http://jwt.io/>

介绍: <http://jwt.io/introduction/>

漏洞: <https://auth0.com/blog/2015/03/31/critical-vulnerabilities-in-json-web-token-libraries/>

## JSON Web Token 的结构

一个 JWT token 看起来是这样的:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjEzODY4OTkxMzEsImlzcyI6ImppcmE6MTU0ODk1OTUiLCJxc2giOiI4MDYzZmY0Y2ExZTQxZGY3YmM5MGM4YWI2ZDBmNjIwN2Q0OTFjZjZkYWQ3YzY2ZWE3OTdiNDYxNGI3MTkyMmU5IiwiaWF0IjoxMzg2ODk4OTUxfQ.uKqU9dTB6gKwG6jQCuXYAiMNdfNRw98Hw\_IWuA5MaMo

可以简化为下面这样的结构:

base64url\_encode(Header) + '.' + base64url\_encode(Claims) + '.' + base64url\_encode(Signature)

## Header

Header 包含了一些元数据，至少会表明 token 类型以及 签名方法。比如

{

"typ" : "JWT",

"alg" : "HS256"

}

* type: 必需。token 类型，JWT 表示是 JSON Web Token.
* alg: 必需。token 所使用的签名算法，可用的值在 [这里](http://tools.ietf.org/html/rfc7518#section-3.1) 有规定。

## Claims (Payload)

Claims 部分包含了一些跟这个 token 有关的重要信息。 JWT 标准规定了一些字段，下面节选一些字段:

* iss: The issuer of the token，token 是给谁的
* sub: The subject of the token，token 主题
* exp: Expiration Time。 token 过期时间，Unix 时间戳格式
* iat: Issued At。 token 创建时间， Unix 时间戳格式
* jti: JWT ID。针对当前 token 的唯一标识

除了规定的字段外，可以包含其他任何 JSON 兼容的字段。

Payload 示例:

{

"iss": "mozillazg.com",

"exp": 1435055117,

"user\_id": 1,

"foo": "bar"

}

## Signature

JWT 标准遵照 JSON Web Signature (JWS) 标准来生成签名。签名主要用于验证 token 是否有效，是否被篡改。 签名时可以 这些算法进行签名，比如 HMAC SHA-256:

content = base64url\_encode(Header) + '.' + base64url\_encode(Claims)

signature = hmacsha256.hash(content)

**说到这里有一点需要特别注意的是，默认情况下，JWT 中信息都是明文的，即 Claims 的内容并没有 被加密，可以通过 base64url\_decode(text) 的方式解码得到 Claims** 。 所以，不要在 Claims 里包含敏感信息，如果一定要包含敏感信息的话，记得先将 Claims 的内容进行加密（比如，使用 JSON Web Encryption (JWE) 标准进行加密） 然后在进行 base64url\_encode 操作。

## [Introduction](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#introduction)

The API model has been used a great amount recently in applications. This has come about because applications can’t just rely on their own data anymore, for a project to fully see its potential, it must be able to have third-party applications, intermingle with other applications, and have its data easily accessilbe by developers.

Think of how Facebook provides an API to grab its data (as long as you are authenticated of course). Facebook also allows third-party applications and other services to access its data. This is all done through an API.

Now when we talk about building our own APIs, there’s always going to be the topic of **how to secure our own API**. We’ve talked a bit on [token based authentication](https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication), and built our own [RESTful Node.js API](https://scotch.io/tutorials/javascript/build-a-restful-api-using-node-and-express-4).

Today we’ll be looking at a standard (JSON Web Tokens) and how to create them.

## [What are JSON Web Tokens?](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#what-are-json-web-tokens?)

JSON Web Tokens (JWT), pronounced “jot”, are a standard since the information they carry is transmitted via JSON. We can read more about the[draft](http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html), but that explanation isn’t the most pretty to look at.

**JSON Web Tokens work across different programming languages**: JWTs work in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript, and Haskell. So you can see that these can be used in many different scenarios.

