Resources

Endpoint

$$\underbrace{\texttt{http}}_{\text{How to retrieve the resource}}\texttt{://}\underbrace{\texttt{acme.com}}_{\text{Container that holds some resource}}\underbrace{\texttt{/api/news/12345}}_{\text{Resource identifier unique within the container}}$$

$$\underbrace{\texttt{GET}}_{\text{Verb}}\ \underbrace{\texttt{/api/news/12345}}_{\text{Noun}}$$

Resource Types

- Types of resources
  - Collection - list of one or more resources
    - Eg. list of employees in a department, all the books by a particular author
  - Item - the resource
    - Eg. employee

Resource Name

- Use plurals for resource collection
  - Eg. `/api/employees`
- Individual resource should be uniquely identifiable
  - Should have a ‘primary key’
  - Eg. `/api/employee/1`

Resource Path

- Resource path are unique - like a primary key in data database
- Use to uniquely identify a specific resource or collection of resources
- Resource path can be have meaning unlike primary key
  - `/api/customer/1` - customer with id 1, returns a single resource
  - `/api/customer/1/orders` - list all of customer 1's orders - a collection of resources
- Sub resource may be used to define relationships between resources
  - `/api/customer/1/orders` is a sub resource of `/api/customer/1`
  - Relationship between the 2 resources is 'has'
  - Cardinality of the relationship is 1 to many

Resource Name

- Use the hierarchical nature of using URL as resource names to impose structure

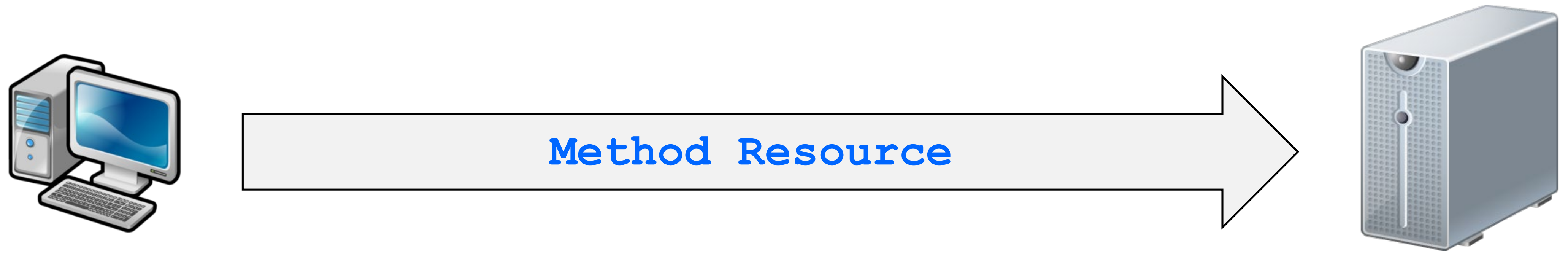
```
/api/customers
/api/customers/apac
/api/customer/1
/api/customer/1/orders
/api/customer/1/order/3
/api/customer/1/orders/today
```

Resource Name

- Use nouns for resource names
  - Eg. `/api/employee` not `/api/getEmployee`
  - Latter is RPC style
- Use HTTP methods to express intent on the resource

Verb	Noun
GET	/customer/1
POST	/customer
PUT	/customer/1
DELETE	/customer/1

Making Request



- Typical information passed from client to server
  - Client and/or route identification
  - Resource filter eg. only return a subset of the data
  - Content type eg. return the requested resource as PDF
  - Payload eg. if the request is uploading a file

Query String as Context

- Query string can be thought of as providing some context or filters to the resource
  - Eg. Find all January's purchase orders

```
/api/orders?month=jan&year=2019
```

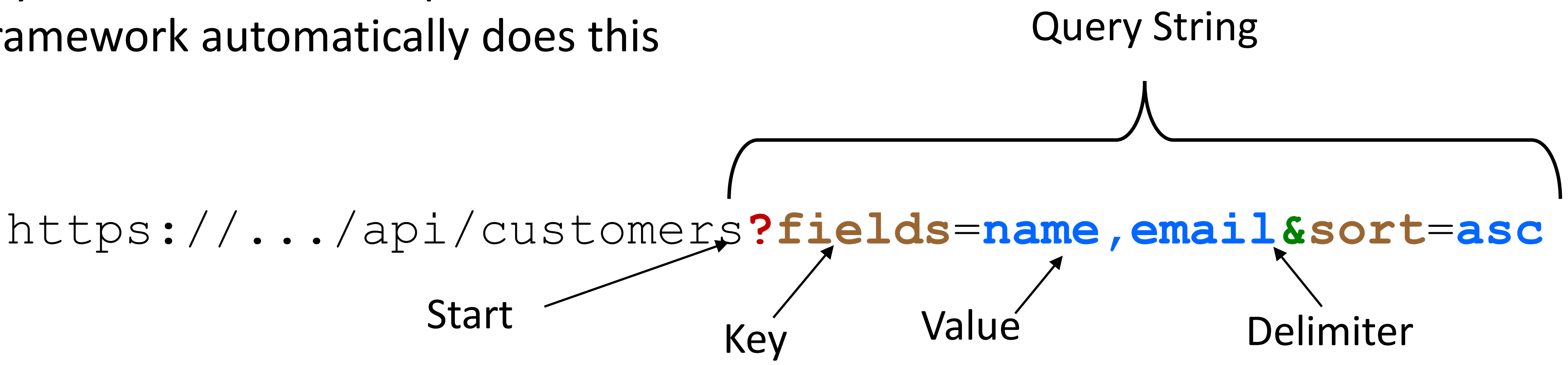
  - Eg. Find all male employees in engineering department

```
/api/engineering/employees?gender=male
```

- Bookmarkable especially if it is a GET
  - Eg. saving a search and sharing it

Query String

- Key value pair that is part of the URL
  - Key value must be escaped with `encodeURI()`
  - Framework automatically does this



```
https://.../api/customers?fields=name,email&sort=asc
```

- Used to provide additional information to the resource
  - To narrow the result
- Mostly used in GET but applicable to other methods where it makes sense
  - Eg delete

Query String

```
<form method="GET" action="/search">
   <input type="text" name="q">
   <button type="submit">
      Search
   <button>
</form>
```

```
const qs = new HttpParams()
      .set('q', aValue);
this.http.get('/search',{ params: qs })
   .toPromise()
   .then(() => { ... })
```

```
GET /search?q=bloom
```

```
app.get('/search', (req, resp) => {
   const q = req.query.q;
   //Do something with q
});
```

Multivalued Query Strings

- Collections are use to hold data belonging to the same category
  - Eg. hobbies - swimming, reading, travelling
- Query string's key/value pair is for holding only a single value
- Workaround to allow for multiple values using libraries
  - Repeat the key/value pairs

```
?hobby=swing&hobby=read&hobby=travel&email=fred@bedrock.com
```

  - Delimit the value for multiple values

```
?hobby=swing,read,travel&email=fred@bedrock.com
```

  - Special syntax

```
?hobby[]=swing&hobby[]=read&hobby[]=travel&email=fred@bedrock.com
```

Common Uses for Query String

- Searching

```
/api/engineering/employees?name=fred
```

- Paging a collection

```
/api/engineering/employees?offset=10&limit=20
```

- Filtering

```
/api/engineering/employees?gender=male
```

- Provide additional information eg. identity

```
/api/engineering/employees?client_id=abc123
```

Special Resource Suffix

- Used reserved words as suffixes for resource’s path segment
  - Replace query string for common use cases
  - More meaningful and easier to remember
- Examples
  - Latest purchase order

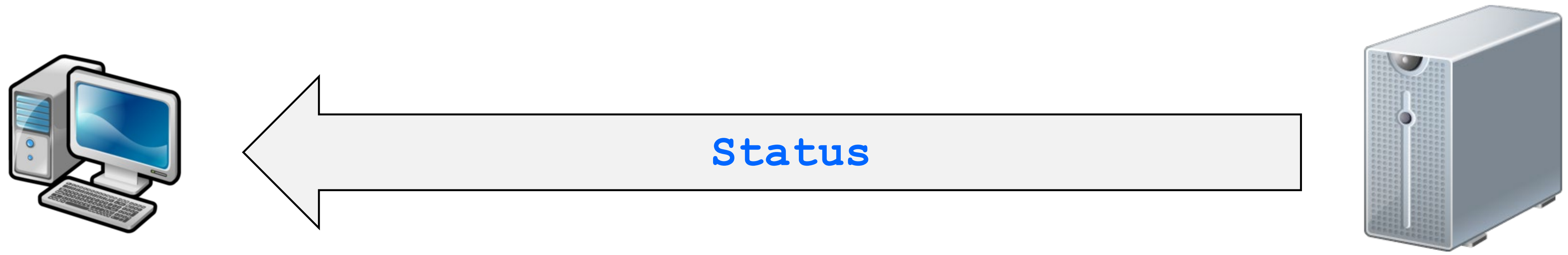
    `/api/orders/today`
  - Searching a department for employees

    `/api/engineering/employees/search?q=fred`
  - Special locations in collections

    `/api/books/genre/thriller/first`
  - Stock quantity

    `/api/item/abc123/count`

Sending Response



- Typical response passed from server to client
  - Information about the server
  - Caching information
  - Payload’s representation
  - Payload
  - Compression information

