Building Web Applications with

express

CS572 Modern Web Applications Programming

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

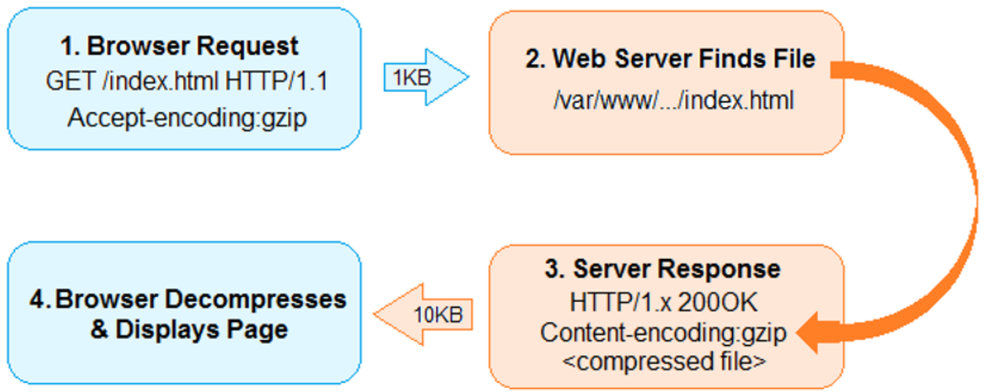
PREV NEXT

OUTLINE

Search...