**JWTs are self-contained**: They will carry all the information necessary within itself. This means that a JWT will be able to transmit basic information about itself, a payload (usually user information), and a signature.

**JWTs can be passed around easily**: Since JWTs are self-contained, they are perfectly used inside an HTTP header when authenticating an API. You can also pass it through the URL.

## [What does a JWT look like?](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#what-does-a-jwt-look-like?)

A JWT is easy to identify. It is three strings separated by .

For example:

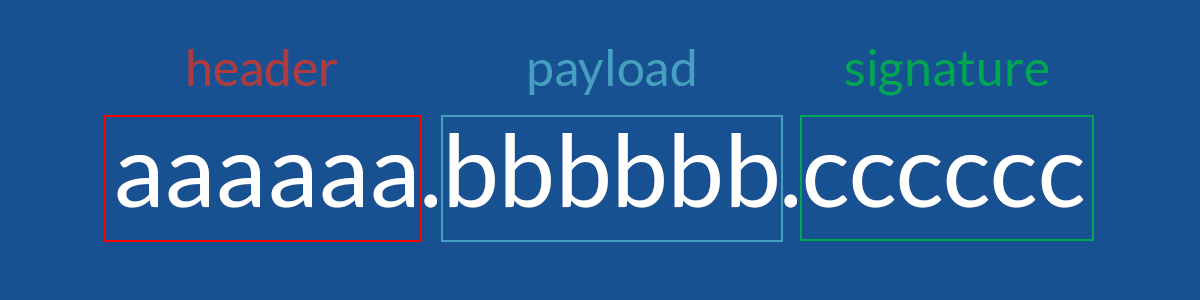
aaaaaaaaaa.bbbbbbbbbbb.cccccccccccc

Let’s break down the **3 parts** and see what each contains.

## [Breaking Down a JSON Web Token](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#breaking-down-a-json-web-token)

Since there are 3 parts separated by a ., each section is created differently. We have the 3 parts which are:

* header
* payload
* signature

[](https://cask.scotch.io/2014/11/json-web-token-overview1.png)

## [Header](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#header)

The header carries 2 parts:

* declaring the type, which is JWT
* the hashing algorithm to use (HMAC SHA256 in this case)

Here’s an example:

{

"typ": "JWT",

"alg": "HS256"

}

Now once this is base64encode, we have the first part of our JSON web token!

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

## [Payload](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#payload)

The payload will carry the bulk of our JWT, also called the [JWT Claims](http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html#RegisteredClaimName). This is where we will put the information that we want to transmit and other information about our token.

There are multiple claims that we can provide. This includes registered claim names, public claim names, and private claim names.

### Registered Claims

Claims that are not mandatory whose names are reserved for us. These include:

* iss: The issuer of the token
* sub: The subject of the token
* aud: The audience of the token
* exp: This will probably be the registered claim most often used. This will define the expiration in NumericDate value. The expiration MUST be after the current date/time.
* nbf: Defines the time before which the JWT MUST NOT be accepted for processing
* iat: The time the JWT was issued. Can be used to determine the age of the JWT
* jti: Unique identifier for the JWT. Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.

### Public Claims

These are the claims that we create ourselves like user name, information, and other important information.

### Private Claims

A producer and consumer may agree to use claim names that are private. These are subject to collision, so use them with caution.

## [Example Payload](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#example-payload)

Our example payload has two registered claims (iss, and exp) and two public claims (name, admin).

{

"iss": "scotch.io",

"exp": 1300819380,

"name": "Chris Sevilleja",

"admin": true

}

This will encode to:

eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0

That will be the second part of our JSON Web Token.

## [Signature](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#signature)

The third and final part of our JSON Web Token is going to be the signature. This signature is made up of a hash of the following components:

* the header
* the payload
* secret

This is how we get the third part of the JWT:

var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(payload);

HMACSHA256(encodedString, 'secret');

The secret is the signature held by the server. This is the way that our server will be able to verify existing tokens and sign new ones.