Status Code

- Important to use the correct status code
  - Allow JavaScript clients to respond to result accordingly
  - Eg. Status code 400/500 will cause the promise to be rejected

```
this.http.get('/api/customers')
  .toPromise()
  .then(result => {
    //Do something with the result
  })
  .catch(error => {
    //Handle the error
  })
```

Between 200 and 299

Greater than 400

Response - Data Type

- Numbers - float and integer
- Boolean
- Strings - quoted, JSON uses " rather than '
- Date/time/timestamp - need to standardize because of different timezones. Use one of the following
  - ISO 8601

    ```
    const isoTime = (new Date()).toISOString()
    ```
  - Unix Epoch - milliseconds since Jan 1 1970

    ```
    const unixEpoch = (new Date()).getTime()
    ```
  - UTC/GMT format

    ```
    const utc = (new Date()).toUTCString()
    ```

Content Type

- Set the appropriate MIME type for the result
  - Different MIME type may cause the client to handle the response differently
- HTTP header `Content-Type` specifies the representation of the payload

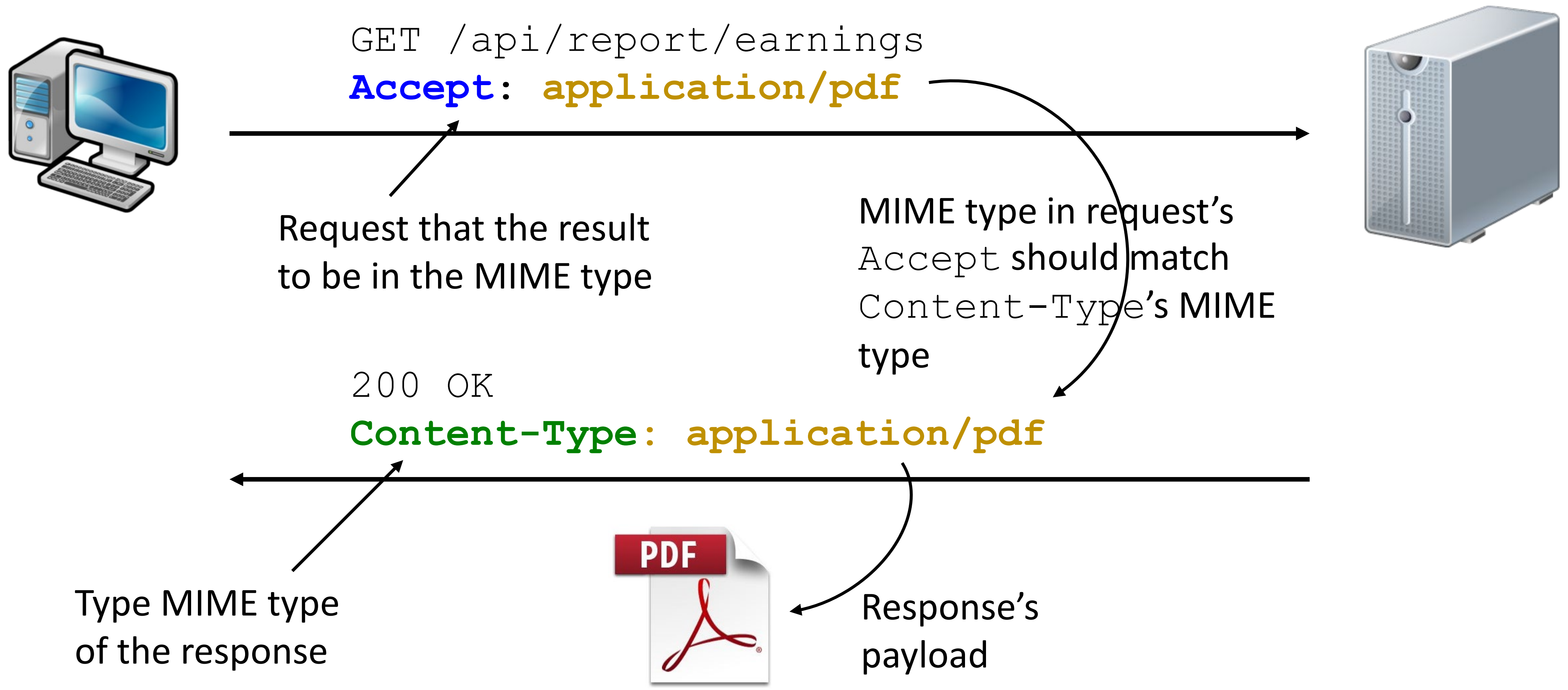
```
app.get('/api/time', (req, resp) => {
   const time = (new Date()).toISOString();
   resp.status(200).type('text/plain');
   resp.send(`<h2>The current time is ${time}</h2>`);
})
```

Setting the `Content-Type` to text/plain will cause the browser to treat the response as text even thought it is a HTML

Content Negotiation

- One of the constraint of uniform interface
- Serve different representation for the same resource
- Negotiate
  - Encoding - HTML, PDF, text, JSON
  - Version - older or newer version of the resource
- To allow resources to be consumed by
  - Different types of clients - eg. browser, Android application, JavaScript
  - Different versions of clients - eg. legacy or latest and greatest

`Accept` and `Content-Type` Headers



```
GET /api/report/earnings
Accept: application/pdf
```

Request that the result to be in the MIME type

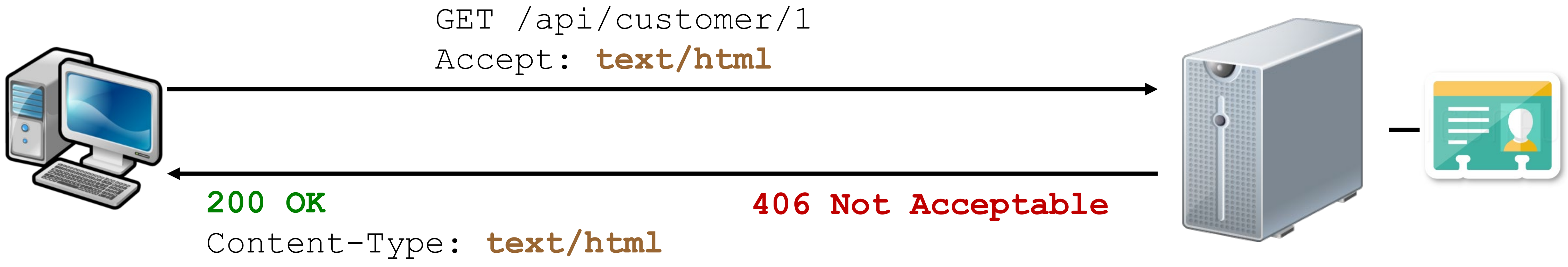
MIME type in request's `Accept` should match `Content-Type`'s MIME type

```
200 OK
Content-Type: application/pdf
```

Type MIME type of the response

Response's payload

Content Negotiation



- Use HTTP headers `Accept` to request for the specific a specific representation
- Common representation
  - `application/json`, `application/csv`, `text/html`
- Response with
  - `200` if the resource can be returned in the requested representation
  - `406` if the resource does not have a representation that is acceptable to the client
  - `Content-Type` header that matches the request's `Accept`

Content Negotiation

- Specific representation/MIME type separated by comma

```
Accept: image/png, image/jpeg
```

