https://hackernoon.com/using-session-cookies-vs-jwt-for-authentication-sd2v3vci

HTTP is a stateless protocol and is used to transmit data. It enables the communication between the client side and the server side. It was originally established to build a connection between web browsers and web servers.

Let’s understand this with the help of an example:

Suppose we are shopping online, we add some items eg. headphone to our cart. Then we proceed to look for other items, during this time, we want the state of the cart items to be stored and not lost while performing any further tasks. This means that we want our states to be remembered during the entire shopping operation.

Since HTTP is a stateless protocol, to overcome this, we can use either session or tokens. This blog will help you understand the difference between both the authentication methods used for user authentication.

So let’s dive into the concepts of session (cookie) based and token based authentication, along with understanding the structure of the JSON Web Tokens i.e JWT.

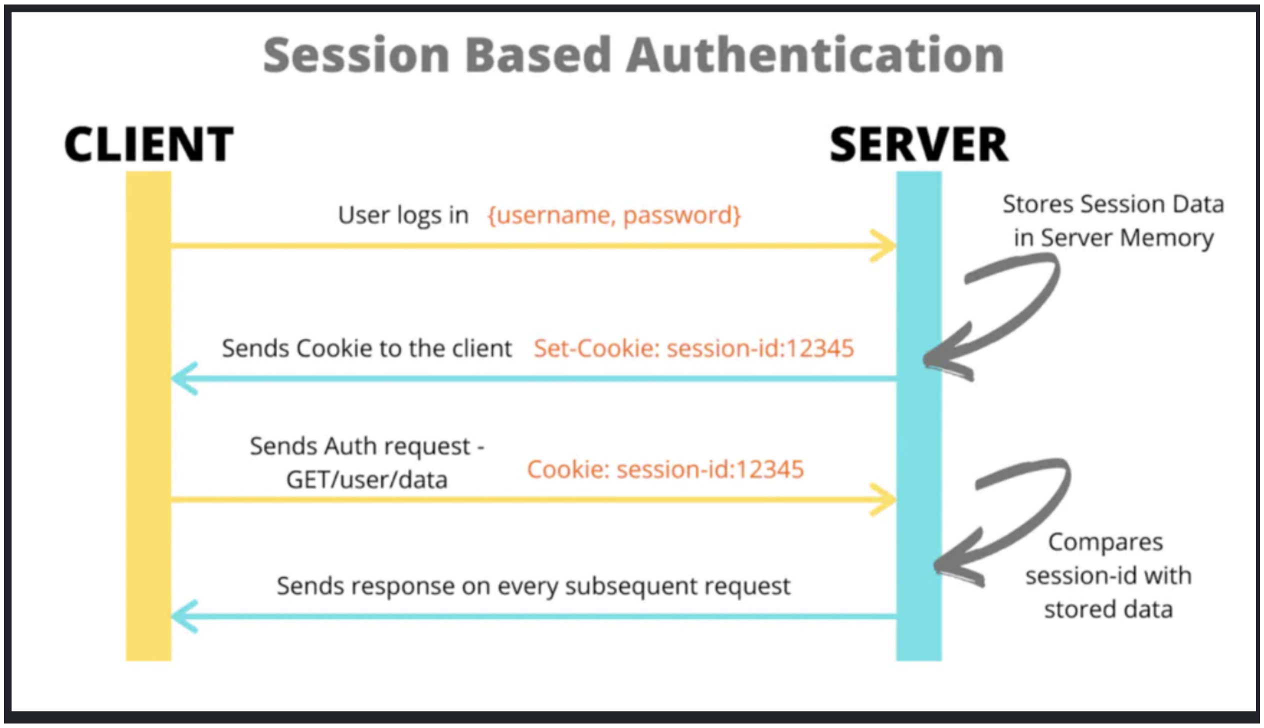
Session-Based Authentication

Before the emergence of JSON Web Tokens, we predominantly used this type of authentication.

In this type of authentication method, the server is responsible for the authentication and the client does not know what happens at the server side after sending a request.

So what is session cookies?

The cookie created above is a session cookie: it is deleted when the client shuts down, because it didn't specify an Expires or Max-Age directive. However, web browsers may use session restoring, which makes most session cookies permanent, as if the browser was never closed.


