

REST API Design, Security Headers, Authentication and Authorization

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

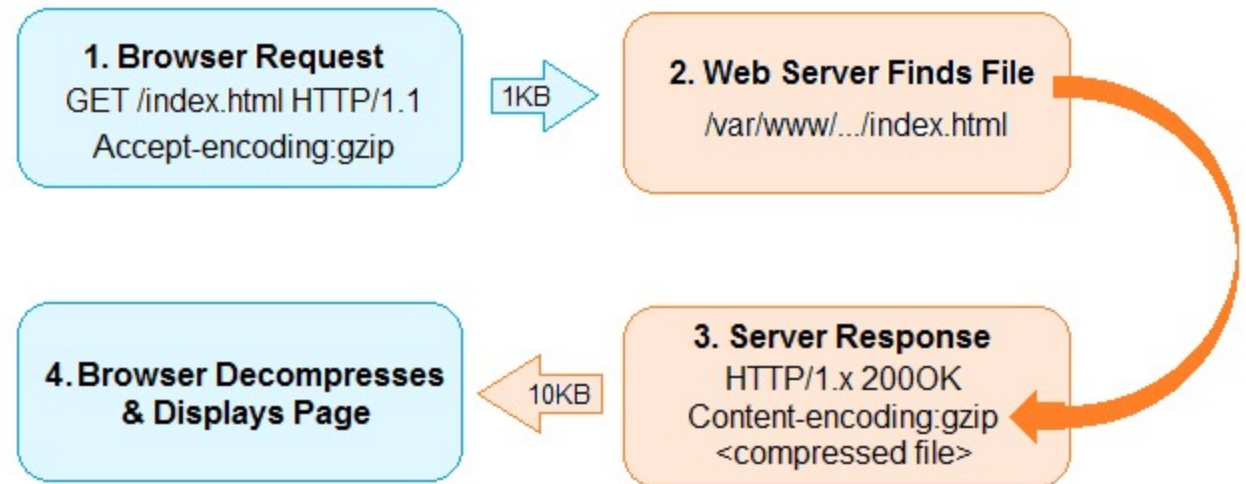
Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

HTTP Request

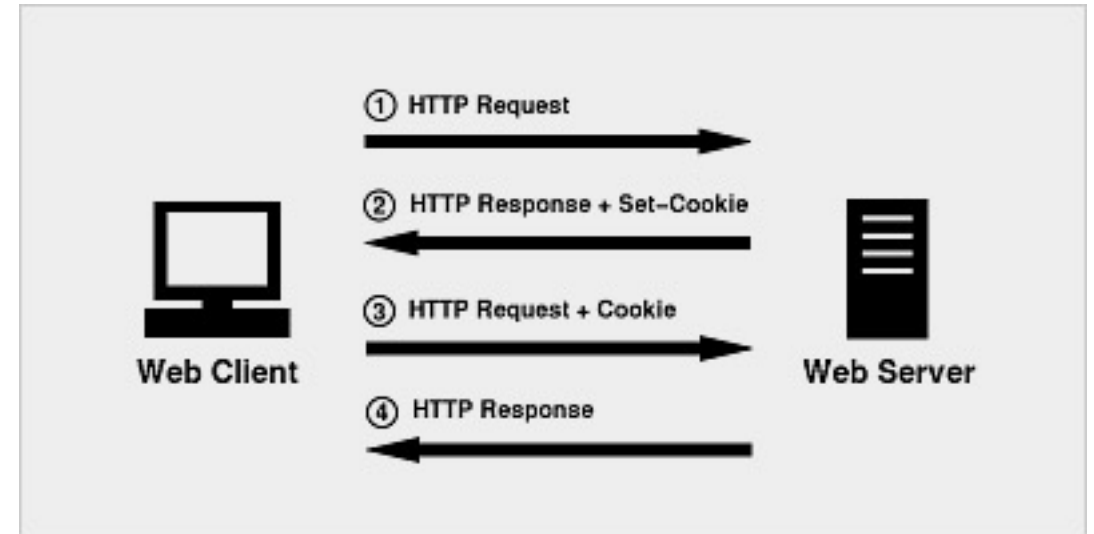
- Your browser is going to send a request to the server (GET, POST)
- The server sends a response back (HTML, JavaScript)
- The browser creates the DOM and wait for user interaction.