- Any type of representation

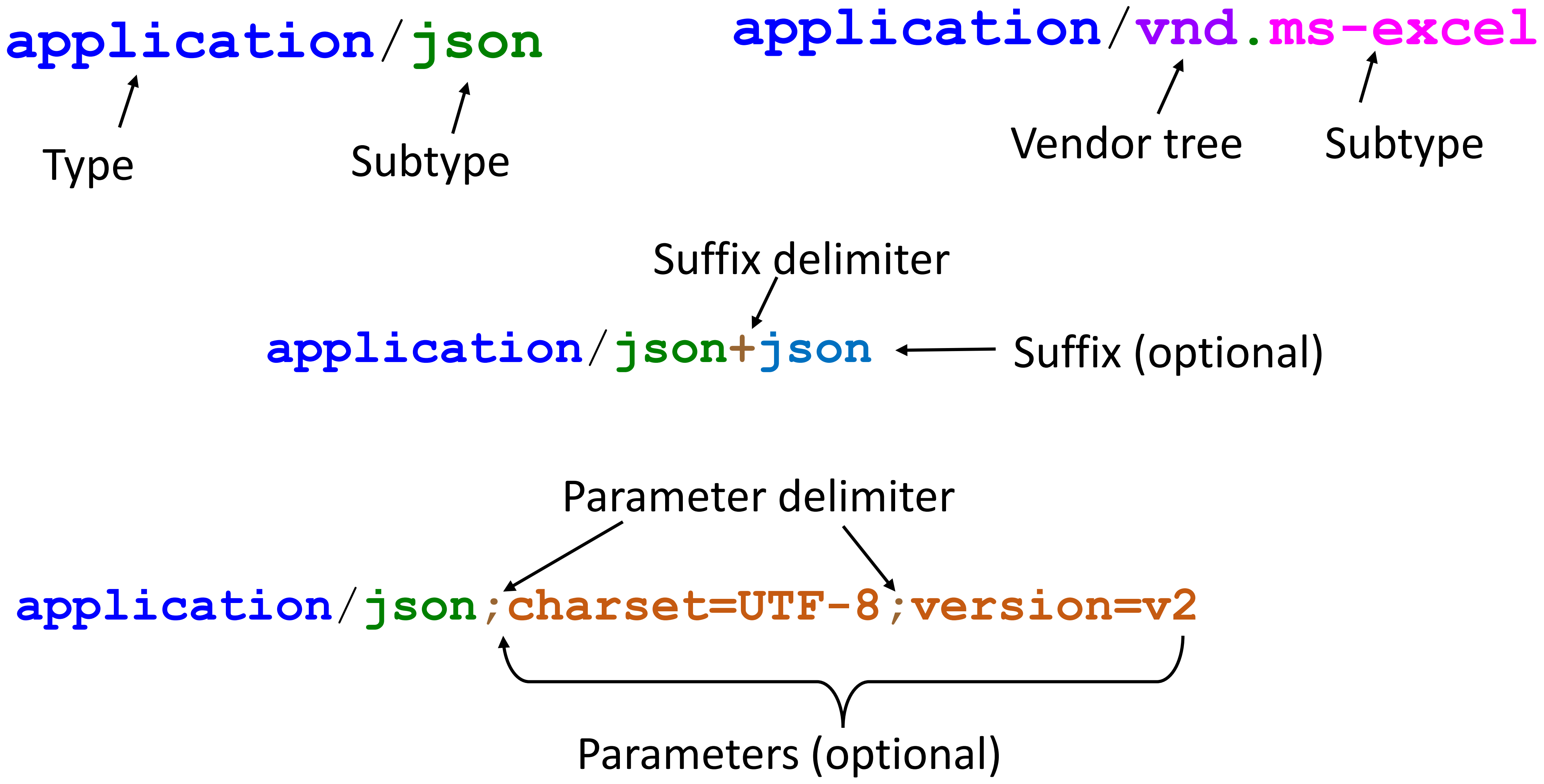
```
Accept: image/*
```

- Preference for a particular representation
  - `q` value between 0 and 1, the higher
  - Not specified means `q=1.0`

```
Accept: image/png, image/jpeg;q=0.8, image/gif;q=0.5
```

- Use the wild card `*/*` representation as a 'catch-all'
  - Allow the resource to return its preferred type

MIME Type



```
application/json
application/vnd.ms-excel
application/json+json
application/json;charset=UTF-8;version=v2
```

Example - Content Negotiation

Express

```
app.get('/customers', (req, resp) => {

   const mime = req.accepts(['application/json', 'text/html']);

   if (!mime)
      return resp.status(406).end();

   const data = //Retrieve data
   resp.status(200);
   switch (mime) {
      case: 'application/json':
         resp.json({ ... });
         break;
      case: 'text/html':
         resp.type('text/html').send(...);
         break;
      default:
         resp.type('text/plain').send(...);
   }
});
```

`accepts()` computes the Accept preferences. Returns `false` if none found

Return a 406 status if cannot produce the requested representation

Produce the required representation

Request Payload

- Request may contain a payload
  - Eg. file upload, sending some data to the server
- Most common HTTP method is POST
  - Use `Content-Type` to let the server know the payload’s representation
    - What is the payload’s MIME type
  - Eg. by default Angular’s `HttpClient` sends data in JSON representation
- Common request payload representation
  - `application/json` - JSON
  - `application/x-www-form-urlencoded` - a HTML form submitted by POST
  - `multipart/form-data` - typically used for uploading files or large amount of data
    - Not efficient for form fields as the boundary may be larger the than the actual data

Example - `json`

```
const data = { name: 'fred', email: 'fred@bedrock.com' }
this.http.post('/api/customers', data)
```

Angular uses JSON for sending data

```
POST /api/customers
Content-Type: application/json

{ "name": "fred", "email": "fred@bedrock.com" }
```

```
app.use(bodyParser.json());

app.post('/api/customers', (req, resp) => {
  const payload = req.body;
  ...
});
```

Express middleware `body-parser` parses JSON payload

Example - x-www-form-urlencoded

```
<form method="POST" action="/api/customer">
  <input type="text" name="name">
  <input type="email" name="email">
  <button type="submit">Submit</button>
</form>
```

```
const data = new HttpParams()
     .set('name', 'fred')
     .set('email', 'fred@bedrock.com');
const headers = new HttpHeaders()
     .set('Content-Type', 'application/x-www-form-urlencoded');
this.http.post('/api/customer',
     data.toString(), { headers: headers })
```

Example - x-www-form-urlencoded

```
POST /api/customers
Content-Type: application/x-www-form-urlencoded

name=fred&email=fred@bedrock.com
```

Form data is encoded like a query string viz. key/value pairs. Part of the request's body

Express

```
app.use(bodyParser.urlencoded({extended: true}));

app.post('/api/customers', (req, resp) => {
   const payload = req.body;
   ...
});
```

Example - multipart/form-data

```
<form method="POST" action="/api/invoice"
    enctype="multipart-form-data">
  <input type="file" name="invoice">
  <button type="submit">Submit</button>
</form>
```

```
upload($event) {
  const data = new FormData();
  data.set('invoice', $event.target.files[0]);
  this.http.post('/api/invoice', data)
}
```

Example - multipart/form-data

Boundary of the body parts

```
POST /api/invoice
Content-Type: multipart/form-data; boundary=random_string

--random_string
Content-Disposition: form-data; name=“invoice” filename=“foo.pdf”
Content-Type: application/pdf
... Contents of foo.pdf encoded in Base64 ...
```

One or more of these parts in the body of the request

Express middleware `multer to` parses multipart payload

```
app.post(‘/api/invoice’, multer.single(‘invoice’),
   (req, resp) => {
      const invoice = req.file;
      ...
   });
```

Contains the uploaded file information. See https://github.com/expressjs/multer#file-information

`Content-Type` for `POST`

- `application/json`
  - For richly structured data
  - Eg. data with embedded arrays and objects
- `application/x-www-form-urlencoded`
  - For tablular data
  - Eg. flat 2-D form information
- `multipart/form-data`
  - For binary objects/BLOB
  - Eg. files, images, videos

Response - Collection

- Response may contain/embedded resource
  - Eg. returning a collection of resource
  - Eg. purchase orders belonging to a particular customer
- Use URLs when referencing other resource rather then the actual resource
- URLs as a way to show relationships between resources
  - Maintain the ‘web spirit’ of linking things
  - Navigable from one resource to another via this relationships

```
[ { custId: 0, name: ‘ACME’ },
  { custId: 1, name: ‘Wayne’ },
  { custId: 2, name: ‘Yoyodyne’ },
     ...
]
```

References to resources rather than the actual resource

```
[ “/api/customer/0”,
  “/api/customer/1”,
  “/api/customer/2”,
     ...
]
```

Large Payload

- Optimize request or response when transferring large amount of data
- Request
  - Ensure that the web application is willing to receive a large upload
  - Eg. non-premium customers only allow to upload 20MB
- Response
  - Simple strategy is to implement paging/cursor
    - Specify the number of datum/records to return - limit
    - Where to start from - offset
  - Not possible to cursor if the response is a ‘blob’, eg. a large image
    - Compress response before sending
    - Encode the response in a more compact format
    - Stream the response allowing for

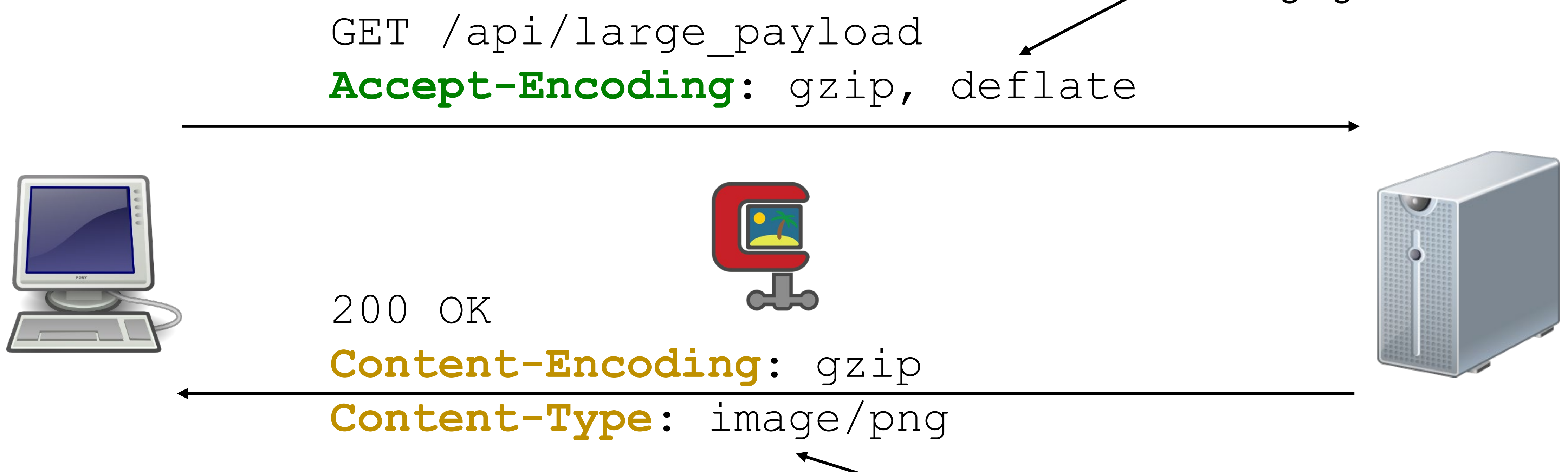
Large Payload

- Compression
- Use more compact data representation eg msgpack
  - Binary data are based64 encoded
- Stream the content
  - Instead of the store-and-forward method where the entire content is transfer as a large blob
- Allow partial responses
  - Present a cursor like object for the client to control the payload
  - Restart from a certain position if content were streamed

