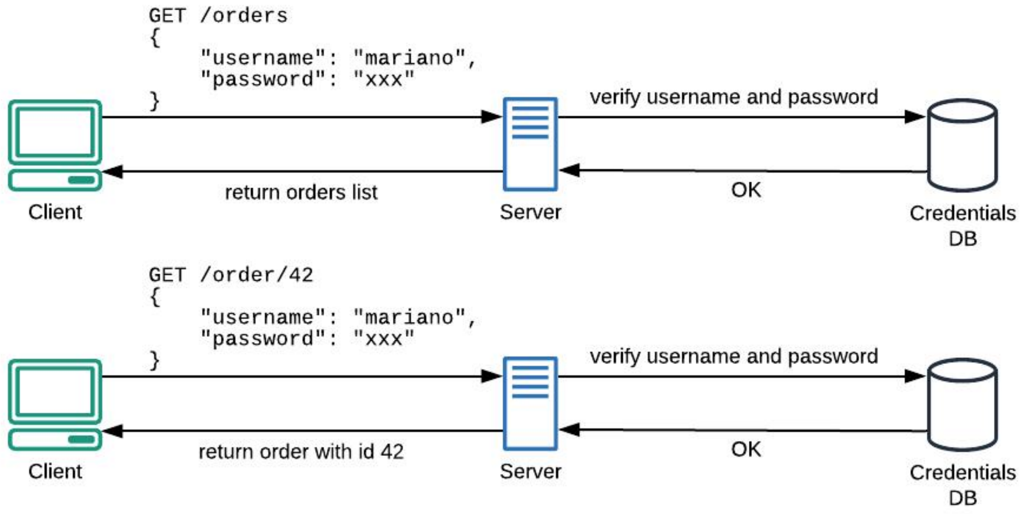
Spring Security

How to restrict access to a resource to the authorized users only

Once upon a time

Suppose you have a [REST API]() (e.g. GET /orders) and you want to restrict access to the authorized users only.
In the most naïve approach, the API would ask for a username and password; then it will be searched into a database if those credentials really exists. We check for *authenticity*. Finally, it will be checked if the *authenticated* user is also *authorized* to perform that request. If both checks passes the real API will be executed. It seems logical.

The HTTP protocol is *stateless*, that means a new request (e.g. GET /order/42) won't know anything about the previous one, **so we need to re-authenticate for each new reque**


```
GET /orders
{
    "username": "mariano",
    "password": "xxx"
}
```

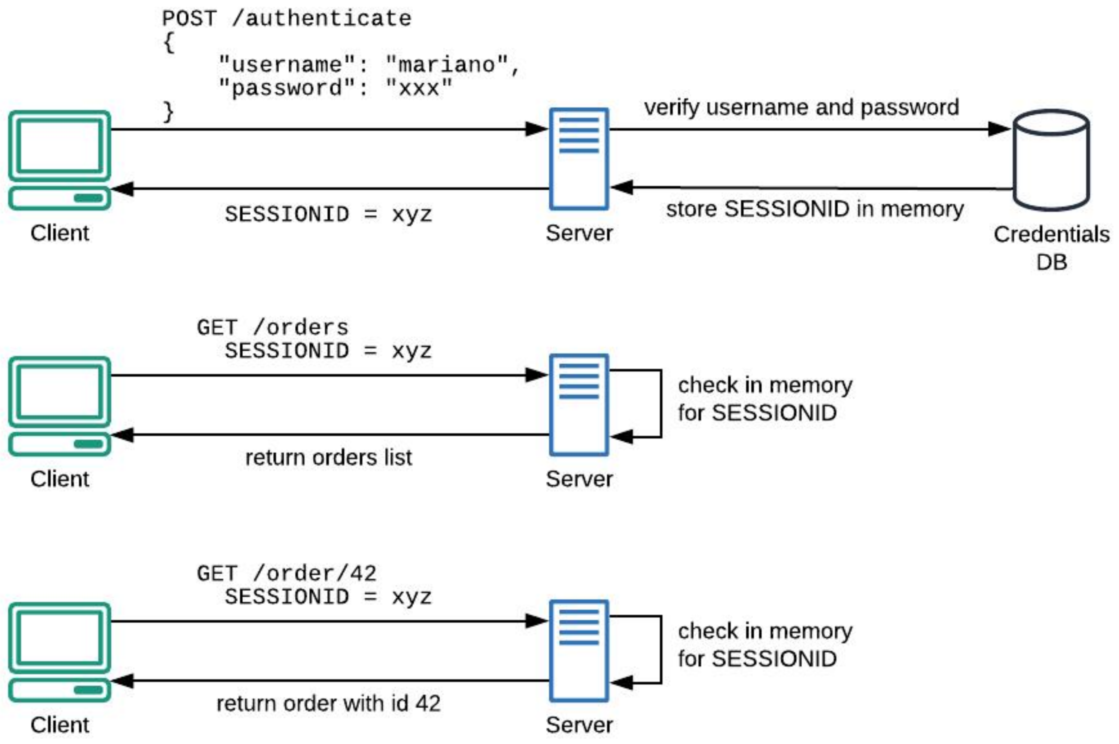
```
GET /order/42
{
    "username": "mariano",
    "password": "xxx"
}
```