Cookies for GET/POST Requests

In the first response, the server will ask the browser to save a cookie for your domain

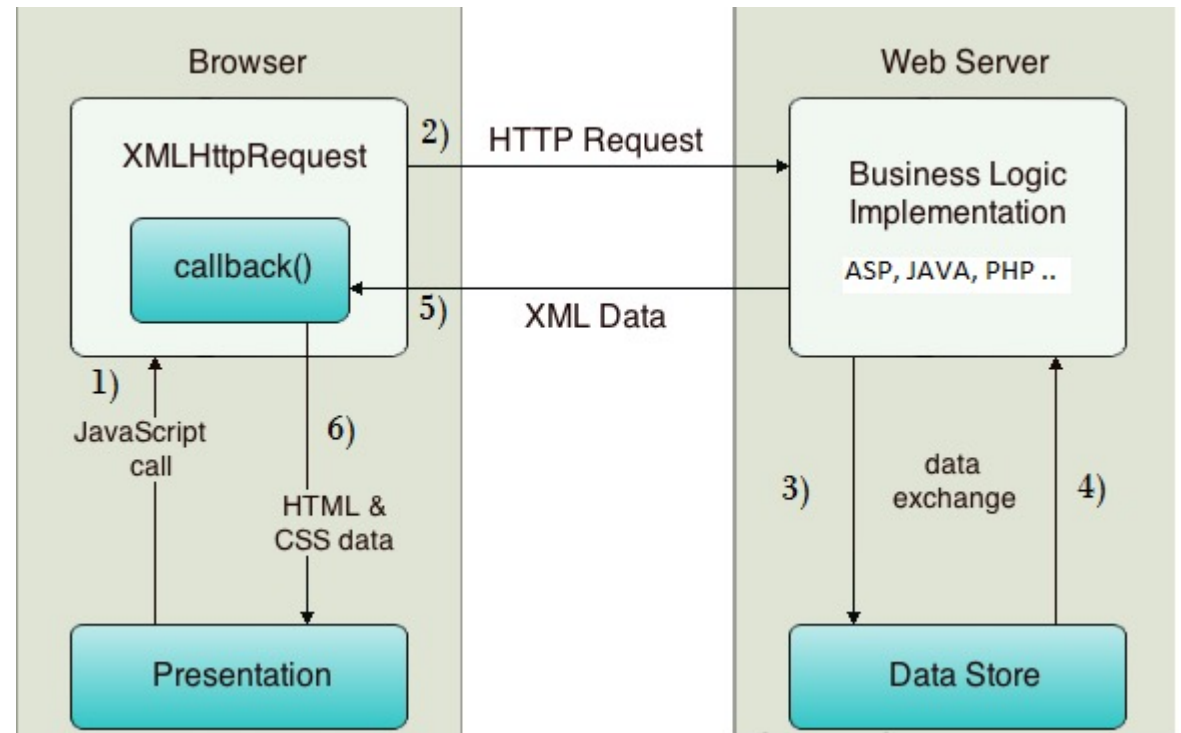
The browser will send all cookies that belong to your domain in all upcoming requests