Compression

- Should use compression when payload is large
- Use `Content-Encoding` and `Accept-Encoding` headers
  - `Content-Encoding` specifies the payload's is compressed
  - `Accept-Encoding` let the server knows what algorithm the client will accept
- `Content-Encoding` header and `Accept-Encoding` accepts the following values for compression algorithm
  - Provide a comma separated string with one of the following algorithms
  - `gzip`, `compress`, `deflate`, `br`
  - `identity` indicates no compression
- Browser automatically decompresses response
  - But does not compress outgoing request

Example Compression



The client tells the server the supported encoding algorithms

```
GET /api/large_payload
Accept-Encoding: gzip, deflate
```

```
200 OK
Content-Encoding: gzip
Content-Type: image/png
```

Request's payload has been compressed with gzip

Example Compression

Set content type to application/octet-stream to prevent Angular from converting to JSON, the default Angular content type

Request's payload has been compressed with gzip

```
const headers = new HttpHeaders()
     .set('Content-Encoding', 'gzip')
     .set('Content-Type', 'application/octet-stream')
     .set('Accept-Encoding', 'gzip');

const compressData = pako.gzip( /* bytes */ )

this.http.post('/api/large_payload',
     compressData.buffer,
     { headers: headers }
).toPromise()
.then(result => {
  //The uncompressed data
})
```

Request the server to compress response with gzip

pako package
https://www.npmjs.com/package/pako

Send the compressed data as bytes array

Example adapted from https://stackoverflow.com/questions/51942451/compress-outgoing-requests-in-angular-2

Example Compression

compression middleware for Express
https://github.com/expressjs/compression

Express

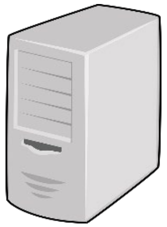
```
const compression = require('compression');
...
const app = express();

app.get('/api/large_payload, compression(),
   (req, resp) => {
     ...
     resp.status(200).type('image/png')
     resp.send(...);
   })
```

compression package will automatically deflate and compress incoming/outgoing payload

Partial Response

- Responses with large payload maybe interrupted
  - Eg over unstable connection like mobile data
- Streaming large media files
  - Request files in chunks rather getting the entire file which may be too large for the client
- HTTP `Accept-Ranges` header allows response to resume at a certain point
  - Client resume a response by using the `Range` header in a request
  - Values for `Accept-Ranges` may be `bytes` or `items` or any application units
- HTTP `Content-Range` is used in response to confirm the requested datum

Example Partial Response



```
HEAD /api/movie/abc123
```

1 Use the `HEAD` method to retrieve the HTTP header only.
Purpose is to check if the server supports partial responses

```
200 OK
Content-Type: video/x-matroska
Accept-Ranges: bytes
```

2 Server confirms that it can support partial responses with `Accept-Ranges` header.
Since this is a response to `HEAD`, no payload is sent

```
GET /api/movie/abc123
Range: bytes=0-1048575
```

3 Client request for the first MB

```
206 Partial Content
Content-Type: video/x-matroska
Accept-Ranges: bytes
Content-Range: bytes=0-1048575/1073741824
```

4 Server returns the first MB. The number after the slash is the total content size

Example Range

```
import { Range } from ‘http-range’;

getRecords(start=0, end=20): Promise<Customer[]> {
  const headers = new HttpHeaders()
        .set(‘Range’,
              (new Range(‘items’, `${start}-${end}`).toString()));

  this.get<Promise[]>(‘http://.../api/customers’, { headers: headers })
    .then(result => {
      //do something with result
    })
}
```

Request units

Construct the required range for the items

```
GET /api/customers
Accept: application/json
Range: items=0-20
```

Example Range

Express

```
const range = require('express-range');

app.get('/api/customers',
   range({ accept: 'items', limit: 20 }),
   (req, resp) => {
      Promise.all([
         getItems(req.range.first, req.range.last), getItemCount() ])
         .then(result => {
            resp.status(206);
            resp.type('application/json');
            resp.range({
               first: req.range.first,
               last: req.range.last,
               length: result[1],
            });
            resp.json(resul[0]);
         });
   }
);
```

Accept items and limit it to 20 if no last element is specified

Parses the HTTP headers for range values or use defaults if not found

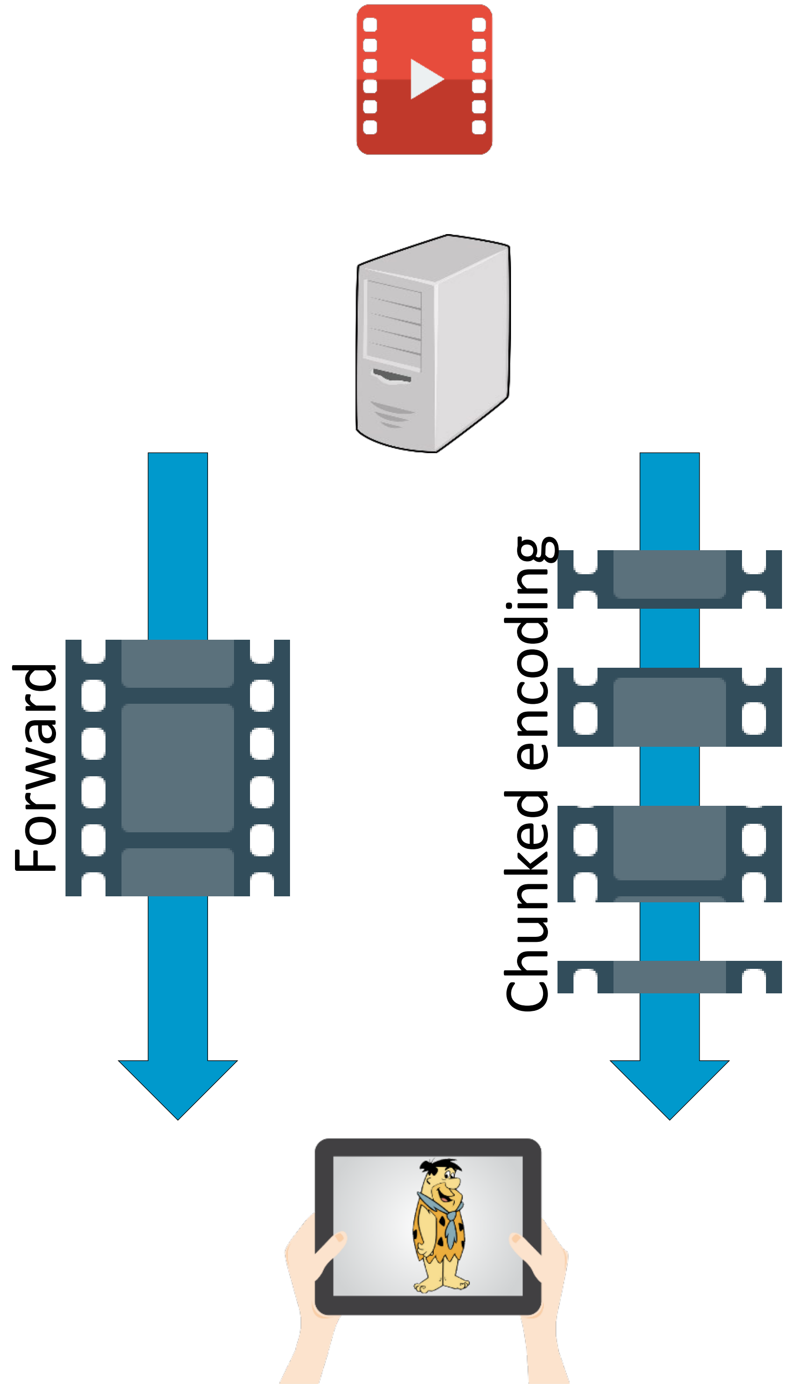
Partial response

Construct the `Content-Range` HTTP header

```
206 Partial Content
Content-Type: application/json
Content-Range: items=0-20/100
```