Server Side Sessions (SSS)

The traditional way of dealing with this is the use of *Server Side Sessions* (SSS).

In this scenario, we first check for username and password; if they are authentic the server will save a *session id* in memory and return it to the client. From now on, client will just need to send its *session id* to be recognized

Using SSS, we reduce the number of authentications towards the Credentials database



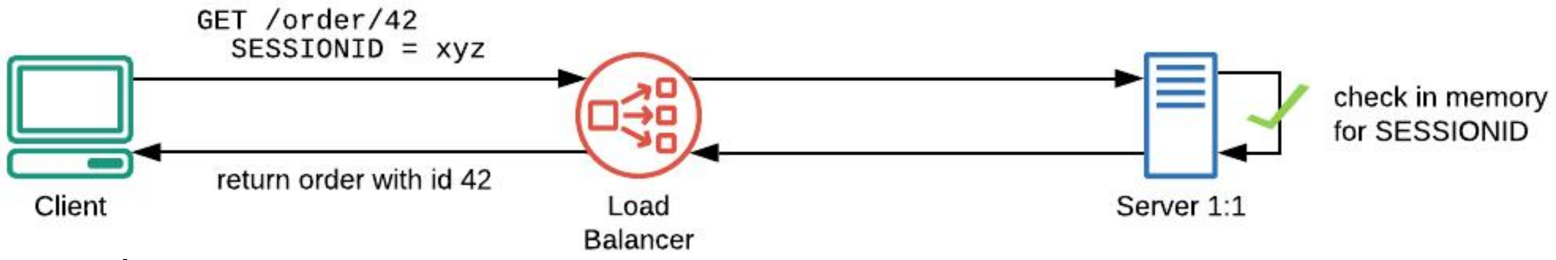
Problem of scaling

In the APIs era, our endpoints can face a huge amount of requests, so our infrastructures needs to scale. There are two types of scaling:

- vertical scaling – scaling up your infrastructure means merely add more resources to a server. This is an expensive solution with a low upper limit (i.e. the server's max resources allocation)
- horizontal scaling – scaling out your infrastructure is simpler and cost-effective as add a new server behind a load balancer

Now it's seems pretty clear that the second approach will be far most beneficial; but let's take a look at what may happens.