AJAX Calls

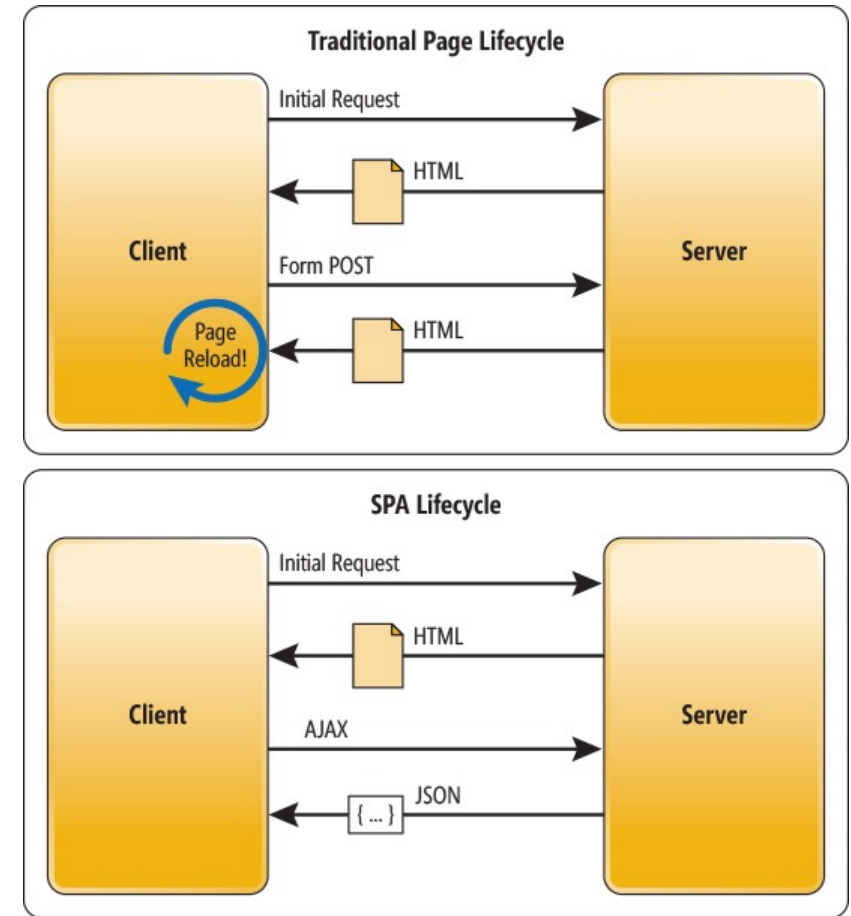
The browser XMLHttpRequest Object will send a request to the server (XHR) and register a callback function to handle the response

The server will send the response, which will be passed to the callback function



SPA Applications

- Your browser will send a request to your server and the server will send you back a response with full client-side application.
- All upcoming requests will be handled by your JavaScript in the browser (Routes will be on client side)
- The browser will **send AJAX** calls behind the scene to retrieve data from the server through Restful API when needed



Cross-Origin Resource Sharing (CORS)

CORS has been adopted by API providers as the primary way to share resources, it allows cross-origin requests without relying on JavaScript but rather HTTP.

CORS headers allow servers to specify a set of origins that are allowed to access its resources. If the request referrer header is on that list, It will be able to inspect the answer and use the data.



Preflight Request

A preflight request uses the **OPTIONS** method and allows the browser to signal that it only wants to check what is allowed and what is not. The server should not execute any kind of business logic, but only return the headers, similar to a **HEAD** request.


Not all requests will be preflighted. Requests that are made because of image tags or forms will not be preflighted. So any kind of **GET** request will be sent straight away. You just won't be able to read the answer if CORS doesn't allow it.

Requests that come from a form will not be preflighted.

Preflighted Request

Suppose web content on domain `http://foo.example` wishes to invoke content on domain `http://bar.other`

```
fetch('http://bar.other/API/public/',  
      { method: 'POST',  
        body: JSON.stringify(data),  
        headers:{ 'X-PING-COURSE':'CS572',  
                  'Content-Type': 'application/json' }  
      })  
.then(res => res.json())  
.then(response => console.log('Success:', JSON.stringify(response)))
```



Note that the `X-PING-COURSE` is the custom header that is inserted by JavaScript, and should differ from site to site.

Preflighted Request

1 Request 1

```
OPTIONS /API/public/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh)
Accept: text/html,application/json;
Accept-Language: en-us,en;
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PING-COURSE
```

2 Response 1

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PING-COURSE
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

The preflight request is a way of asking permissions for the actual request, before making the actual request. The server should inspect the two headers above to verify that both the HTTP method and the requested headers are valid and accepted.