Session Based Authentication
CLIENT
SERVER
User logs in {username, password}
Stores Session Data in Server Memory
Sends Cookie to the client Set-Cookie: session-id:12345
Sends Auth request - GET/user/data
Cookie: session-id:12345
Compares session-id with stored data
Sends response on every subsequent request

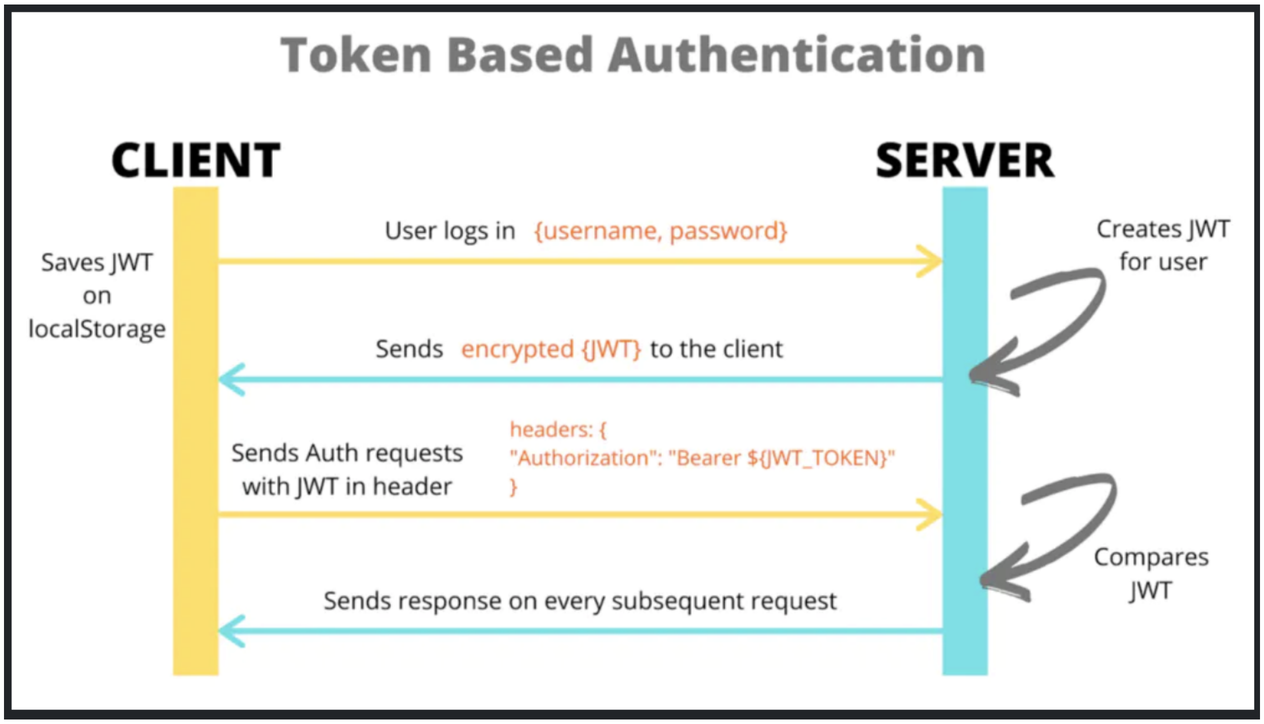
Let's understand what happens typically when a user logs into any website on the web browser. For instance, the user logs in, the server will create a session for the user and store the session data in the server memory.

There is a session ID created which is stored in a cookie in the client's browser while the user performs certain activity on the website. On every request that the user makes, a cookie is sent along with it.

The server can then verify the session data on the cookie with the session data stored in the server memory when the user logged in initially. When the user logs out from the website, that session data is deleted from the database and the server memory.

Token-Based Authentication:-

In token-based authentication, we use JWTs (JSON Web Tokens) for authentication. This is the widely used method for RESTful APIs.


Token Based Authentication
CLIENT
SERVER
User logs in {username, password}
Creates JWT for user
Saves JWT on localStorage
Sends encrypted {JWT} to the client
Sends Auth requests with JWT in header
headers: {
"Authorization": "Bearer ${JWT_TOKEN}"
}
Compares JWT
Sends response on every subsequent request

Here, when the user sends a request for user authentication with the login details, the server creates an encrypted token in the form of JSON Web Token (JWT) and sends it back to the client. When the client receives a token, it means that the user is authenticated to perform any activity using the client.