Chunked Encoding

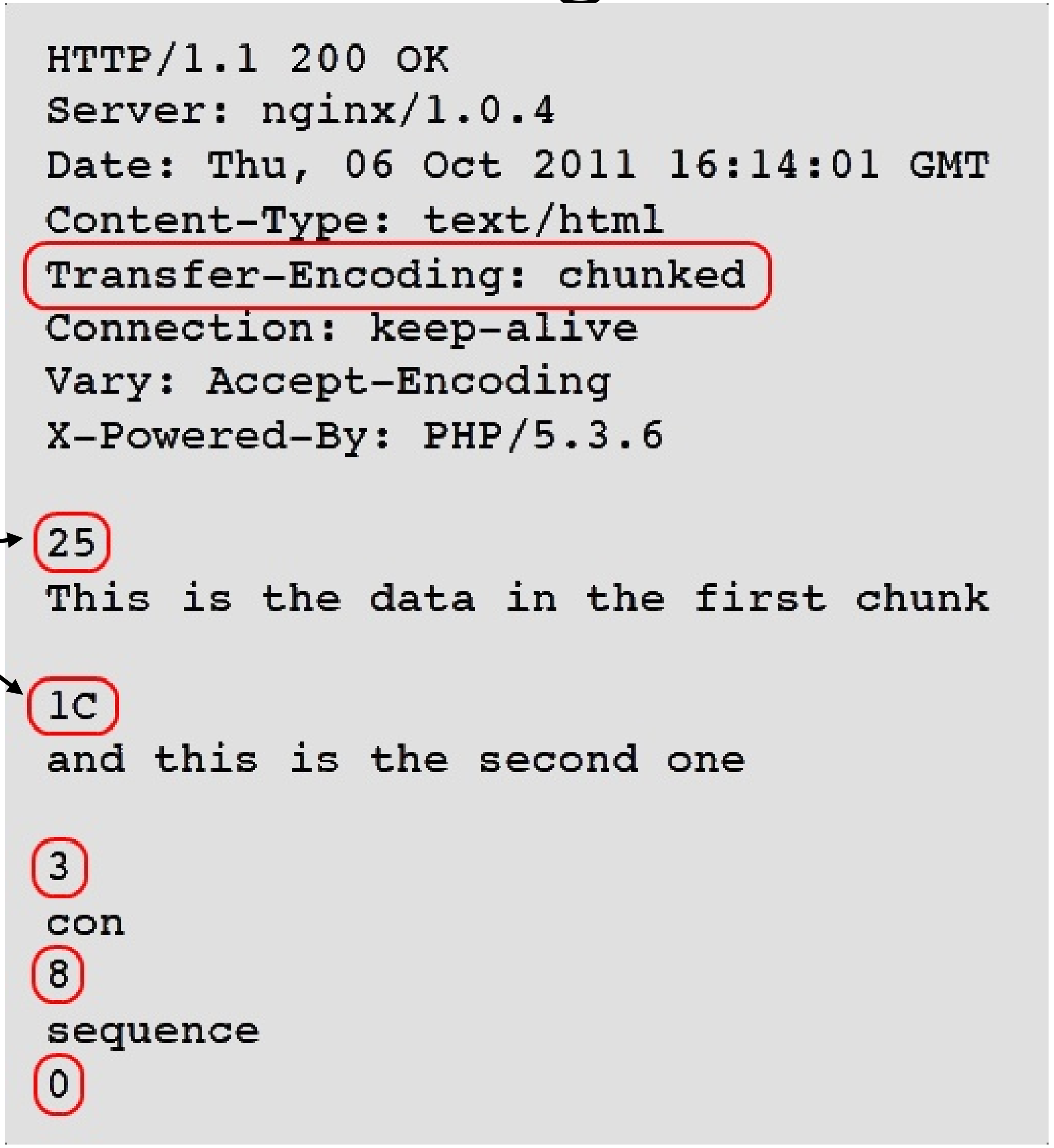
- Stream the response payload in chunks/blocks instead of returning the entire content
- Allows the server to dynamically generate content without knowing in advance how large the data is
  - Required to set the `Content-Size` header
- The connection is kept opened until all chunks have been transferred



Example Chunked Encoding

Set `Transfer-Encoding` to chunked

Chunk size in hex



```
HTTP/1.1 200 OK
Server: nginx/1.0.4
Date: Thu, 06 Oct 2011 16:14:01 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.3.6

25
This is the data in the first chunk

1C
and this is the second one

3
con
8
sequence
0
```

Example Chunk Transfer

```
app.get('/api/photos/:id', (req, resp) => {
  const imgName = req.params('id');
  fs.readFile(imgName, (err, data) => {
    let start = 0; end = data.length;
    if (req.header['range'])
      let range = parseRange(data.length, req.header['range']);
      start = range.start;
      end = range.end;
    resp.status(200).type('image/png');
    resp.write(data.slice(start, end));
    resp.end();
  })
})
```

If there is a Range HTTP header, use the NPM package range-parse to parse the range

Node will set the Transfer-Encoding header to chunked if the data is return with `write()`
Unlike `send()`, you can call `write()` multiple times

Example Chunk Transfer

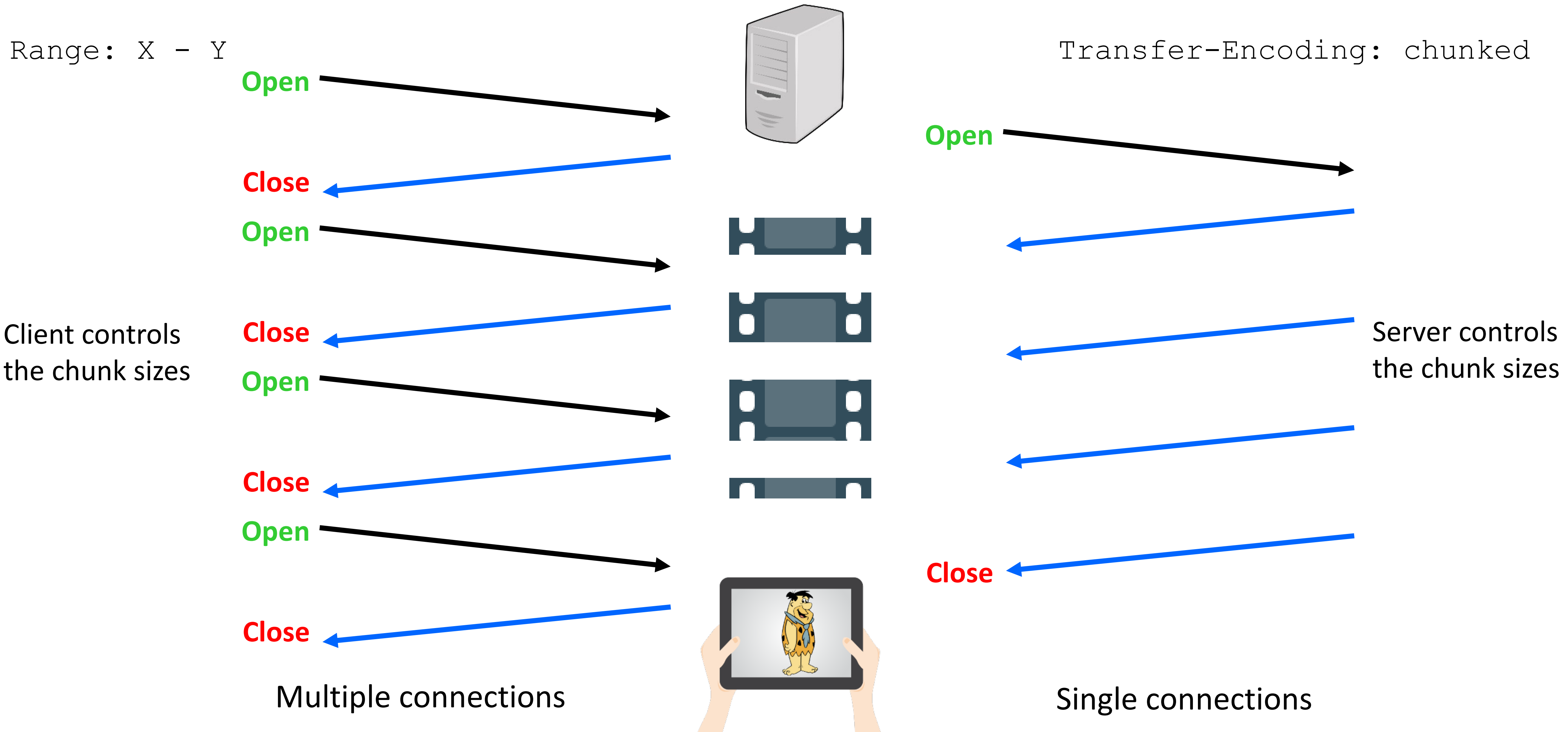
```
let fileSize = 0;

request.get('http://.../photos/2')
  .on('data', (data) => {
    fileSize += data.length;
    //Do something will data
  })
  .on('error', (error) => {
    //Restart transfer
  })
  .on('end', (data) => {
    console.log('end: all done ');
    fileSize = 0;
})
```