Preflighted Request

3

Request 2

```
POST /API/public/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (MacintoshAccept:
text/html,application/json;
Accept-Language: en-us,en;
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;
Connection: keep-alive
X-PING-COURSE: CS572
Content-Type: application/json; charset=UTF-8
Referer: http://foo.example/page.html
Content-Length: 55
Origin: http://foo.example
Pragma: no-cache
Cache-Control: no-cache

{username: "asaad", password="123"}
```

4

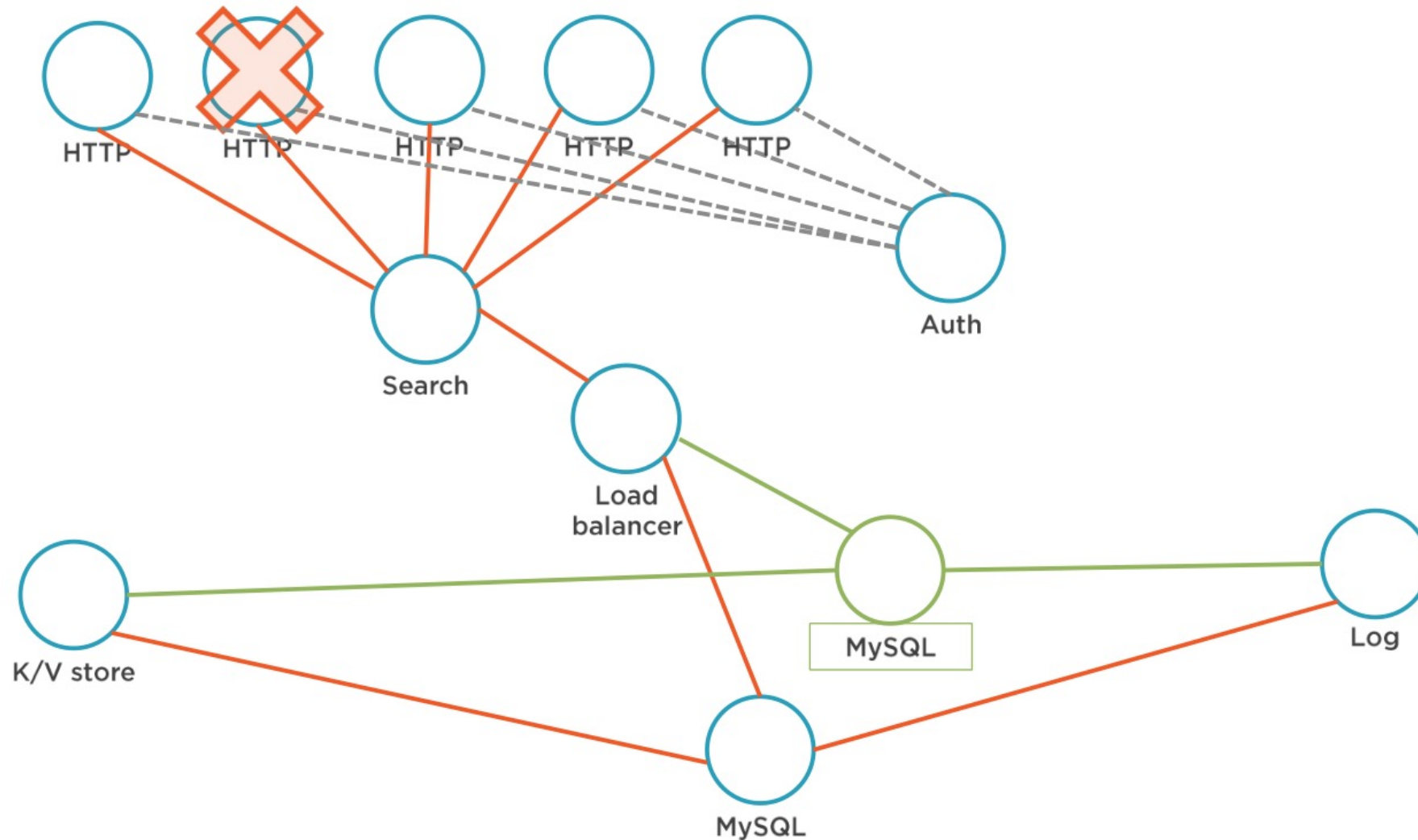
Response 2

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: application/json

[Some GZIP'd JSON payload]
```

Once the preflight request gives permissions, the browser makes the actual request. The actual request looks like the simple request

Modern Web Applications



Question?

What's going to happen if your application was hosted on multiple sub-domains or ports or protocols? How to handle session migration to protect your services and give access to authenticated users only?



Stateless vs Stateful Web Apps

Stateful

Keeps data (Authorization servers are often stateful services, they store issued tokens in database/memory for future checking)

Stateless

Does not keep any data (issuing JWT tokens as access tokens)

REST = Representational State Transfer

Architectural style for distributed systems

Exposes resources (state) to clients

Resources identified with a URI

Uses HTTP, URI, MIME types (JSON)

HTTP Verbs and CRUD Consistency

The following are the most commonly used server architecture HTTP methods and their corresponding Express methods:

GET `app.get()` Retrieves an entity or a list of entities

HEAD `app.head()` Same as GET, only without the body

POST `app.post()` Submits a new entity

PUT `app.put()` Updates an entity by complete replacement

PATCH `app.patch()` Updates an entity partially

DELETE `app.delete()` and `app.del()` Delete an existing entity

OPTIONS `app.options()` Retrieves the capabilities of the server

REST Example

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data.

URL	HTTP Verb	POST Body	Result
http://yourdomain.com/api/entries	GET	empty	Returns all entries
http://yourdomain.com/api/entries	POST	JSON String	New entry Created
http://yourdomain.com/api/entries/:id	GET	empty	Returns single entry
http://yourdomain.com/api/entries/:id	PUT	JSON string	Updates an existing entry
http://yourdomain.com/api/entries/:id	DELETE	empty	Deletes existing entry

REST API HTTP response codes

GET – with results: 200

GET – with no results: 204

POST – Insert new record: 201

PUT – Update resource: 202

Auth Failed: 401

Route Not Found: 404

System Error: 500

Naming your Endpoints