The JWT is stored on the client side usually in localStorage and it is sent as an unique key of that user when the user requests any data from the server or is performing any activity for that website. So, when the request is received by the server, it validates the JWT for every request that it is that particular user only and then sends the required response back to the client.

This is how the header is sent along with every request to the server :

```
headers: {
"Authorization": "Bearer ${JWT_TOKEN}"
}
```

The user's state is stored in the JWT on the client side. When the user logs out, the token is deleted from the Client side(localStorage). Thus, most of the data is stored in the client side and accessed directly instead of sending requests to the server.

Which one is better to use?

In modern web applications, JWTs are widely used as it scales better than that of a session-cookie based because tokens are stored on the client-side while the session uses the server memory to store user data, and this might be an issue when a large number of users are accessing the application at once. Since JWT is sent along with every request and contains all the user information, even though it is encoded, it is necessary to use the necessary information in JWT and sensitive information should be avoided or can be encrypted to prevent security attacks.

There is no fixed method to use at all times, it depends on the developer and the type of requirements to figure out which needs to be used for which scenario. For instance, OAuth uses a specific bearer-token and longer-lived refresh token to get bearer token. It is better to use a hybrid of both the types to create flexibility and robust application.

—---

SAML vs. OAuth

SAML (Security Assertion Markup Language) is an alternative federated authentication standard that many enterprises use for Single-Sign On (SSO). SAML enables enterprises to monitor who has access to corporate resources.

There are many differences between SAML and OAuth. SAML uses XML to pass messages, and OAuth uses JSON. OAuth provides a simpler mobile experience, while SAML is geared towards enterprise security. That last point is a key differentiator: OAuth uses API calls extensively, which is why mobile applications, modern web applications, game consoles, and Internet of Things (IoT) devices find OAuth a better experience for the user. SAML, on the other hand, drops a session cookie in a browser that allows a user to access certain web pages – great for short-lived work days, but not so great when have to log into your thermostat every day.

—---

https://blog.e-zest.com/rest-authentication-using-oauth-2-0-resource-owner-password-flow-protocol/

What is Oauth 2.0?

Introduction

In this blog we will look at how to perform authentication for an application exposing its functionalities as Rest Services. All requests made to access protected resource of such an application needs to be accessible only to Authenticated Users.

We will use Spring Security with OAuth 2.0 to authenticate Users and provide access to protected resources.

What is Oauth 2.0?

Oauth 2.0 is an open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. It is a simple way to publish and interact with protected data. It's also a safer and more secure way for server to give access to its protected resources. OAuth protocol enables a user to access protected resources from trusted client through an API.

How it works?

We will use the OAuth - Resource Owner Password Flow protocol to provide access to a protected resource on the server.

Obtain the Access Token

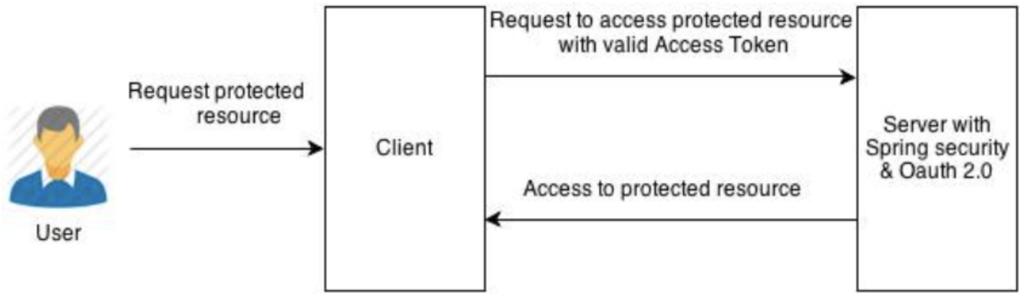

Request to access protected resource with valid Access Token
Request protected resource
Client
Server with Spring security & Oauth 2.0
Access to protected resource
User

Fig: Accessing protected resource

- **Client needs to provide the access token retrieved in the above steps along with the request to access the protected resource. Access token will be sent as an authorization parameter in the request header**
- **Server will authenticate the request based on the token**
- **If token is valid then client will get an access to protected resource otherwise access is denied**

The benefit of OAuth is that the API does not require users to disclose their credentials every time they access a protected resource. User provides the access token obtained during login, for all subsequent requests to access the protected resources.