data event is fired on each `write()`

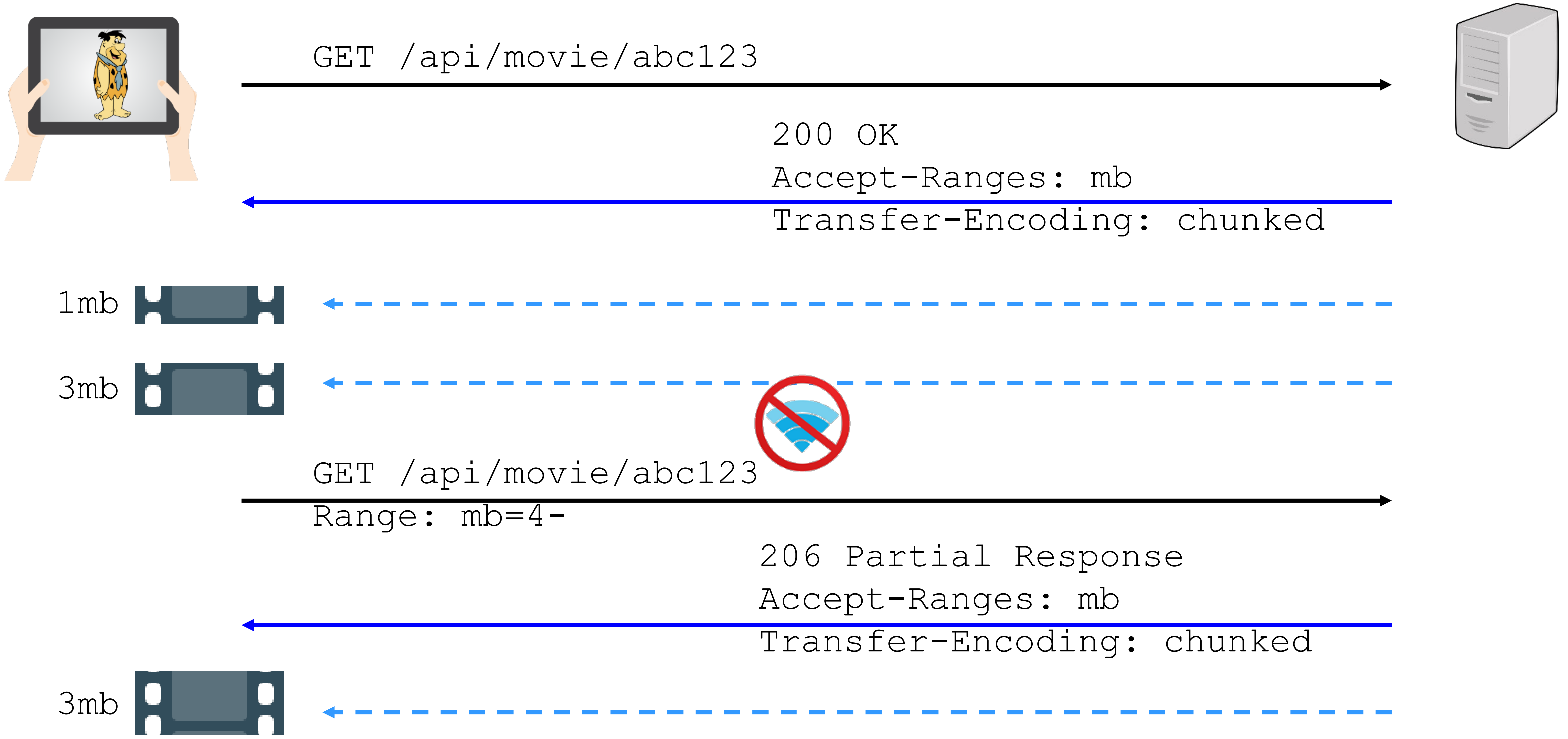
Save the size read. Use to specify the `Range` if there is an error in the transfer

Transfer complete

Range and Chunked Transfer



Combining Chunk Transfer and Partial Response



```
GET /api/movie/abc123
```

```
200 OK
Accept-Ranges: mb
Transfer-Encoding: chunked
```

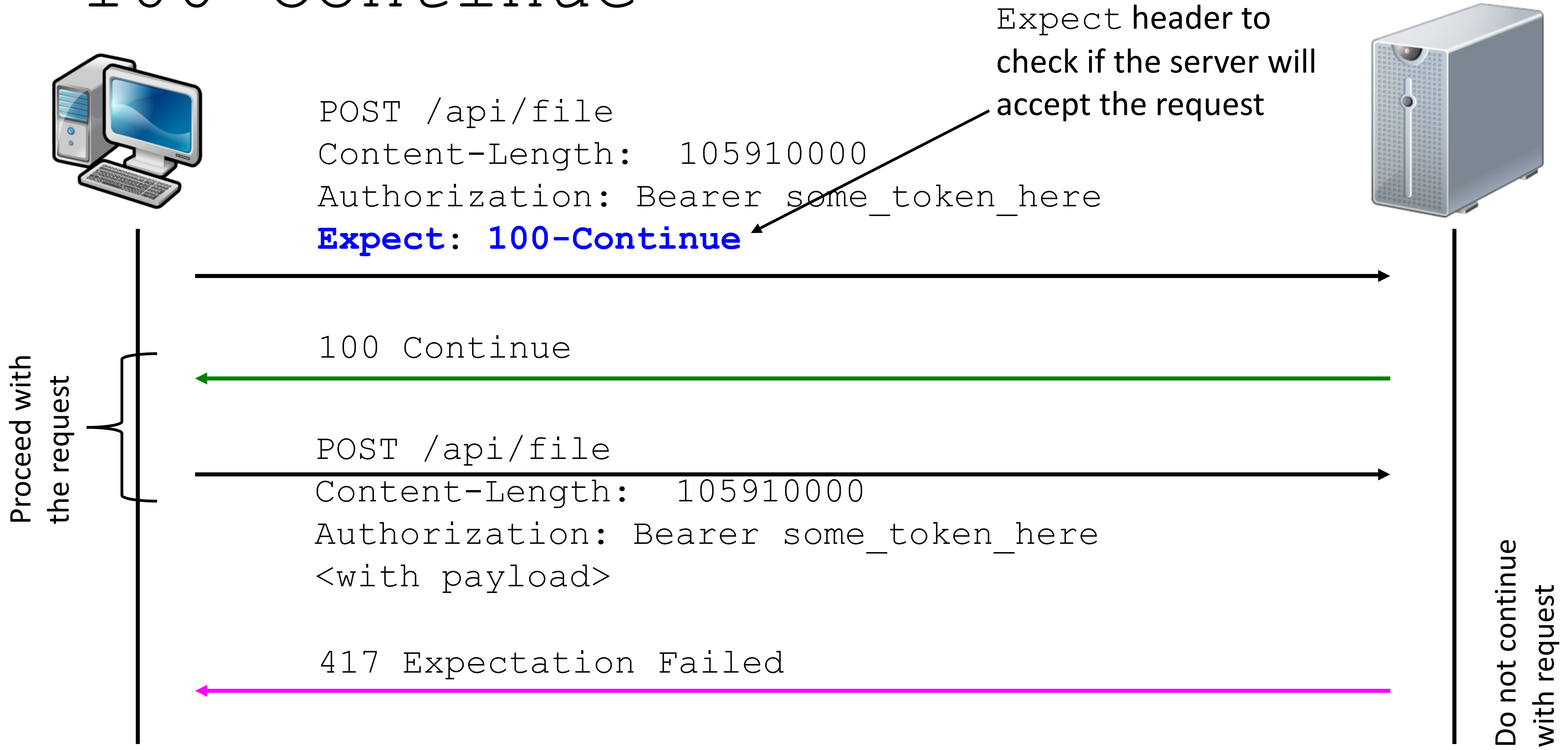
```
GET /api/movie/abc123
Range: mb=4-
```

```
206 Partial Response
Accept-Ranges: mb
Transfer-Encoding: chunked
```

`100-Continue`

- Used by the client to check if it should continue
  - Eg. before uploading a large file
- The client only sends the header information not the payload
  - Allows the server to examine the header information
  - Eg. check `Content-Length` to see if the client has exceeded certain limitations
  - Eg. check authorization before sending the actual data
- Server response with one of the following status code
  - `100 Continue` – which indicate to the client the it can proceed with the request. The actual request will be made
  - `417 Expectation Failed` – client should not proceed with the request

100-Continue



`Expect` header to check if the server will accept the request

```
POST /api/file
Content-Length:  105910000
Authorization: Bearer some_token_here
Expect: 100-Continue
```

```
100 Continue
```

```
POST /api/file
Content-Length:  105910000
Authorization: Bearer some_token_here
<with payload>
```

```
417 Expectation Failed
```

Handing **`100-Continue`** on Node

Express

```
const app = express();

app.post('/api/upload', ...}

const server = app.listen(PORT, () => { ... });

server.on('checkContinue', (req, resp) => {
  const contentSize = parseInt(req.headers['content-length']);

  if (contentSize > MAX_SIZE)
    return resp.status(417).end();

  resp.writeContinue();
  resp.emit('request', req, resp);
})
```

Start the web application. `listen()` will return the HTTP server

`checkContinue` will fire when there is an `Expect: 100-continue` in the request

If the request fails expectation return a 417 status

Sends a `100-Continue` status back to the client. Ask it to proceed with the actual request