To begin, define a **prefix** for all your API endpoints, it can be a subdomain or a route param. You may add a **version** into the prefix (or within the request header), with support of serving the latest version if no API version were specified.

Use all **lowercase**, hyphenated endpoints are also acceptable as long as you are consistent about it.

Use a **noun** or two to describe the resource. Think as if resources and DB models were equivalents.

Always describe resources in **plural**.

API Versioning

Versioning is useful because it allows you to commit breaking changes to your service without breaking the code for existing consumers. It can be implemented in two ways:

API version should be set in **HTTP headers**, and that if a version isn't specified in the request, you should get a response from the latest version of the API.

API version is embedded into the **API endpoint prefix**:/api/v1/.... This also identifies right away which version of the API your application wants by looking at the requested endpoint.

Request and Response

An API is expected to take **arguments** via the endpoint. Having the identifier as part of the request endpoint is great because it allows DELETE and GET requests to use the same endpoint.

Responses should conform to a consistent data-transfer format, so you have no surprises when parsing the response. You should figure out the envelope in which you'll wrap your responses.

If our API is built using JSON , then all the responses produced by our API should be valid JSON (granted the user is accepting a JSON response in the HTTP headers).

```
{
  "data": {} // the actual response
}

{
  "error": { "code": "404",
             "message": "Product not found."}
}
```

Response Paging Headers

To implement paging, we use the **Link** header. The **rel** attribute describes the relationship between the requested page and the linked page.

```
Link: <http://example.com/api/products/?p=1>; rel="first",  
      <http://example.com/api/products/?p=1>; rel="prev",  
      <http://example.com/api/products/?p=3>; rel="next",  
      <http://example.com/api/products/?p=54>; rel="last"
```

Sometimes data flows too rapidly for traditional paging methods, if a few records make their way into the database between requests for the first page and the second one, the second page results in duplicates of items that were on page one but were pushed to the second page as a result of the inserts. One solution is to use identifiers instead of page numbers. This allows the API to figure out where you left off, and even if new records get inserted, you'll still get the next page in the context of the last range of identifiers that the API gave you.

Express Pagination `res.links()`

Populate the response Link HTTP header field.