This gives us the final part of our JWT.

03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

Now we have our full JSON Web Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0.03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

[Auth0](https://auth0.com/) has created a [great site](http://jwt.io/) to go through and test out how JWTs are made. You can see as you change the content on the fly, you are able to see the JWT get updated immediately. Auth0 provides great tools and they also maintain the [jsonwebtoken](https://github.com/auth0/node-jsonwebtoken) Node package to handle creating and verifying JWTs in Node.

## [Conclusion](https://scotch.io/tutorials/the-anatomy-of-a-json-web-token#conclusion)

The JSON Web Token standard can be used across multiple languages and is quickly and easily interchangeable.

You can use the token in a URL, POST parameter, or an HTTP header. The versatility of the JSON Web Token let’s us authenticate an API quickly and easily by passing information through the token.

For a full code example on how to authenticate a Node API using JWTs, check out our book: [MEAN Machine](https://leanpub.com/mean-machine).

# JSON Web Token Tutorial

With the rising popularity of single page applications, mobile applications, and RESTful API services, the way[web developers](http://www.toptal.com/web) write back-end code has changed significantly. With technologies like AngularJS and BackboneJS, we are no longer spending much time building markup, instead we are building APIs that our front-end applications consume. Our back-end is more about business logic and data, while presentation logic is moved exclusively to the front-end or mobile applications. These changes have led to new ways of implementing authentication in modern applications.

Authentication is one of the most important parts of any web application. For decades, cookies and server-based authentication were the easiest solution. However, handling authentication in modern Mobile and Single Page Applications can be tricky, and demand a better approach. The best known solutions to authentication problems for APIs are the [OAuth 2.0](https://tools.ietf.org/html/rfc6749) and the [JSON Web Token](http://jwt.io/) (JWT).

## What is a JSON Web Token?

A JSON Web Token, or [JWT](https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32), is used to send information that can be verified and trusted by means of a digital signature. It comprises a compact and URL-safe JSON object, which is cryptographically signed to verify its authenticity, and which can also be encrypted if the payload contains sensitive information.

Because of it’s compact structure, JWT is usually used in HTTP Authorization headers or URL query parameters.

## Structure of a JSON Web Token

A JWT is represented as a sequence of [base64url](http://en.wikipedia.org/wiki/Base64) encoded values that are separated by period characters.



JSON Web Token example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.

yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK\_ZXw

### Header

The header contains the metadata for the token and it minimally contains the type of signature and the encryption algorithm.

**Example Header**

{

“alg”: “HS256”,

“typ”: “JWT”

}

This JWT Header declares that the encoded object is a JSON Web Token, and that it is signed using the HMAC SHA-256 algorithm.

Once this is base64 encoded, we have the first part of our JWT.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

### Payload (Claims)

In the context of JWT, a claim can be defined as a statement about an entity (typically, the user), as well as additional meta data about the token itself. The claim contains the information we want to transmit, and that the server can use to properly handle authentication. There are multiple claims we can provide; these include registered claim names, public claim names and private claim names.

**Registered Claims**

These are the claims that are registered in the [IANA JSON Web Token Claims registry](https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32#section-10.1). These claims are not intended to be mandatory but rather to provide a starting point for a set of useful, interoperable claims.

These include:

* **iss**: The issuer of the token
* **sub**: The subject of the token
* **aud**: The audience of the token
* **exp**: Token expiration time defined in Unix time
* **nbf**: “Not before” time that identifies the time before which the JWT must not be accepted for processing
* **iat**: “Issued at” time, in Unix time, at which the token was issued
* **jti**: JWT ID claim provides a unique identifier for the JWT

**Public Claims**

Public claims need to have collision-resistant names. By making the name a [URI](http://en.wikipedia.org/wiki/Uniform_resource_identifier) or [URN](http://en.wikipedia.org/wiki/Uniform_resource_name) naming collisions are avoided for JWTs where the sender and receiver are not part of a closed network.