Implementation

Below are some important configurations that need to be done in the spring-servlet.xml file to configure Oauth 2.0 -

```
<http pattern="/oauth/token" create-session="stateless"
                authentication-manager-ref="authenticationManager"
                xmlns="http://www.springframework.org/schema/security" >
<intercept-url pattern="/oauth/token" access="IS_AUTHENTICATED_FULLY" />
        <anonymous enabled="false" />
        <http-basic entry-point-ref="clientAuthenticationEntryPoint"/>
        <custom-filter ref="clientCredentialsTokenEndpointFilter"
before="BASIC_AUTH_FILTER" />
        <access-denied-handler ref="oauthAccessDeniedHandler" />
</http>
```

```
<bean id="clientCredentialsTokenEndpointFilter"
class="org.springframework.security.oauth2.provider.client.ClientCredentialsTokenEndpoint
Filter">
<property name="authenticationManager" ref="authenticationManager" />
</bean>

<authentication-manager alias="authenticationManager"
                        xmlns="http://www.springframework.org/schema/security">
<authentication-provider user-service-ref="clientDetailsUserService" />
</authentication-manager>
<bean id="clientDetailsUserService"
class="org.springframework.security.oauth2.provider.client.ClientDetailsUserDetailsService"
>
<constructor-arg ref="clientDetails" />
</bean>
<bean id="clientDetails"
class="demo.oauth2.authentication.security.ClientDetailsServiceImpl"></bean>
<bean id="clientAuthenticationEntryPoint"
class="org.springframework.security.oauth2.provider.error.OAuth2AuthenticationEntryPoint"
>
<property name="realmName" value="springsec/client" />
<property name="typeName" value="Basic" />
</bean>
<bean id="oauthAccessDeniedHandler"
class="org.springframework.security.oauth2.provider.error.OAuth2AccessDeniedHandler"/>
```

<http> configuration:

This tag is defined to process request for authentication.

1. For Authenticating a User the Client application will send request to the URL *"/oauth/token"*. This request will contain the clientId, client password, userName and user password details
2. The *clientCredentialsTokenEndpointFilter* filter will extracts client credentials from request & create authentication object in security context
3. Then authentication manager extracts the Authentication Object set in the SecurityContext and passes it to the *clientDetailsUserService*
4. The clientDetailsUserService holds reference to an instance of a custom bean clientDetails. The ClientDetailsServiceImpl class implements the ClientDetailsService interface and implements the method loadClientByClientId. The clientDetailsUserService bean calls this method from to authenticate the client

5. If the Client Credentials are valid then authentication manager will update authentication object in the application security context and set the authenticated flag to true
6. The request is then forwarded to the Oauth server for retrieving the access token
7. If the authentication fails the request is forwarded to the clientAuthenticationEntryPoint which displays the appropriate error to the User

```
<oauth:authorization-server

client-details-service-ref="clientDetails" token-services-ref="tokenServices">

<oauth:authorization-code />

        <oauth:implicit/>

        <oauth:refresh-token/>

        <oauth:client-credentials />

        <oauth:password authentication-manager-ref="userAuthenticationManager"/>

</oauth:authorization-server>


<authentication-manager id="userAuthenticationManager"

        xmlns="http://www.springframework.org/schema/security">

        <authentication-provider  ref="customUserAuthenticationProvider">

        </authentication-provider>

</authentication-manager>


<bean id="customUserAuthenticationProvider"

class="demo.oauth2.authentication.security.CustomUserAuthenticationProvider">

</bean>


<bean id="tokenServices"

class="org.springframework.security.oauth2.provider.token.DefaultTokenServices">

<property name="tokenStore" ref="tokenStore" />
```

```
<property name="supportRefreshToken" value="true" />

<property name="accessTokenValiditySeconds" value="900000000"></property>

<property name="clientDetailsService" ref="clientDetails" />

</bean>


<bean id="tokenStore"

class="org.springframework.security.oauth2.provider.token.InMemoryTokenStore" />


<bean id="oauthAuthenticationEntryPoint"

class="org.springframework.security.oauth2.provider.error.OAuth2AuthenticationEntryPoint"
>

</bean>


<oauth:authorization-server>
```

This tag is defined to configure authorization-server of oauth. Server serves requests based on grant type.