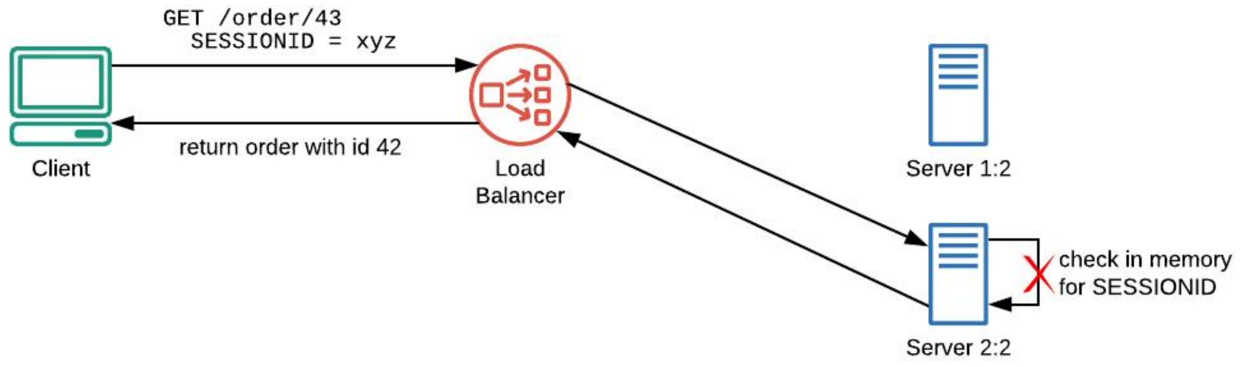
In the initial scenario, behind the load balancer, there's just one server. When a client will perform a request, using session id xyz, its record will be surely found in server's memory



So far, so good

Problem of scaling

Now imagine that the above infrastructure needs to scale. A new server (i.e. Server 2:2) will be added behind the load balancer and this brand new server will handle the next request issued by xyz client…



Unauthenticated! The brand new server, has no `xyz` sessions in its memory so the authentication process will fail. To fix this we have mainly three *workarounds* that can be used:

1. Synchronize sessions between servers — tricky and error-prone;

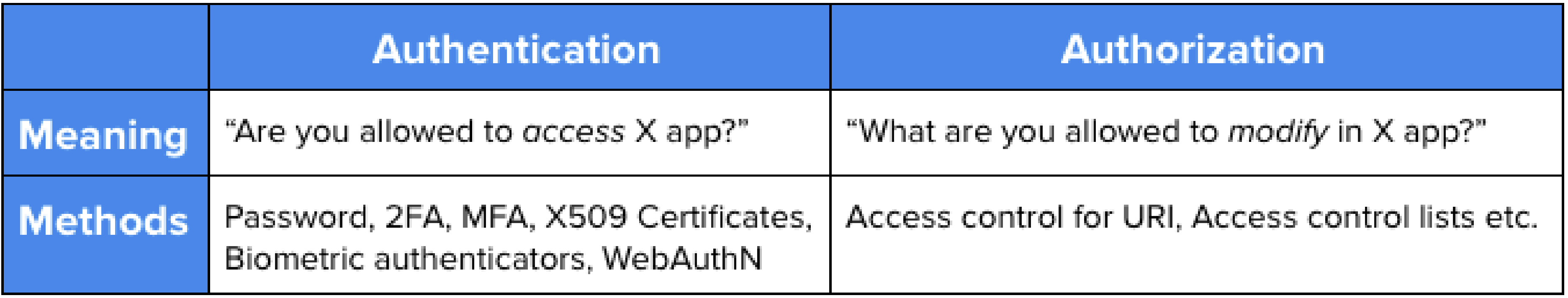
2. Use an external in-memory database — Good solutions but it will add another infrastructure's component

3. JWT : Before we look into JWT, let us understand few concepts

Authentication and Authorization

Authentication means confirming your own identity, while authorization means granting access to the system.

In simple terms, authentication is the process of verifying who you are, while authorization is the process of verifying what you have access to.



	Authentication	Authorization
Meaning	"Are you allowed to *access* X app?"	"What are you allowed to *modify* in X app?"
Methods	Password, 2FA, MFA, X509 Certificates, Biometric authenticators, WebAuthN	Access control for URI, Access control lists etc.

2FA : 2-factor authentication,
MFA: Multi Factor Authentication

Authentication

Based on the security level, authentication factor can vary from one of the following:

Single-Factor Authentication – Also known as primary authentication, this is the simplest and most common form of authentication. Single Factor Authentication requires, of course, only one authentication method such as a password, security pin, PIV card, etc. to grant access to a system or service.

Two-Factor Authentication(2FA) - Adding a layer of complexity, 2FA requires a second factor to verify a user's identity. Common examples include tokens generated by a registered device, One Time Passwords (OTP) or PIN numbers.