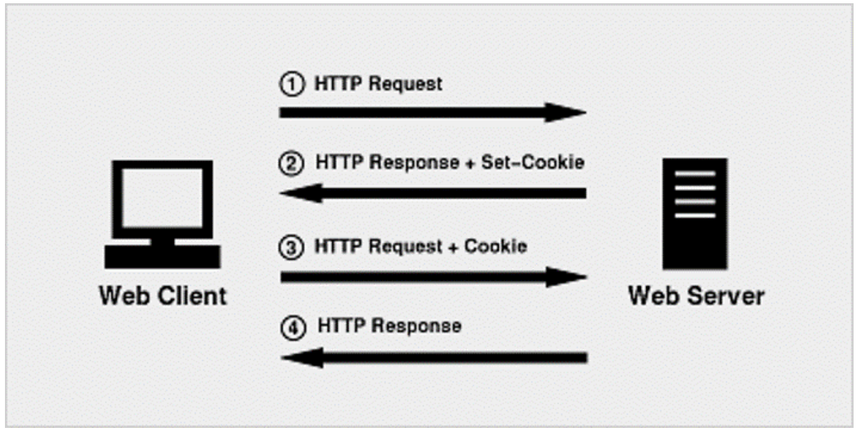
HTTP Request

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



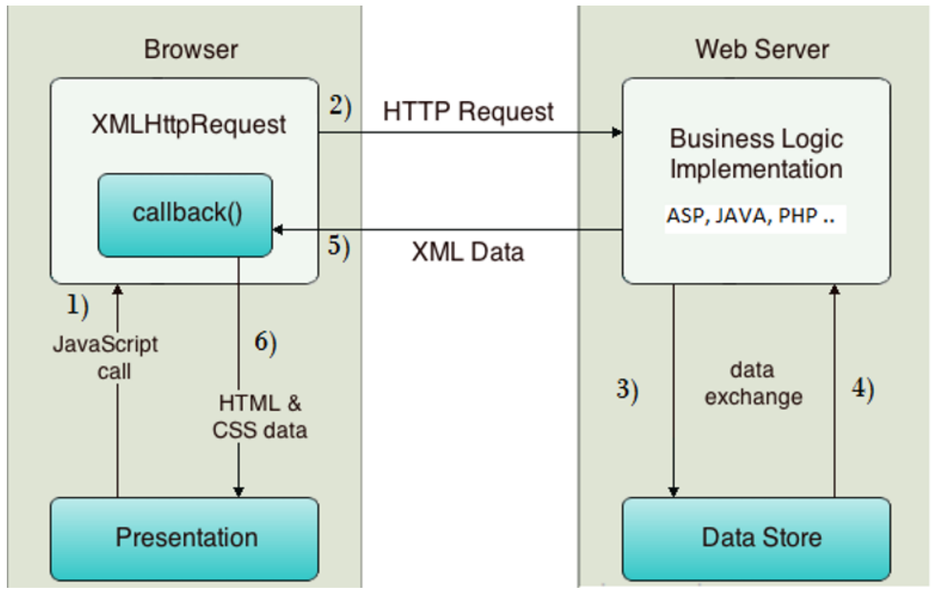
Cookies for GET/POST Requests

- 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

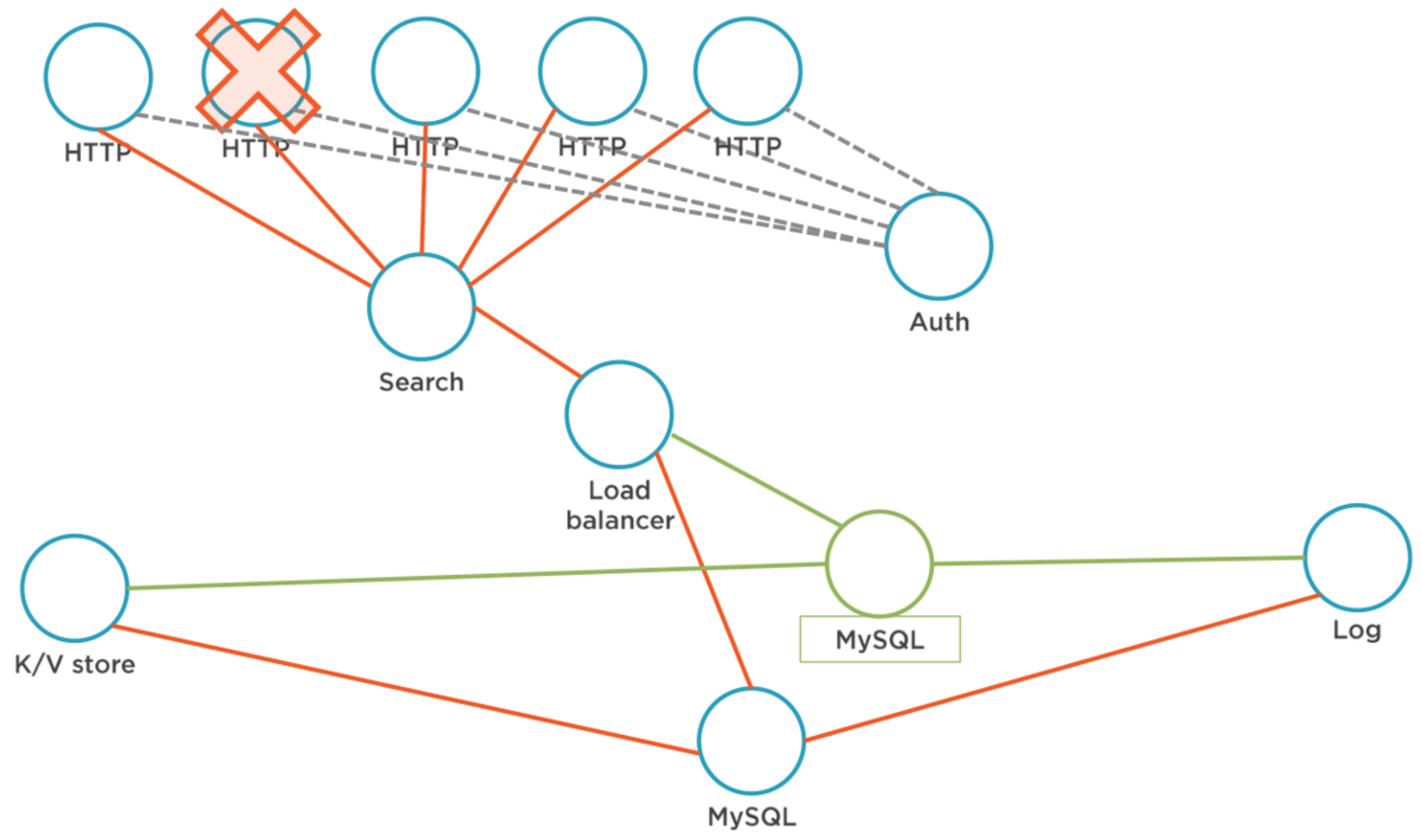


AJAX Calls

- 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



Modern Web Applications



6

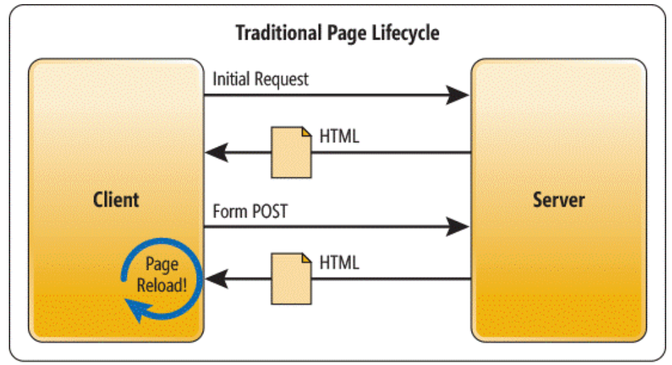
PREV NEXT

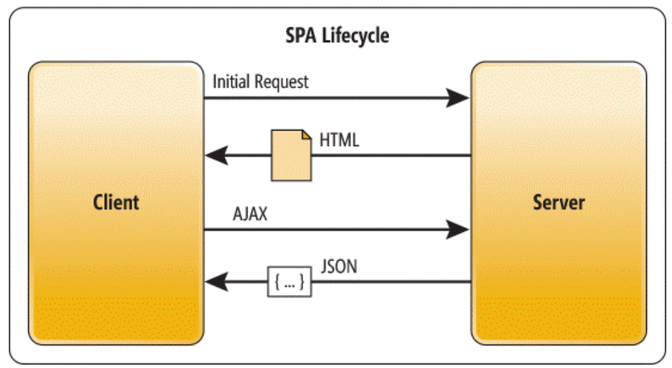
OUTLINE

Search...

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





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)

PREV NEXT

OUTLINE

Search...

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)

PREV NEXT

OUTLINE

Search...

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() Deletes an existing entity
- **OPTIONS** app.options() Retrieves the capabilities of the server

PREV NEXT

Idempotent vs. Safe Methods

- **Safe methods** are HTTP methods that do not modify resources.
- An **idempotent HTTP method** is a HTTP method that can be called many times without different outcomes.

HTTP Method	Idempotent	Safe
OPTIONS	yes	yes
GET	yes	yes
HEAD	yes	yes
PUT	yes	no
POST	no	no
DELETE	yes	no
PATCH	no	no

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.

PREV NEXT

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

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.

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 2018 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 Dec 2018 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 Dec 2018 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.

Request Throttling

Throttling (rate limiting), is a technique to limit the number of requests a client can make to your API in a certain window of time. You will reset the quota after a certain period of time. It's up to you to decide how to implement such limiting (per IP address, per users authentication).

The **`X-RateLimit-Reset`** header should contain a UNIX timestamp describing the moment when the limit will be reset:

First Respons

```
X-RateLimit-Limit: 2000
X-RateLimit-Remaining: 1999
X-RateLimit-Reset: 1404429213925
```

Once the request quota is drained, the API should return a `429 Too Many Requests` response:

```
HTTP/1.1 429 Too Many Requests
X-RateLimit-Limit: 2000