```
res.links({  
  next: 'http://api.example.com/users?page=2',  
  last: 'http://api.example.com/users?page=5' })
```

Yields the following results:

```
Link: <http://api.example.com/users?page=2>; rel="next",  
      <http://api.example.com/users?page=5>; rel="last"
```

Response Caching

It's up to the client to cache API results as necessary. The API can make suggestions on how its responses should be cached.

Setting the **Cache-Control** header to **private** bypasses intermediaries (such as proxies like nginx)

The **Expires** header tells the browser that a resource should be cached and not requested again until the expiration date has elapsed

```
Cache-Control: private
```

```
Expires: Fri, 5 Dec 2020 18:31:12 GMT
```

It's hard to define future Expires headers in API responses because if the data in the server changes, it could mean that the client's cache becomes stale.

Conditional Requests

Conditional requests can be time-based, specifying a **Last-Modified** header in your responses. It's best to specify a **max-age** in the **Cache-Control** header, to let the browser invalidate the cache after a certain period of time even if the modification date doesn't change.

response {
Cache-Control: private, max-age=86400
Last-Modified: Fri, 5 Jan 2020 18:31:12 GMT

The next time the browser requests this resource, it will use the **If-Modified-Since** request header:

request {
If-Modified-Since: Fri, 5 Jan 2020 18:31:12 GMT

If the resource hasn't changed since the specified date, the server will return with an empty body with the 304 Not Modified status code.

Entity Tag

A hash that represents the resource in its current state. This allows the server to identify if the cached contents of the resource are different than the most recent version

Response {
Cache-Control: private, max-age=86400
ETag: "d5aae96d71f99e4ed31f15f0ffffdd64"

On subsequent requests, the **If-None-Match** request header is sent with the **Etag** value of the last requested version for the same resource:

Request {
If-None-Match: "d5aae96d71f99e4ed31f15f0ffffdd64"

If the current version has the same **Etag** value, your current version is what the client has cached and a 304 Not Modified response will be returned.

HTTP 2

Security/Encryption

A secure connection is required

Streams

Multiplexing streams, Header compression, Request prioritization, Server push

Brotli Compression

A new compression algorithm that has the potential to outperform gzip

SEO

Google uses HTTPS as a ranking signal so serving over HTTPS will increase your page ranking.

Security Headers

CSP Content Security Policy

content-security-policy:[CSP_POLICY]

STS Strict Transport Security

strict-transport-security:[STS_POLICY]

CSP Content Security Policy

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. Possible directives:

default-src, font-src, img-src, script-src, style-src,
block-all-mixed-content, upgrade-insecure-request

Example:

```
content-security-policy: script-src ajax.googleapis.com; upgrade-insecure-request;
```

```
content-security-policy-report-only: default-src 'self' mum.edu;  
report-uri https://my-api-to-accept-json-csp-report.io
```

STS Strict Transport Security

The HTTP Strict-Transport-Security response header (HSTS) lets a web site tell clients that it should **only be accessed using HTTPS**, instead of using HTTP. It doesn't only save the client from making one extra round to the server but also to stop the man-in-the-middle attacks.

Strict-Transport-Security: max-age=31536000; includeSubDomains

All present and future subdomains will be HTTPS for a max-age of 1 year. This blocks access to pages or subdomains that can only be served over HTTP.

Manipulating the Response Header

Both `res.header()` and `res.set()` are used to set the headers HTTP response.