Multi-Factor Authentication(MFA)-

This is the most sophisticated authentication method that leverages 2 or more independent factors to grant user access to a system. In typical scenarios, MFA methods leverage at least 2 or 3 of the following categories.

1. **Something you know** - a password or a pin
2. **Something you have** - mobile phone or a security token
3. **Something you are** - fingerprint or FaceID
4. **Something you do** - typing speed, locational information etc.

API Authentication methods

In the age of the API economy, API’s handle large volumes of data and add a new dimension to the security surface of an online service.

While there are many API authentication methods, most of them can be categorized within one of three methods

1. HTTP Basic Auth

Using this approach, a user agent simply provides a username and password to prove their authentication.

This approach does not require cookies, session ID’s, or login pages because it leverages the HTTP header itself.

While simple to use, this method of authentication is vulnerable to attacks that could capture the user’s credentials in transit.

API Authentication

2. API Keys

An **application programming interface key** (**API key**) is a unique identifier used to authenticate a user, developer, or calling program to an API.

API keys are generally not considered secure; they are typically accessible to clients, making it easy for someone to steal an API key. Once the key is stolen, it has no expiration, so it may be used indefinitely, unless the project owner revokes or regenerates the key.

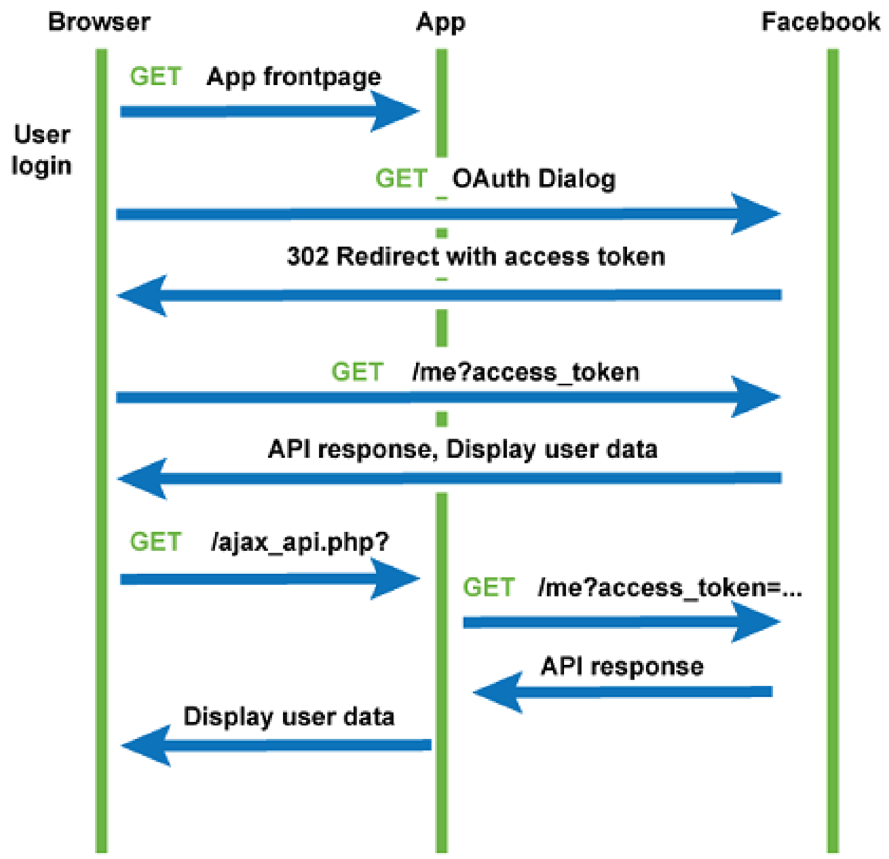
Oauth 2

3. Oauth 2

OAuth (Open Authorization) is an open standard authorization framework for token-based authorization also called as bearer authentication.

The general way it works is allowing an application to have an access token (which represents a user's permission for the client to access their data) which it can use to authenticate a request to an API endpoint.

The process for obtaining the token is called an authorization flow.



Authorization Flow

OAuth 2 Terms

Conceptually, OAuth2 has a few components interacting:

The **resource server** (the API server) contains the resources to be accessed.