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1404429213925
```

Security / HTTPS configuration

There are certain performance advantages that become available to you once you serve your content over a secure connection:

HTTP/2

Multiplexing streams, Header compression, Request prioritization, Server push

Brotli Compression

Brotli is a new compression algorithm that has the potential to outperform gzip and also supports static compression, so you don't need to compress assets on the fly.

SEO

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

PREV NEXT

Security Headers

- **CSP Content Security Policy**

  `content-security-policy:[CSP_POLICY]`

- **PKP Public Key Pins**

  `public-key-pins:[PKP_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.googleapi.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
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

PKP Public Key Pins

HTTP Public Key Pinning (HPKP) is a security feature that tells a web client to **associate a specific cryptographic public key with a certain web server** to decrease the risk of MITM attacks with forged certificates.

```
public-key-pins:pin-sha256="8B2DA066BBE619D882B7AE5E437CFBFC1FAD6118E9FC42EF1FF8DFB53B23E55A";
                max-age=5184000; includeSubDomains;
```

pin-sha256 contains the server public key used in production. **max-age=5184000** tells the client to store this information for two months, which is a reasonable time limit according to the IETF RFC. The key pinning is also valid for all subdomains, which is told by the **includeSubDomains** declaration.

```
<script src="https://code.jquery.com/jquery-3.3.1.js"
        integrity="sha256-2Kok7MbOyxpgUVvAk/HJ2jigOSYS2auK4Pfzbm7uH60="
        crossorigin="anonymous"></script>
```

The "**anonymous**" keyword means that there will be no exchange of user credentials.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning

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 sub domains that can only be served over HTTP.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

HTTP Access Control

For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts. For example, `XMLHttpRequest` follows the same-origin policy. So, a web application using `XMLHttpRequest` could only make HTTP requests to its own domain.