```
// multiple headers can be set
res.set({
  'content-type': 'application/json',
  'content-length': '100',
  'warning': "this course is the best course ever"
});
```

Authentication vs Authorization

Authentication is the process of identifying that somebody really is who they claim to be.

Authorization is the process of verifying that you have access to something.

The two concepts are completely independent, but both are central to security design, and the failure to get either one correct opens up the avenue to compromise.

Authentication = login + password (who you are)

Authorization = permissions (what you are allowed to do)

Authentication in SPA

Traditional session-based authentication isn't a good fit for SPAs that use data APIs because it requires state on the server.

REST services are stateless and cannot hold a state on the server.

A better way to do authentication in modern web applications is with JSON Web Tokens (JWTs).

What is JSON Web Token?

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.

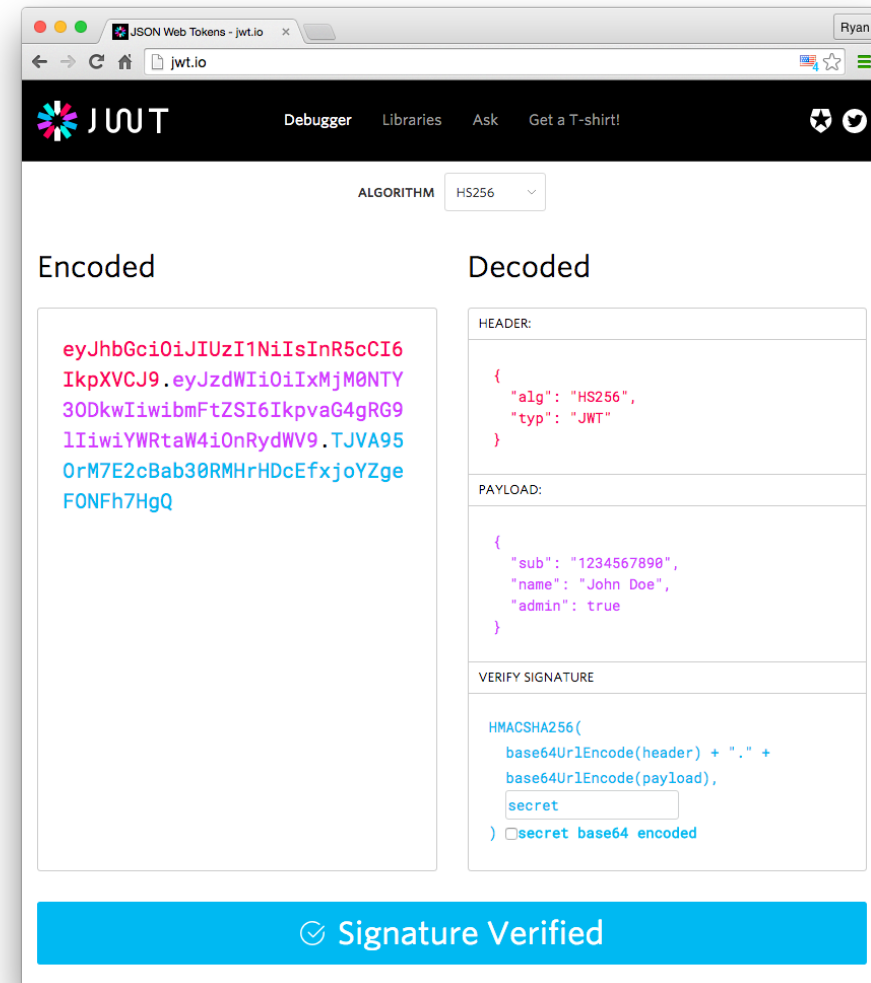
Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header.

JWT is Self-contained: The payload contains all the required information about the user, avoiding the need to query the database more than once.

What is the JSON Web Token structure?

JSON Web Tokens consist of
three parts separated by dots (.)
Header.Payload.Signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9.TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```



The screenshot shows the JWT.io website interface. At the top, there's a navigation bar with the JWT logo, a 'Debugger' tab, and links for 'Libraries', 'Ask', and 'Get a T-shirt!'. Below the navigation bar, there's a dropdown menu for 'ALGORITHM' set to 'HS256'. The main content area is split into two columns: 'Encoded' and 'Decoded'. The 'Encoded' column displays the full JWT string: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9.TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ`. The 'Decoded' column shows the decoded structure with three sections: 'HEADER:', 'PAYLOAD:', and 'VERIFY SIGNATURE'. The 'HEADER:' section shows a JSON object: `{ "alg": "HS256", "typ": "JWT" }`. The 'PAYLOAD:' section shows a JSON object: `{ "sub": "1234567890", "name": "John Doe", "admin": true }`. The 'VERIFY SIGNATURE' section shows the HMACSHA256 function being applied to the base64 encoded header and payload, with a 'secret' field for the key. At the bottom, a blue banner indicates 'Signature Verified'.

JWT

Debugger Libraries Ask Get a T-shirt!

ALGORITHM HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9.TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Decoded

HEADER:

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

PAYLOAD:

```
{  "sub": "1234567890",  "name": "John Doe",  "admin": true}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  secret  ) secret base64 encoded
```

Signature Verified

JWT Structure

Header: Where we define the **algorithm** used to sign the token, as well as the token type.

Payload: This is where we keep a JSON object of all the **claims** we want. Claims can include those that are registered in the JWT spec, as well as any arbitrary data we want. Some of the reserved claims are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.

Signature: The signature is where the signing action happens. To get the signature, we take the Base64URL encoded header, add the Base64URL encoded payload next to it, and run that string along with the secret key through the hashing algorithm we've chosen.

The Bearer Schema

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

Authorization: Bearer <token>

You may save the authorization data for the user in the JWT.

How Are JWTs Used to Authenticate Angular Apps?

Users send their credentials to the server which are verified against a database. If everything checks out, a JWT is sent back to them.

The JWT is saved in the user's browser by holding it in local storage.

The presence of a JWT saved in the browser is used as an indicator that a user is currently logged in.

The JWT's expiry time is continually checked to maintain an authenticated state in the app, and the user's details are read from the payload to populate views such as the their profile.

Access to protected client-side routes (such as the profile area) are limited to only authenticated users via Guards

When the user makes XHR requests to the API for protected resources, the request must be intercepted and the JWT gets sent as an Authorization header using the Bearer.

Middleware on the server, which is configured with the app's secret key checks the incoming JWT for validity and, if valid, sends the response.

JWT packages for Angular and Node

Angular

```
npm install jwt-decode
```

Node

```
npm install jsonwebtoken
```

Persist The Login Session

If the user logs in, we want to be able to remember that they are the same user across page views, and preferably across different sessions.

We basically have three choices: **localStorage**, **sessionStorage**, and **cookies**. All three of these mechanisms allows us to store data in key-value pairs as strings in the browser's memory.

LocalStorage

Data is **persistent in browser**.

Data is **only** accessible through **client-side JavaScript**.

Data is **not automatically sent to the server** during an HTTP request.

```
localStorage.setItem("courseId", "CS572");  
localStorage.removeItem("courseId");
```

SessionStorage

Data is **only** accessible through **client-side JavaScript**.

Data is **not automatically sent to the server** during an HTTP request.

Data is deleted once the browser tab or window is closed. Refreshing the web-page will not delete the data. Only by closing and reopening the browser tab, the data will be deleted.

```
sessionStorage.setItem("greeting", "Hello World!");  
sessionStorage.removeItem("greeting");
```

Cookies

Data is persistent in **browser**.

Data is **inaccessible with JavaScript on the client** (when `HttpOnly` flag is set).

Data is **automatically sent** to the server during an HTTP request.

By setting a cookie to `HttpOnly`, we ensure that the cookie can only be accessed from the server and can't be tampered with by JavaScript running on the browser.

localStorage & sessionStorage Security

localStorage & sessionStorage are accessible through JavaScript running in the browser. Authentication data stored in these types of storage are vulnerable to cross-site scripting (XSS) attacks. XSS is a vulnerability where attackers can inject client-side scripts to run on a web app.

To prevent XSS, we could:

Ensure that all links are from a trusted source.

Escape untrusted data (which is often automatically done when using a modern front-end framework such as Angular and React).

Cookie Security

Cookies, when used with the **HttpOnly** flag, are not accessible through JavaScript and thus are immune to XSS. We can also set a **Secure** flag to a cookie to guarantee the cookie is only sent over HTTPS. These are some of the reasons why cookies are often used to store tokens or session data.

On the other hand, cookies are vulnerable to a different type of attack - Cross-Site Request Forgery (CSRF).

The **SameSite** flag, is designed to counter CSRF attacks by ensuring that cookies are not sent with cross-site requests.