1. **The Oauth Server receives the request once the client is authenticated using above steps**
2. **Since the Grant type of request is a password, client and user needs to be authenticated by the Oauth server**
3. **Since the Client is already authenticated in process mentioned above the Oauth server now checks for the User authentication using the userAuthenticationManager**
4. **Then *userAuthenticationManager* extracts the Authentication Object set in the SecurityContext and passes it to the *customUserAuthenticationProvider***

   <- (Uniware)

- ***Within OAuth2AuthenticationProvider we make an HTTP post to UniAuth Server (API to validate user on uniAuth server) for validating user credentials.***
- ***If the User Credentials are valid then the authentication manager will update the authentication object in the application security context and set the authenticated flag to true.***

  ->

5. **The *CustomUserAuthenticationProvider* class implements the AuthenticationProvider interface and implements the method authenticate to authenticate the user**
6. **If the User Credentials are valid then authentication manager will update authentication object in the application security context and set the authenticated flag to true**
7. **Then *tokenServices* will then generate the access token using *tokenStore***
8. **The *tokenStore* uses *InMemoryTokenStore* to create access token and wraps authentication from SecurityContext into token**
9. **Authorization-server will send the access token in response to user request**
10. **If authentication fails, the request is forwarded to the *oauthAuthenticationEntryPoint* which displays the appropriate error to the User**

```
<http pattern="/resources/**" create-session="never"

            entry-point-ref="oauthAuthenticationEntryPoint"

            xmlns="http://www.springframework.org/schema/security">

            <anonymous enabled="false" />

            <intercept-url pattern="/resources/**"      method="GET" />

            <custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER" />

            <access-denied-handler ref="oauthAccessDeniedHandler" />

</http>

<oauth:resource-server id="resourceServerFilter"

resource-id="springsec" token-services-ref="tokenServices" />
```

<http> configuration:

This tag is defined to process request to access protected resource.

1. **For accessing the protected resource Client application will send request to the URL *"/resources/**"*. This request will contain the access token in request header**
2. **The *resourceServerFilter* indicates that requested resources is oauth 2 protected**
3. **Then Oauth-authorization-server will retrieve the access token and authenticate the request**
4. **If the token is valid user will be redirected to protected resource otherwise the request is forwarded to the *oauthAuthenticationEntryPoint* which displays the appropriate error to the User**

```
<http pattern="/logout" create-session="never"
entry-point-ref="oauthAuthenticationEntryPoint"
access-decision-manager-ref="accessDecisionManager"
xmlns="http://www.springframework.org/schema/security">

        <anonymous enabled="false" />

        <intercept-url pattern="/logout" access="ROLE_CLIENT" method="GET" />

<sec:logout invalidate-session="true" logout-url="/logout" success-handler-
ref="logoutSuccessHandler"/>

        <custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER" />

        <access-denied-handler ref="oauthAccessDeniedHandler" />

</http>

<bean id="logoutSuccessHandler"

                class="demo.oauth2.authentication.security.LogoutImpl" >

<property name="tokenstore" ref="tokenStore"></property>

</bean>
```

This tag is defined to process request to logout the user from application.

1. For logout Client application will send request to the URL *"/logout"*. This request will contain the access token in request header
2. This request will be authenticated similar to request of accessing protected resource using *resourceServerFilter* and *accessDecisionManager*
3. Then session of the user is invalidated and the request is forwarded to the *logoutSuccessHandler*
4. The *LogoutImpl* class implements LogoutSuccessHandler interface and implements onLogoutSuccess method to remove access token of the user from *tokenStore*
5. The client is successfully logged out

STEPS TO RUN THE APPLICATION:

1. Import downloaded source demo.rest.springsecurity.oauth2.0.authentication as maven project in eclipse IDE
2. Start the application in Tomcat
3. Then use CURL(command line utility) to run HTTP requests using following commands