If you're building an application (a REST API server) that serves requests coming from front-end clients hosted on different domains, you might encounter cross-domain limitations when making XHR/AJAX calls. The workaround is to use:

1. **Web proxy**
2. **JSONP** - Express.js has a **`res.jsonp()`** method that makes JSONP a breeze.
3. Cross-Origin Resource Sharing (**CORS**) headers on the server.

https://developer.mozilla.org

Same-Origin Policy

The same-origin policy permits scripts running in a browser to only make XHR requests to pages on the same domain. This means that requests must have the same URI scheme, hostname, and port number.

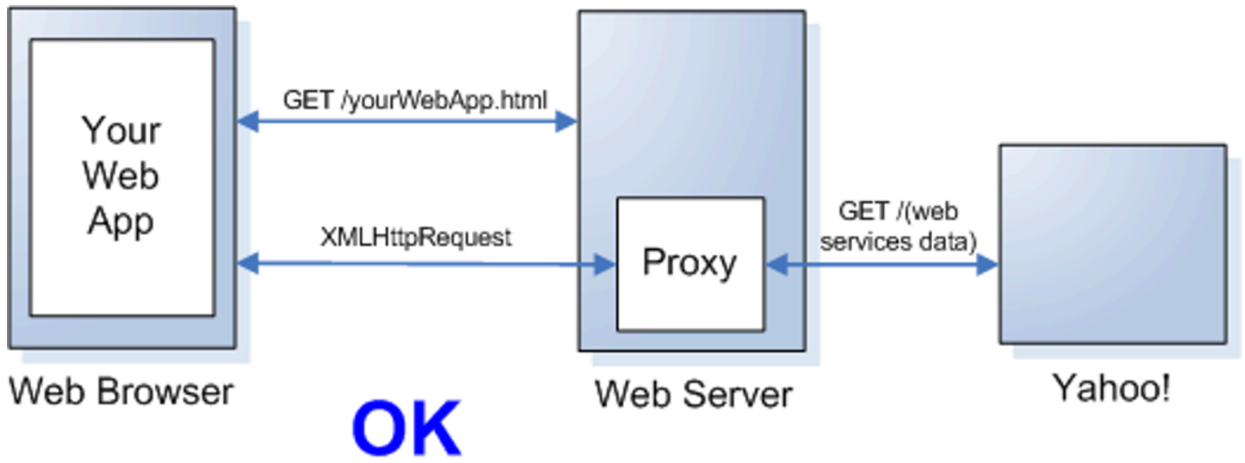
If you send an XHR request from `http://www.mysite.com`, the following types of requests result in failure:

- `https://www.mysite.com` – Different protocol (or URI scheme).
- `http://www.mysite.com:8080/myUrl` – Different port (since HTTP requests run on port 80 by default).
- `http://www.myothersite.com/` – Different domain.
- `http://mysite.com/` – Treated as a different domain as it requires the exact match (Notice there is no www.).

PREV NEXT

Using a Web Proxy

Instead of sending an XHR request directly from your domain (http://www.mysite.com) to a new domain (http://www.alienSite.com) which is not allowed by default, you send an XHR request to your own server (http://www.mysite.com/?connectTo=alienSite.com), which in turns sends a regular request to the foreign domain (http://www.alienSite.com/). To the browser, it appears you are exchanging the data with your own server. In reality, in the background, you have accessed data on a alien domain from your server.



Yahoo Developers

JSONP

Another way of implementing cross browser requests is by using JSONP, or **JSON with padding**.

JSONP takes advantage of the fact that <script> tags are not subject to the same-origin policy.

JSONP requests are made by dynamically requesting a <script> tag. The interesting part is that **the response is JSON wrapped in a function call**.

When making the request, **you specify the function name** as a callback function. When the server responds, the callback function (which must exist on your page) is executed with the data returned from the server.

Why NOT To Use JSONP?

- The only HTTP method you can use is **GET**. You cannot connect to a RESTful APIs that use other HTTP verbs to do fun stuff like CRUD.
- Since you are relying on a script tag, **Error handling** with JSONP is tricky or impossible.
- JSONP only supports cookies for **authorization** (cannot customize header) while CORS supports many types of authorization (OAuth).
- Finally, keep in mind that JSONP has a security vulnerability. Since the `<script>` tag doesn't respect the same-origin policy, it is possible for a malicious website to get sensitive data using the same URL. **Cross Site Request Forgery** CSRF attack.