Notify Node to handle this request

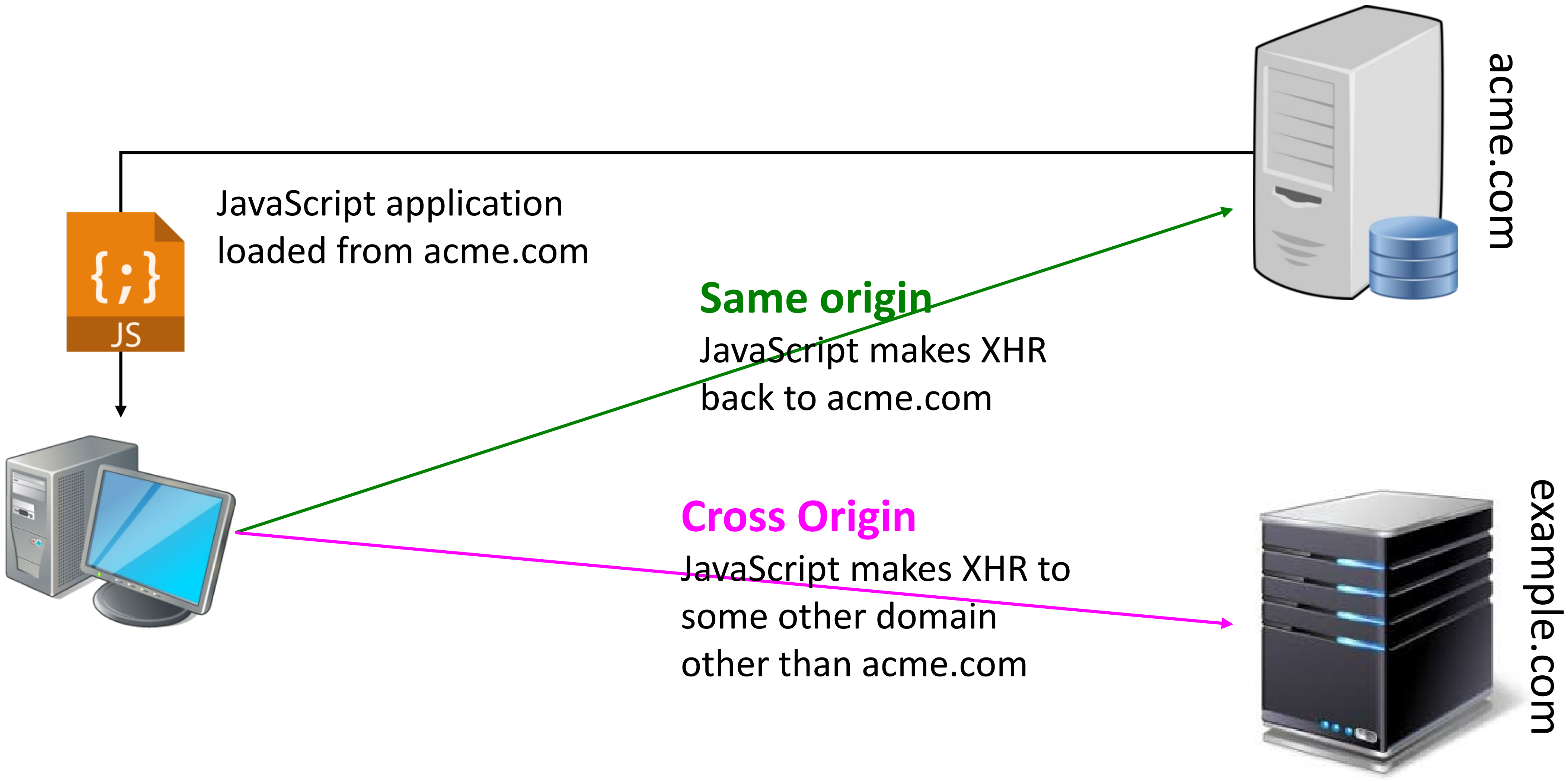
Requesting and Handle **`100-Continue`**

```
const request = require('request');

const formData = { ... };
const headers = {
   Expect: '100-continue'
};

request.post({
   url: '/api/upload', formData: formData, headers: headers
}).on('response', (resp) => {
   if (417 == resp.statusCode) {
      //Failed expectation
   }
   //POST request has been successfully processed
})
```

Set the `Expect` header in the request

When the response event is fired, check the status code

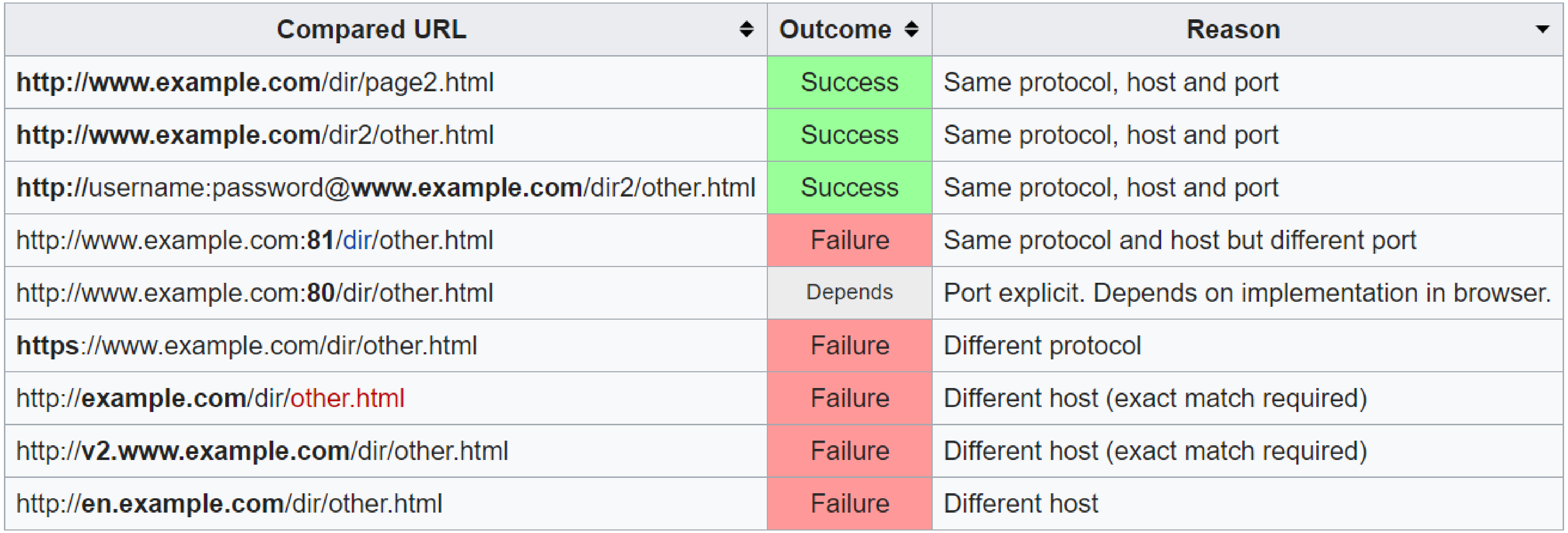
Same Origin and Cross Origin


JavaScript application loaded from acme.com
JS
Same origin
JavaScript makes XHR back to acme.com
Cross Origin
JavaScript makes XHR to some other domain other than acme.com
acme.com
example.com

Same Origin Policy

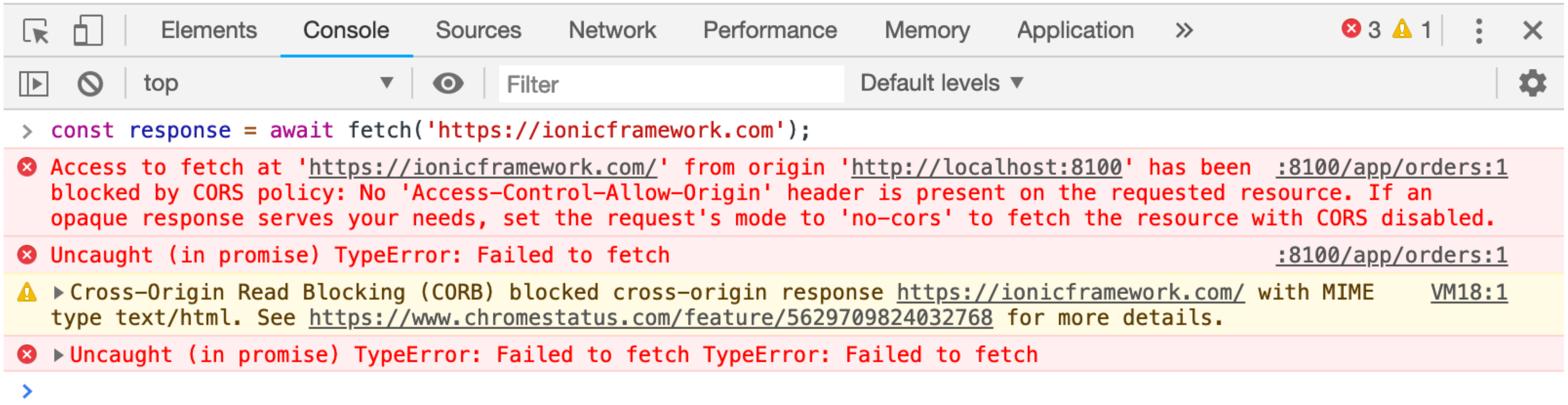
`http://www.example.com/dir/page1.html`

- Web security model for JavaScript
- Browser will not allow data requested by XHR from a different origin



Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol, host and port
http://www.example.com/dir2/other.html	Success	Same protocol, host and port
http://username:password@**www.example.com**/dir2/other.html	Success	Same protocol, host and port
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
http://www.example.com:**80**/dir/other.html	Depends	Port explicit. Depends on implementation in browser.
https://www.example.com/dir/other.html	Failure	Different protocol
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)
http://**en.example.com**/dir/other.html	Failure	Different host

CORS Error



```
> const response = await fetch('https://ionicframework.com');
✖ Access to fetch at 'https://ionicframework.com/' from origin 'http://localhost:8100' has been   :8100/app/orders:1
  blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an
  opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
✖ Uncaught (in promise) TypeError: Failed to fetch                                              :8100/app/orders:1
⚠ ▶Cross-Origin Read Blocking (CORB) blocked cross-origin response https://ionicframework.com/ with MIME   VM18:1
  type text/html. See https://www.chromestatus.com/feature/5629709824032768 for more details.
✖ ▶Uncaught (in promise) TypeError: Failed to fetch TypeError: Failed to fetch
>
```