- **Request To authenticate:**

```
curl –X -v –d
"username=user1&amp;password=user1&amp;client_id=client1&amp;client_secret=client1&
amp;grant_type=password"   -X
POST"http://localhost:8044/demo.rest.springsecurity.oauth2.0.authentication/oauth/token"
```

- **Response after above request :**

```
{"access_token":"6fd0f4b7-ca03-49ff-ae46-eea5e6929325","token_type":"bearer","re
fresh_token":"49bdb713-1827-4d83-83dc-59fe225f4726","expires_in":299999}
```

Here access token from this response need to be use in every subsequent request to access protected resource as follows.

- **Request To access protected resource getMyInfo :**

```
curl -H "Authorization:Bearer 6fd0f4b7-ca03-49ff-ae46-eea5e6929325"
"http://localhost:8044/demo.rest.springsecurity.oauth2.0.authentication/getMyInfo”
```

—---

https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f

https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-2-rest-constraints-129a4b69a582

What you should know before reading this article:

You should have some understanding of what is HTTP and what is an API.

REST is an architectural style, or design pattern, for APIs.

Before we dive into what makes an API RESTful and what constraints and rules you should follow if you want to create RESTful APIs, let's explain 2 key terms:

1. ***Client — the client is the person or software who uses the API. It can be a developer, for example you, as a developer, can use Twitter API to read and write data from Twitter, create a new tweet and do more actions in a program that you write. Your program will call Twitter's API. The client can also be a web browser. When you go to Twitter website, your browser is the client who calls Twitter API and uses the returned data to render information on the screen.***
2. ***Resource — a resource can be any object the API can provide information about. In Instagram's API, for example, a resource can be a user, a***

photo, a hashtag. Each resource has a unique identifier. The identifier can be a name or a number.

Now let's get back to REST.

A RESTful web application exposes information about itself in the form of information about its resources. It also enables the client to take actions on those resources, such as create new resources (i.e. create a new user) or change existing resources (i.e. edit a post).

In order for your APIs to be RESTful, you have to follow a set of constraints when you write them. The REST set of constraints will make your APIs easier to use and also easier to discover, meaning a developer who is just starting to use your APIs will have an easier time learning how to do so.

REST stands for REpresentational State Transfer.

It means when a RESTful API is called, the server will transfer to the client a representation of the state of the requested resource.

For example, when a developer calls Instagram API to fetch a specific user (the resource), the API will return the state of that user, including their name, the number of posts that user posted on Instagram so far, how many followers they have, and more.

The representation of the state can be in a JSON format, and probably for most APIs this is indeed the case. It can also be in XML or HTML format.

What the server does when you, the client, call one of its APIs depends on 2 things that you need to provide to the server:

1. ***An identifier for the resource you are interested in. This is the URL for the resource, also known as the endpoint. In fact, URL stands for Uniform Resource Locator.***
2. ***The operation you want the server to perform on that resource, in the form of an HTTP method, or***

verb. The common HTTP methods are GET,

POST, PUT, and DELETE.

For example, fetching a specific Twitter user, using Twitter's RESTful API, will require a URL that identify that user and the HTTP method GET.

Another example, this URL: www.twitter.com/jk_rowling has the unique identifier for J. K. Rowling's Twitter user, which is her username, jk_rowling. Twitter uses the username as the identifier, and indeed Twitter usernames are unique — there are no 2 Twitter users with the same username.

The HTTP method GET indicates that we want to get the state of that user.

This part explains the 6 REST constraints-

In order for an API to be RESTful, it has to adhere to 6 constraints:

- *Uniform interface*

- ***Client — server separation***
- ***Stateless***
- ***Layered system***
- ***Cacheable***
- ***Code-on-demand***

Uniform interface

This constraint has 4 parts:

1. ***The request to the server has to include a resource identifier***
2. ***The response the server returns include enough information so the client can modify the resource***
3. ***Each request to the API contains all the information the server needs to perform the request, and each response the server returns***

contain all the information the client needs in order to understand the response.