Cross-Origin Resource Sharing (CORS)

- This solution allows the full array of HTTP verbs and errors
- Cross Domain Resource Sharing, works by modifying HTTP headers in your requests to access resources on a different domain.
- Most of the heavy lifting for CORS is handled between the browser and the server. The browser adds some additional headers, and makes additional requests, during a CORS request on behalf of the client. These additions are hidden from the client (but can be discovered using a packet analyzer such as *Wireshark*).

PREV NEXT

Simple Requests vs Preflight Requests

- **Simple requests** are characterized as such they can be made from a browser without using CORS. *(For example, a JSONP request can issue a cross-domain GET request. Or HTML could be used to do a form POST).*
- Any request that does not meet the criteria above is a not-so-simple request, and requires a little extra communication between the browser and the server (called a **Preflight Request**).

www.html5rocks.com

How does CORS work?

The Cross-Origin Resource Sharing standard works by adding new HTTP headers that **allow servers** to describe the set of **origins** that are permitted to read/write that information using a web browser. The specification mandates that browsers **preflight** the request, asking for the supported methods from the server with an HTTP OPTIONS request method, and then, upon "approval" from the server, sending the actual request with the actual HTTP request method. Servers can also notify clients whether credentials should be sent with requests.

PREV NEXT

Simple Request

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

```
fetch('http://bar.other/API/public/')
   .then(data => console.log('success: '+ data));
```

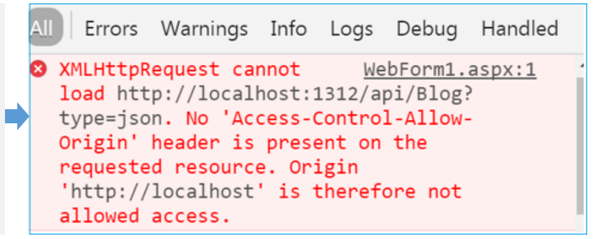
1 Request

```
GET /API/public/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh)
Accept: text/html,application/xml;
Accept-Language: en-us,en;
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;
Connection: keep-alive
Referrer: http://foo.example/page.html
Origin: http://foo.example
```

2 Response

```
HTTP/1.1 403 FORBIDDEN
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: none
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
```

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
```



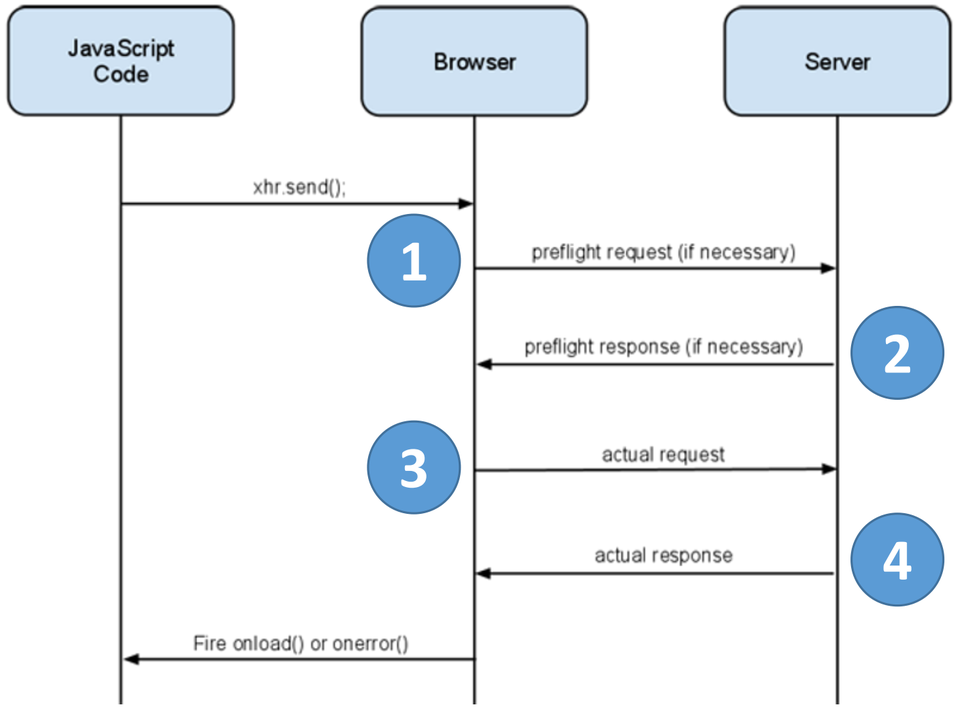
Preflighted Requests

Preflighted requests first send an HTTP request by the OPTIONS method to the resource on the other domain, in order to determine whether the actual request is safe to send. A request is preflighted if:

- It uses methods other than GET, HEAD or POST.
- If POST is used to send request data with a Content-Type other than application/x-www-form-urlencoded, multipart/form-data, or text/plain, e.g. if the POST request sends an XML payload to the server using application/xml or text/xml, then the request is preflighted.
- It sets custom headers in the request (the request uses a header such as X-PING-OTHER)

PREV NEXT

Cross-Origin Resource Sharing (CORS)



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.

46

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

Express

Express.js is a web framework based on the core Node.js **`http`** module and **`connect`** components.

express

PREV NEXT

OUTLINE

Search...

33. Simple Requests vs Preflight Requests

34. How does CORS work?

35. Simple Request

36. Preflighted Requests

37. Cross-Origin Resource Sharing (CORS)

38. Preflighted Request

39. Preflighted Request

40. Preflighted Request

41. Express

Layered Architecture

- **Routing layer**: controller logic and API request handling.
- **Service layer**: business logic concerns, composed of modular components that handle a piece of the business logic.
- **Data layer**: unified data access models, communicates with the service layer using data models.

Features

- Parsing HTTP request bodies
- Extracting URL parameter
- Parsing cookies
- Managing sessions
- Organizing routes with a chain of `if` conditions based on URL paths and HTTP methods of the requests
- Determining proper response headers based on data types
- Handling errors

Installation & Creating a Web Server