An example of a public claim name could be: http://toptal.com/jwt\_claims/is\_admin, and the best practice is to place a file at that location describing the claim so that it can be dereferenced for documentation.

**Private Claims**

Private claim-names may be used in places where JWTs are only exchanged in a closed environment between known systems, such as inside an enterprise. These are claims that we can define ourselves, like user IDs, user roles, or any other information.

Using claim-names that might have conflicting semantic meanings outside of a closed or private system are subject to collision, so use them with caution.

It is important to note that we want to keep a web token as small as possible, so use only necessary data inside public and private claims.

**Example Payload**

{

“iss”: “toptal.com”,

“exp”: 1426420800,

“http://toptal.com/jwt\_claims/is\_admin”: true,

“company”: “Toptal”,

“awesome”: true

}

This example payload has two registered claims, one public claim and two private claims. Once it is base64 encoded, we have the second part of our JWT.

eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0

### Signature

The JWT standard follows the JSON Web Signature (JWS) specification to generate the final signed token. It is generated by combining the encoded JWT Header and the encoded JWT Payload, and signing it using a strong encryption algorithm, such as HMAC SHA-256. The signature’s secret key is held by the server so it will be able to verify existing tokens and sign new ones.

$encodedContent = base64UrlEncode(header) + “.” + base64UrlEncode(payload);

$signature = hashHmacSHA256($encodedContent);

This gives us the final part of our JWT.

yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK\_ZXw

## Security and Encryption with JWT

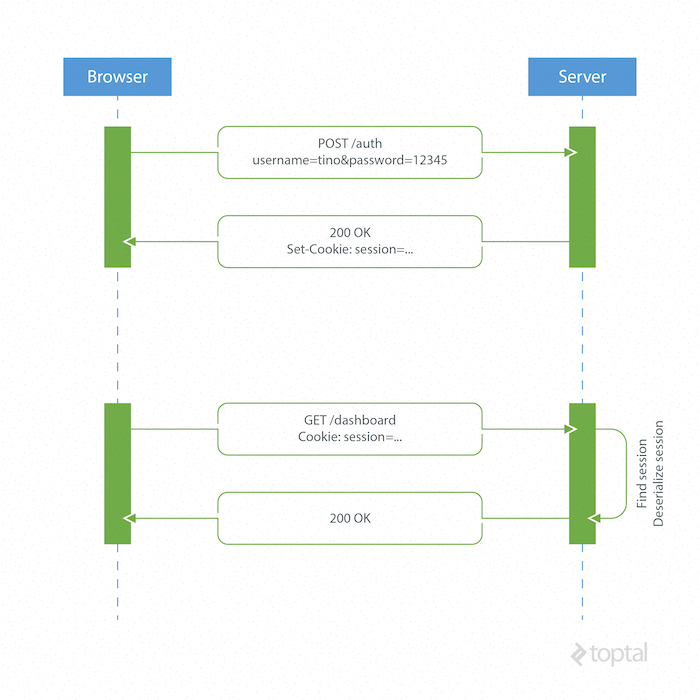
It is critical to use TLS/SSL in conjunction with JWT, to prevent man-in-the-middle attacks. In most cases, this will be sufficient to encrypt the JWT payload if it contains sensitive information. However, if we want to add an additional layer of protection, we can encrypt the JWT payload itself using the [JSON Web Encryption](http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-40) (JWE) specification.

Of course, if we want to avoid the additional overhead of using JWE, another option is to simply keep sensitive information in our database, and use our token for additional API calls to the server whenever we need to access sensitive data.

## Why the need for Web Tokens?

Before we can see all the benefits of using token authentication, we have to look at the way authentication has been done in the past.

### Server-Based Authentication



Because the HTTP protocol is stateless, there needs to be a mechanism for storing user information and a way to authenticate the user on every subsequent request after login. Most websites use cookies for storing user’s session ID.