4. ***Hypermedia as the engine of application state — this may sound a bit cryptic, so let's break it down: by application we mean the web application that the server is running. By hypermedia we refer to the hyperlinks, or simply links, that the server can include in the response. The whole sentence means that the server can inform the client , in a response, of the ways to change the state of the web application. If the client asked for a specific user, the server can provide not only the state of that user but also information about how to change the state of the user, for example how to update the user's name or how to delete the user. It is easy to think about the way it's done by thinking about a***

server returning a response in HTML format to a browser (which is the client). The HTML will include tags with links (this is the hypermedia part) to another web page where the user can be updated (for example a link to a 'profile settings' page). To put all of this in perspective, most web pages do implement hypermedia as the engine of application state, but the most common web APIs do not adhere to this constraint. To further understand this concept, I highly recommend watching this 30 minutes YouTube video.

The result of the uniform interface is that requests from different clients look the same, whether the client is a chrome browser, a linux server, a python script, an android app or anything else.

Client — server separation

The client and the server act independently, each on its own, and the interaction between them is only in the form of requests, initiated by the client only, and responses, which the server send to the client only as a reaction to a request. The server just sits there waiting for requests from the client to come. The server doesn't start sending away information about the state of some resources on its own.

Stateless

Stateless means the server does not remember anything about the user who uses the API. It doesn't remember if the user of the API already sent a GET request for the same resource in the past, it doesn't remember which resources the user of the API requested before, and so on.

Each individual request contains all the information the server needs to perform the request and return a response, regardless of other requests made by the same API user.

Layered system

Between the client who requests a representation of a resource's state, and the server who sends the response back, there might be a number of servers in the middle. These servers might provide a security layer, a caching layer, a load-balancing layer, or other functionality. Those layers should not affect the request or the response. The client is agnostic as to how many layers, if any, there are between the client and the actual server responding to the request.

Cacheable

This means that the data the server sends contain information about whether or not the data is cacheable. If the data is cacheable, it might contain some sort of a version number. The version number is what makes caching possible: since the client knows which version of the data it already has (from a previous response), the client can avoid requesting the same data again and again. The client should also know if the current version of the data is expired, in which case the client will know it should send another request to the server to get the most updated data about the state of a resource.

Code-on-demand

This constraint is optional — an API can be RESTful even without providing code on demand.

The client can request code from the server, and then the response from the server will contain some code, usually in the form of a script, when the response is in HTML format. The client then can execute that code.

Below is the main difference between SOAP and REST API

SOAP	REST

• SOAP stands for Simple Object Access Protocol	• REST stands for Representational State Transfer
• SOAP is a protocol. SOAP was designed with a specification. It includes a WSDL file which has the required information on what the web service does in addition to the location of the web service.	• REST is an Architectural style in which a web service can only be treated as a RESTful service if it follows the constraints of being 1. Client Server 2. Stateless 3. Cacheable 4. Layered System 5. Uniform Interface
• SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern.	• REST can make use of SOAP as the underlying protocol for web services, because in the end it is just an architectural pattern.
• SOAP uses service interfaces to expose its functionality to client applications. In SOAP, the WSDL file provides the client with the necessary information which can be used to understand what services the web service can offer.	• REST use Uniform Service locators to access to the components on the hardware device. For example, if there is an object which represents the data of an employee hosted on a URL as http://demo.guru99 , the below are some of URI that can exist to access them. http://demo.guru99.com/Employee http://demo.guru99.com/Employee/1

• SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.
`<?xml version="1.0"?>`
`<SOAP-ENV:Envelope`
`xmlns:SOAP-ENV`
`="http://www.w3.org/2001/12/soap-envelope"`
`SOAP-ENV:encodingStyle`
`=" http://www.w3.org/2001/12/soap-encoding">`
`<soap:Body>`
` <Demo.guru99WebService`
`xmlns="http://tempuri.org/">`
`<EmployeeID>int</EmployeeID>`
`</Demo.guru99WebService>`
` </soap:Body>`
`</SOAP-ENV:Envelope>`
	• REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages. Below is an example of a JSON message passed to a web server. You can see that the size of the message is comparatively smaller to SOAP.
	`{"city":"Mumbai","state":"Maharastra"}`

• SOAP can only work with XML format. As seen from SOAP messages, all data passed is in XML format.	• REST permits different data format such as Plain text, HTML, XML, JSON, etc. But the

	most preferred format for transferring data is JSON.