Access tokens are provided by the **authorization server** (which can be the same as the API server) to the user.

These tokens are provided by the **resource owner** (the user/client/consumer) to the **resource server** while accessing the resources.

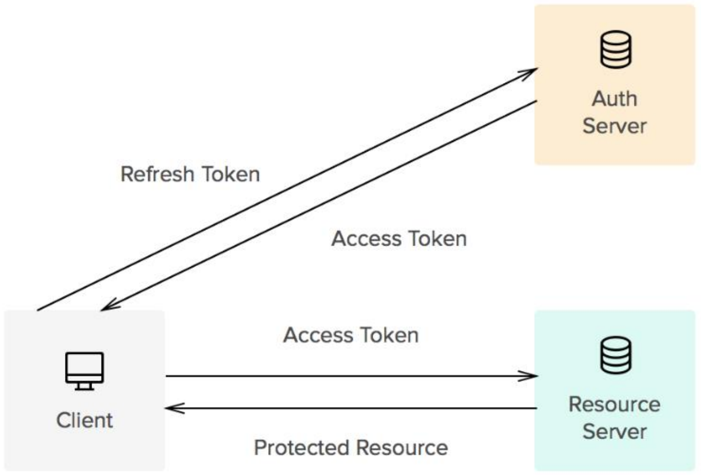
Note: Token is a specially crafted piece of data that carry enough information to authorize the user/client to perform an action

Access Token and Refresh Token

Bearer authentication (also called token authentication) is an HTTP authentication scheme that involves security tokens called bearer tokens. There are two types of bearer tokens involved in OAuth 2 authentication are **Access Token** and **Refresh Token**.

Access Token
The access token is used to for authentication and authorization to get access to the resources from the resource server.

Refresh Token
The refresh token normally is sent together with the access token and is used to get a new access token, when the old one expires.



The idea of refresh tokens is that if an access token is compromised, because it is short-lived, the attacker has a limited window in which to abuse it.

Refresh tokens, if compromised, are useless because the attacker requires the client id and secret in addition to the refresh token in order to gain an access token.

JSON Web Token (JWT)

JWTs can be used as OAuth 2.0 Bearer Tokens to encode all relevant parts of an access token into the access token itself instead of having to store them in a database.

A JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

This information can be verified and trusted because it is digitally signed.

JWTs can be signed using a secret or a public/private key pair

JSON Web Token Structure

JSON Web Token Structure

- Header
- Payload
- Signature

Header

The header typically consists of two parts: the type of token, which is JWT, and the hashing algorithm that is used, such as HMAC SHA256 or RSA.

For example:

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

Then, this JSON is Base64Url-encoded to form the first part of the JWT.

JSON Web Token Structure

Payload or body

The payload is the most important part of a JWT token. It contains information (claims in JWT jargon) about the client:

```
{
  "userId": "abcd12345ghijk",
  "username": "digitech",
  "email": "contact@gmail.com",
  // standard fields
  "iss": "Srini, Author",
  "iat": 1570238918,
  "exp": 1570238992
}
```

iss (Issuer): who issues the JWT
iat (Issued at): time the JWT was issued at
exp (Expiration Time): JWT expiration time

The payload is then encoded as Base64URL

JSON Web Token Structure

Signature

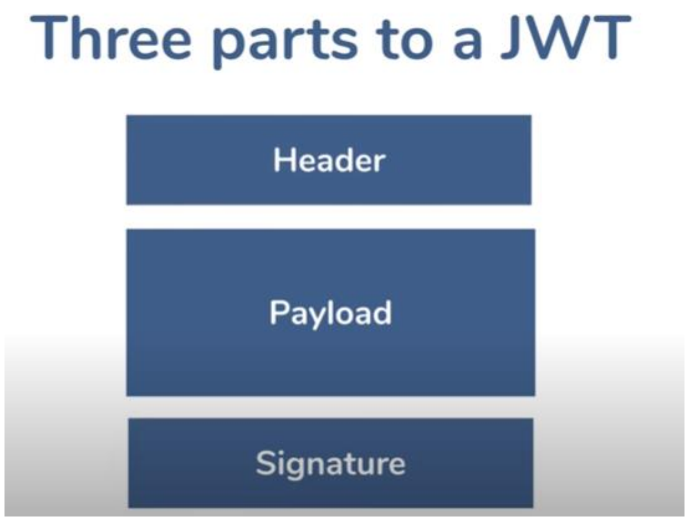
To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Then, you have to put it all together.