**How it Works**

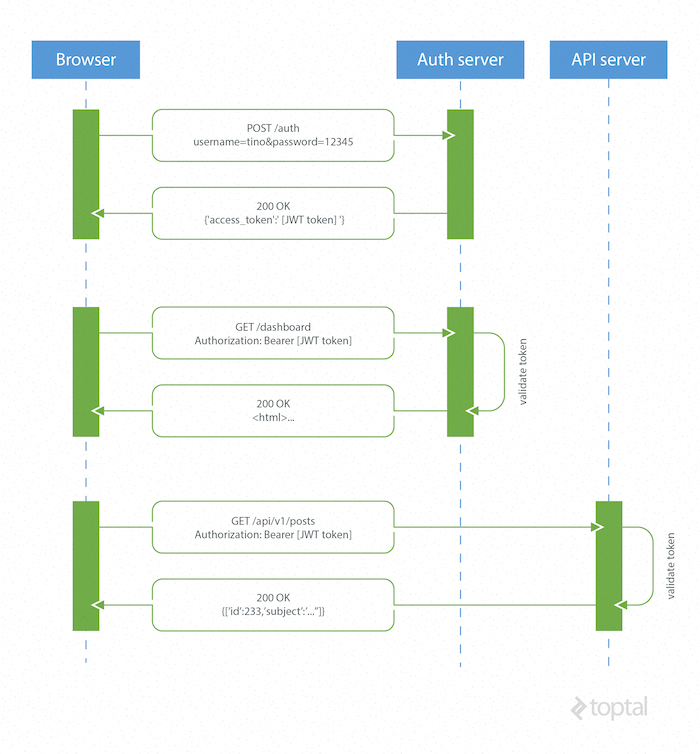
The browser makes a POST request to the server that contains the user’s identification and password. The server responds with a cookie, which is set on the user’s browser, and includes a session ID to identify the user.

On every subsequent request, the server needs to find that session and deserialize it, because user data is stored on the server.

### Drawbacks of Server-Based Authentication

* **Hard to scale**: The server needs to create a session for a user and persist it somewhere on the server. This can be done in memory or in a database. If we have a distributed system, we have to make sure that we use a separate session storage that is not coupled to the application server.
* **Cross-origin request sharing (CORS)**: When using AJAX calls to fetch a resource from another domain ([cross-origin](http://en.wikipedia.org/wiki/Cross-origin_resource_sharing)) we could run into problems with forbidden requests because, by default, HTTP requests don’t include cookies on cross-origin requests.
* **Coupling with the web framework**: When using server-based authentication we are tied to our framework’s authentication scheme. It is really hard, or even impossible, to share session data between different web frameworks written in different programming languages.

### Token-Based Authentication



Token based authentication is stateless, so there is no need to store user information in the session. This gives us the ability to scale our application without worrying where the user has logged in. We can easily use the same token for fetching a secure resource from a domain other than the one we are logged in to.

**How JSON Web Tokens Work**

A browser or mobile client makes a request to the authentication server containing user login information. The authentication server generates a new JWT access token and returns it to the client. On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it’s valid, returns the secure resource to the client.

The authentication server can sign the token using any secure signature method. For example, a symmetric key algorithm such as HMAC SHA-256 can be used if there is a secure channel to share the secret key among all parties. Alternatively, an asymmetric, public-key system, such as RSA, can be used as well, eliminating the need for further key-sharing.

### Advantages of Token-Based Authentication

**Stateless, easier to scale**: The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.

**Reusability**: We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user. It is easy to build an application that shares permissions with another application.

**Security**: Since we are not using cookies, we don’t have to protect against [cross-site request forgery](http://en.wikipedia.org/wiki/Cross-site_request_forgery) (CSRF) attacks. We should still encrypt our tokens using JWE if we have to put any sensitive information in them, and transmit our tokens over HTTPS to prevent man-in-the-middle attacks.

**Performance**: There is no server side lookup to find and deserialize the session on each request. The only thing we have to do is calculate the HMAC SHA-256 to validate the token and parse its content.