```
$ npm install express
```

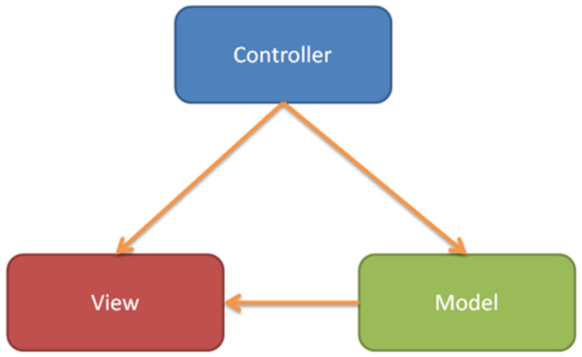
```
var express = require('express');
var app = express();
app.listen(3000);
```

Express.js Generator (express-generator) is a separate module. It allows for rapid app creation because its scaffolding mechanism.

Using the Generator

```
$ npx express-generator MyApp
              OR
$ npm I express-generator -g
$ express MyApp

$ cd MyApp
$ npm install
```




```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```

Watching for File Changes

The following file-watching tools can leverage the watch() method from the core Node.js fs module and restart our servers when we save changes from an editor.

- **forever** https://npmjs.org/package/forever
- **node-dev** https://npmjs.org/package/node-dev
- **nodemon** https://npmjs.org/package/nodemon
- **supervisor** https://npmjs.org/package *Written by the creators of NPM*
- **up** https://npmjs.org/package/up *Written by the Express.js team*