To put these concepts into practice, you can use https://jwt.io/

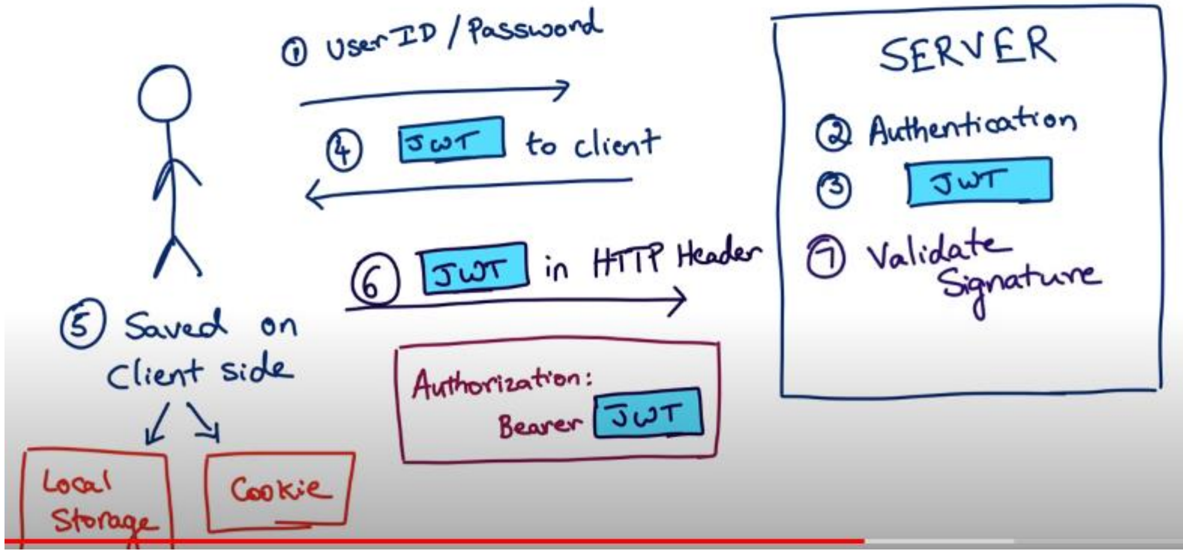
JWT (JSON Web Token) Structure

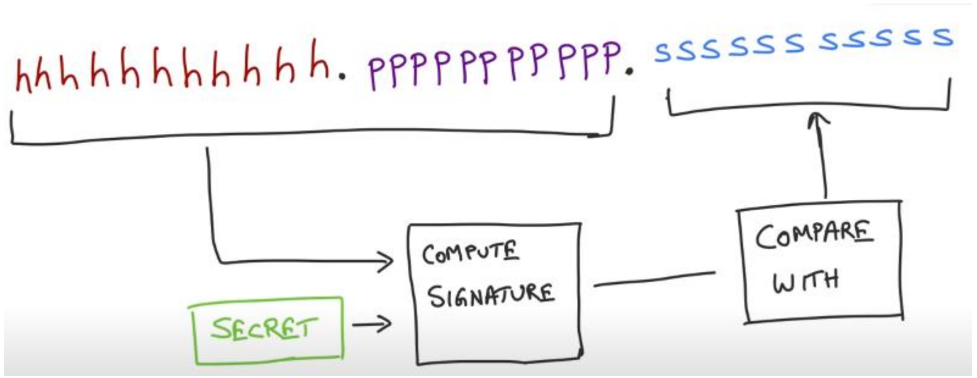

Three parts to a JWT

Header

Payload

Signature





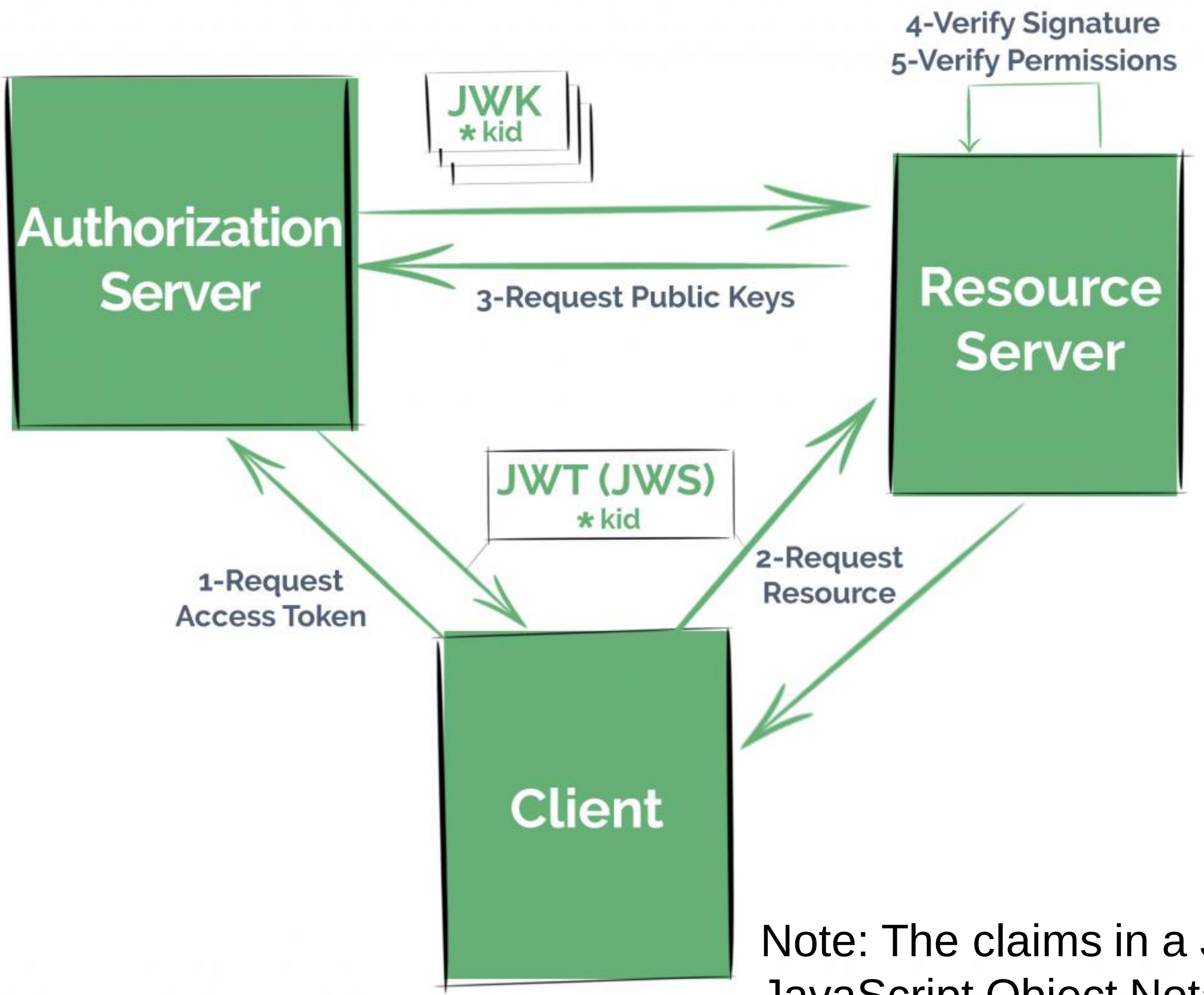
OAuth 2.0 with Spring Security

When building a web application, **authentication** and **authorization** is mandatory.

Spring Security and Spring Boot have made implementing web application using OAuth 2.0 easier.

Let us build an OAuth 2.0 web application and authentication server using Spring Boot and Spring Security.

OAuth 2.0 with Spring Security



Note: The claims in a JWT are encoded as a JavaScript Object Notation (JSON) object that is used as the payload of a JSON Web Signature (JWS) structure

Thank You!