Express Application Structure

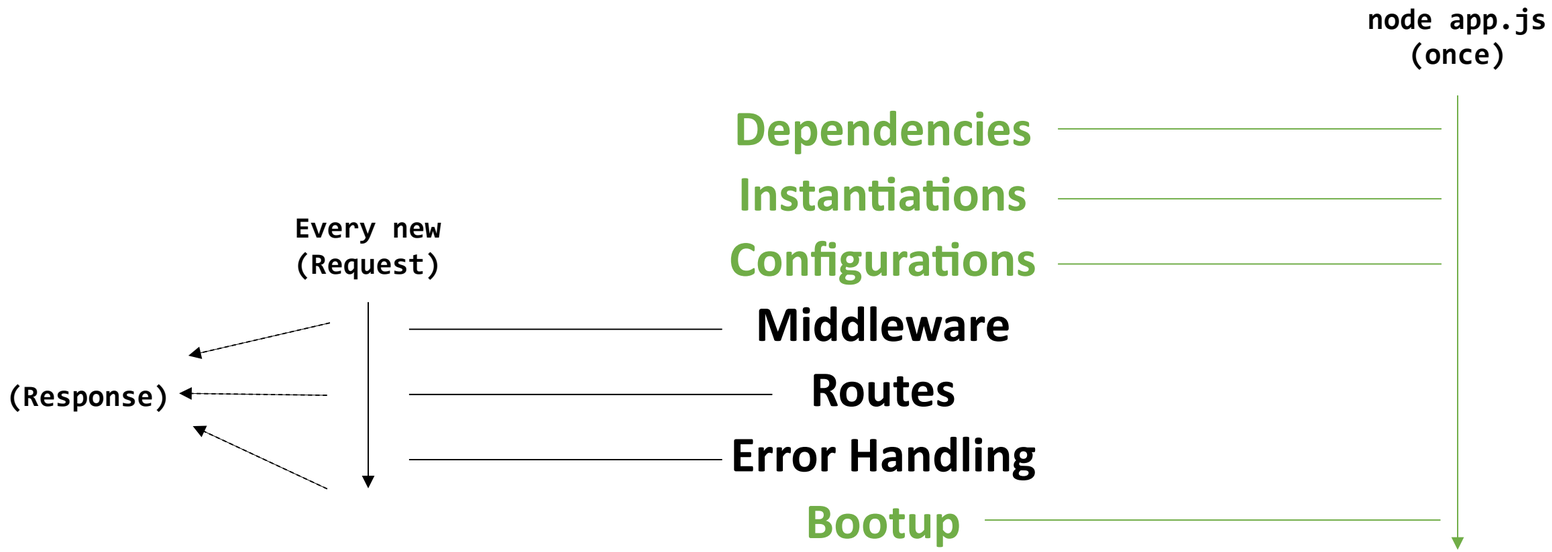
The typical structure of an Express.js app (which is usually app.js file) roughly consists of these parts, in **the order shown**:

workflow

1. **Dependencies**
2. **Instantiations**
3. **Configurations**
4. **Middleware**
5. **Routes**
6. **Error Handling**
7. **Bootup**

PREV NEXT

How Does Express App Work?



Your First Express App

```
var express = require('express');
var app = express();
var port = 3000;

app.get('*', function(request, response){
        response.status(200);
        response.set('Content-Type', 'text/html');
        response.send('Hi');
        response.end();
});
app.listen(port, function(){
        console.log('The server is running on port %s', port);
});
```

The app.get() function supports **Regular Expressions** of the URL patterns in a string format.
The second parameter to the app.get() method is a **Request Handler** *(a function that will be executed every time the server receives a particular Request).*

Configurations and Settings

Setting your Express application properly is **VERY important** especially that you are going to use it for building your **Restful API.**

There are two ways to configure our application:

1. **set**

```
app.set('port', process.env.PORT || 3000);
const port = app.get('port');
```

2. **enable/disable**

```
app.enable('etag') === app.set('etag', true)
app.disable('etag') === app.set('etag', false)
```

Environments

During development, the app error messaging needs to be as verbose as possible, while in production it needs to be user friendly not to compromise any system or user's Personally Identifiable Information (PII) data to hackers. https://en.wikipedia.org/wiki/Personally_identifiable_information

Most of the settings we will learn apply to all backend web applications (Java, C#, PHP.. etc)

Settings – 'env'

```
app.set('env', 'development');
console.log(app.get('env'));
```

The most common values for `env` setting are:

- `development`
- `test`
- `stage`
- `preview`
- `production`

The better way is to start an app with
`$ NODE_ENV=preview node app`

Or to set the **`NODE_ENV`** variable on the machine.

Knowing in what mode the application runs is very important because logic related to error handling, compilation of style sheets, and rendering of the templates can differ dramatically. Not to mention that databases and hostnames are different from environment to environment.

Settings – 'trust proxy'

Set trust proxy to `true` if your Node.js app is working behind reverse proxy such as Varnish or Nginx (or any load balancer). This setting will permit trusting in the `X-Forwarded-*` headers. The trust proxy setting is disabled by default.

```
app.set('trust proxy', true);
app.enable('trust proxy');
```

The `X-Forwarded-*` request header helps you identify the IP address, Protocol and Port of a client when you use an HTTP or HTTPS load balancer. **Because load balancers intercept traffic between clients and servers, your server access logs contain only the IP address of the load balancer**. To see the IP address of the client, use the X-Forwarded-For request header.
Example: `X-Forwarded-For: 203.0.113.7`
`X-Forwarded-Proto: 203.0.113.7`

Settings – 'case sensitive routing'

To disregard the case of the URL paths set this setting to `false`, which is the default value, and do otherwise when the value is set to true.

```
app.enable('case sensitive routing');
```

For example, when it’s enabled, then **/users** and **/Users** won’t be the same. It’s best to leave this option disabled by detault for the sake of avoiding confusion.

Settings – 'strict routing'

It deals with cases of trailing slashes in URLs.

```
app.set('strict routing', true');

app.get('/users', function(request, response){
        response.send('users');
})
app.get('/users/', function(request, response){
        response.send('users/');
})
```

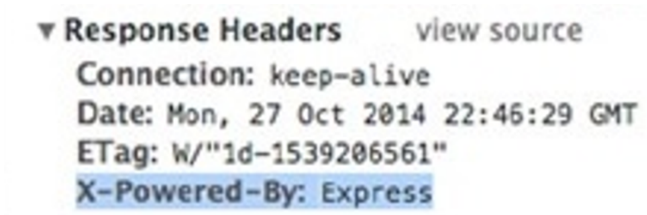
/users and **/users/** will be completely separate routes. By default, this parameter is set to `false`, which means that the trailing slash is ignored and those routes with a trailing slash will be treated the same as their counterparts without a trailing slash.

Settings- x-powered-by

The `x-powered-by` option sets the HTTP response header `X-Powered-By` to the `Express` value. This option is enabled by default.

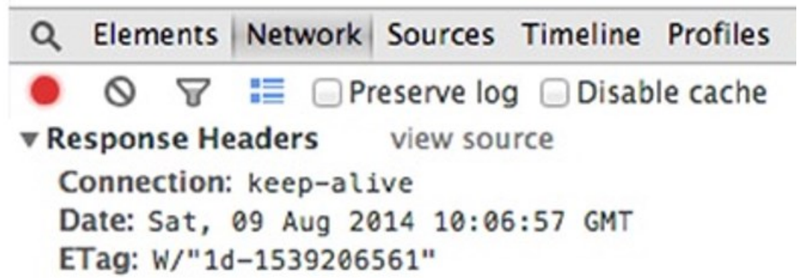
If you want to disable `x-powered-by` (remove it from the response)—which is **recommended for security reasons**, because it's harder to find vulnerabilities if your platform is unknown.

```
app.set('x-powered-by', false)
```



Settings – 'etag'

ETag (Entity Tag) is a **caching tool**. The way it works is similar to the **unique identifier for the content on a given URL**. In other words, if content doesn't change on a specific URL, the ETag will remain the same and the browser will use the cache.



Default value is: `true` (weak ETag), `false` (no ETag), and `strong` (strong ETag).

```
app.disable('etag'); // will eliminate the ETag HTTP response header
app.set('etag', function (body, encoding) {
        return customEtag(body, encoding); // using your own ETag algorithm
})
```

An identical **strong** ETag guarantees the response is byte-for-byte the same. An identical **weak** ETag indicates that the response is semantically the same. So you'll get different levels of caching with weak and strong ETags.

Settings – 'jsonp callback name'

The default callback name, which is a prefix for our JSONP response, is usually provided in the query string of the request with the name callback

```
<script src="http://www.myothersite.com/get_data?callback=updateView"></script>
```

However, if you want to use something different, just set the setting `'jsonp callback name'` to another value, for example, for the requests with a query string param **`?cb=updateView`**, we can use this setting:

```
app.set('jsonp callback name', 'cb');
<script src="http://www.myothersite.com/get_data?cb=updateView"></script>
res.jsonp({ user: 'Asaad' });  // => updateView({ "user": "Asaad" })
```

Handling JSONP in Express

```
<script type="text/javascript">
        var url = 'http://domain:3000/api?callback=myFunc?userID=5';
        function myFunc(data){...}
        fetch(url).then( data => console.log(JSON.stringify(data)) );
</script>
// X GET http://domain:3000/api?callback=myFunc&userID=5
```

```
app.set('jsonp callback name', 'callback');
app.get('/api', function(req, res){
   let out;
   req.query.userID === 5)? out = { name: 'Asaad', course: 'CS572’ } : out = {}
   res.jsonp(obj)
});
// myFunc({ name: 'Asaad', course: 'CS572' })
```

74

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"
});
```


PREV NEXT

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"
```