Displayed in Developer Tools

JSONP

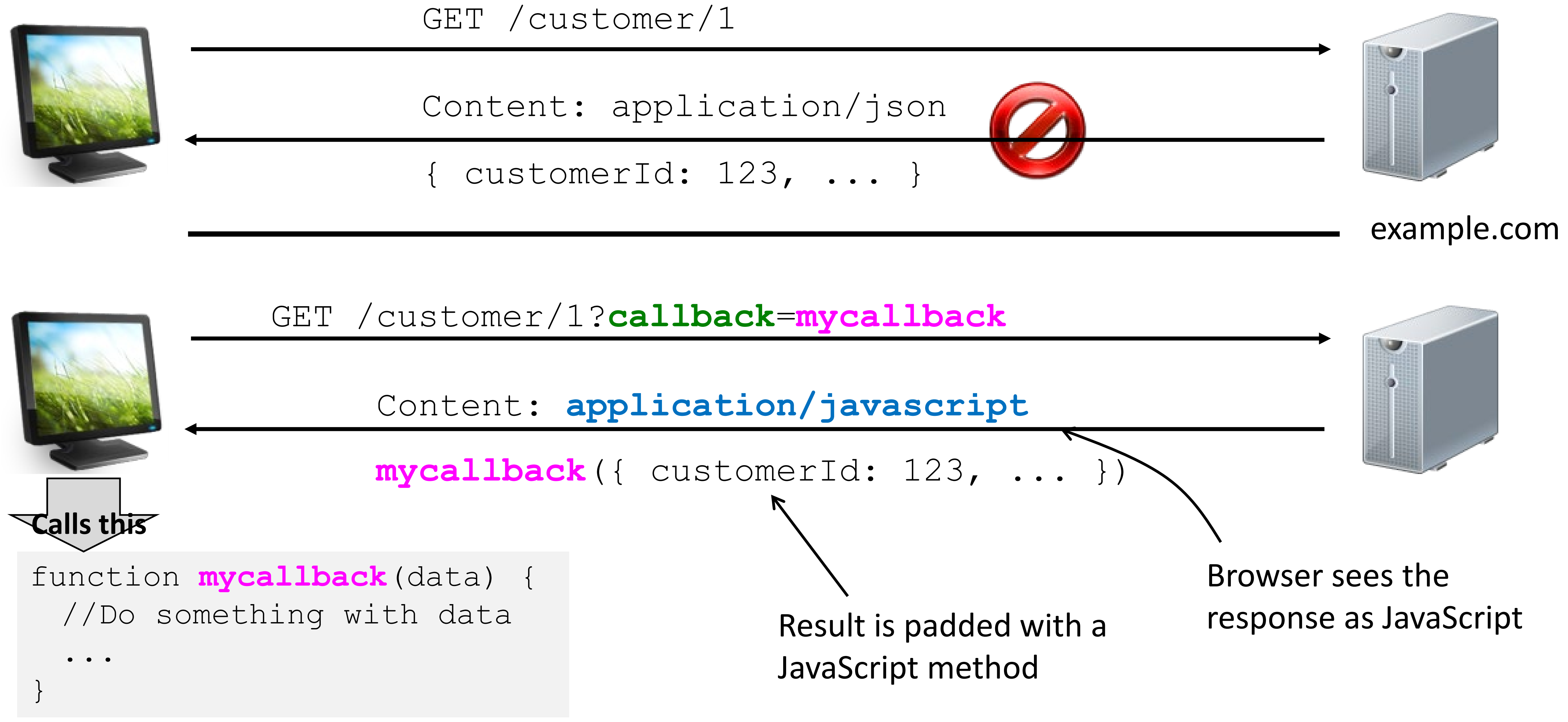
- Hides the JSON data inside a JavaScript
  - By wrapping the JSON data inside a method call
  - Exploit a browser ‘loophole’ to allow cross origin scripts to be loaded

```
mycallback({customerId: 1, ... });
```

  - On receiving the data, the browser will perform the call, passing you the data
- When performing a cross origin request to get JSON data, a callback function is passed as a parameter
  - `callback` is the standard parameter name used

```
http://example.com/customer/1?callback=mycallback
```

JSONP Illustrated


example.com
Calls this
Result is padded with a JavaScript method
Browser sees the response as JavaScript

```
GET /customer/1

Content: application/json
{ customerId: 123, ... }
```

```
GET /customer/1?callback=mycallback

Content: application/javascript
mycallback({ customerId: 123, ... })
```

```
function mycallback(data) {
  //Do something with data
  ...
}
```

Example JSONP

```
import { HttpClientModule, HttpClienJsonpModule }
     from '@angular/common/http';

constructor(private jsonp: Jsonp) {}

this.jsonp('http://example.com/customer/1', 'cb')
   .toPromise()
   .then(result => {
     //Do something
     ...
   })
```

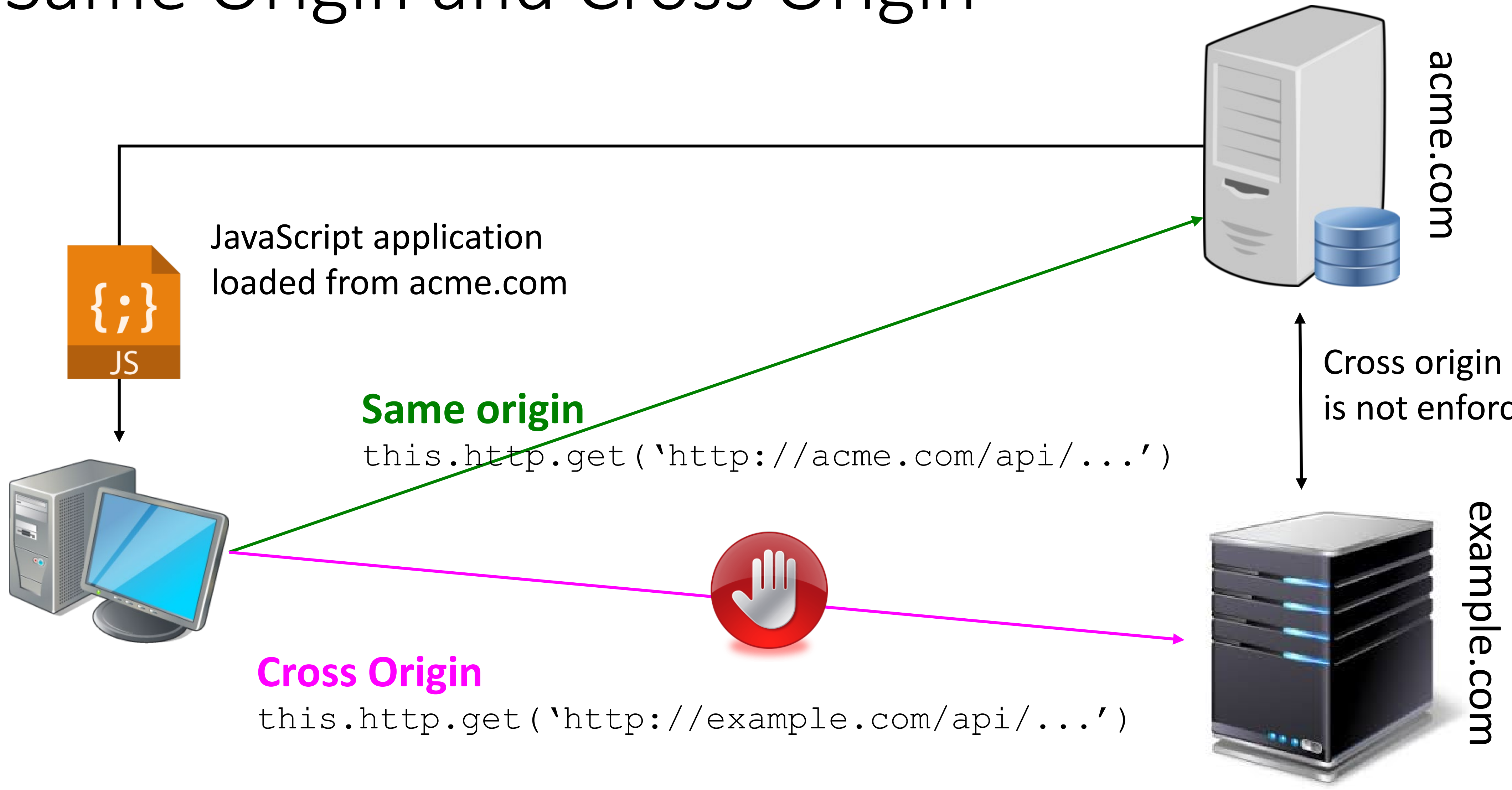
Sets the query parameter's name for the callback function

```
GET /customer/1?cb=abc123
```

```
app.set('jsonp callback name', 'cb');

app.get('/customer/:cid', (req, resp) => {
   const result = //Get customer data
   resp.status(200);
   resp.jsonp({ data: result });
})
```

```
200 OK
Content-Type: application/javascript
abc123({ data: result })
```

Same Origin and Cross Origin



```
this.http.get('http://acme.com/api/...')
```

```
this.http.get('http://example.com/api/...')
```

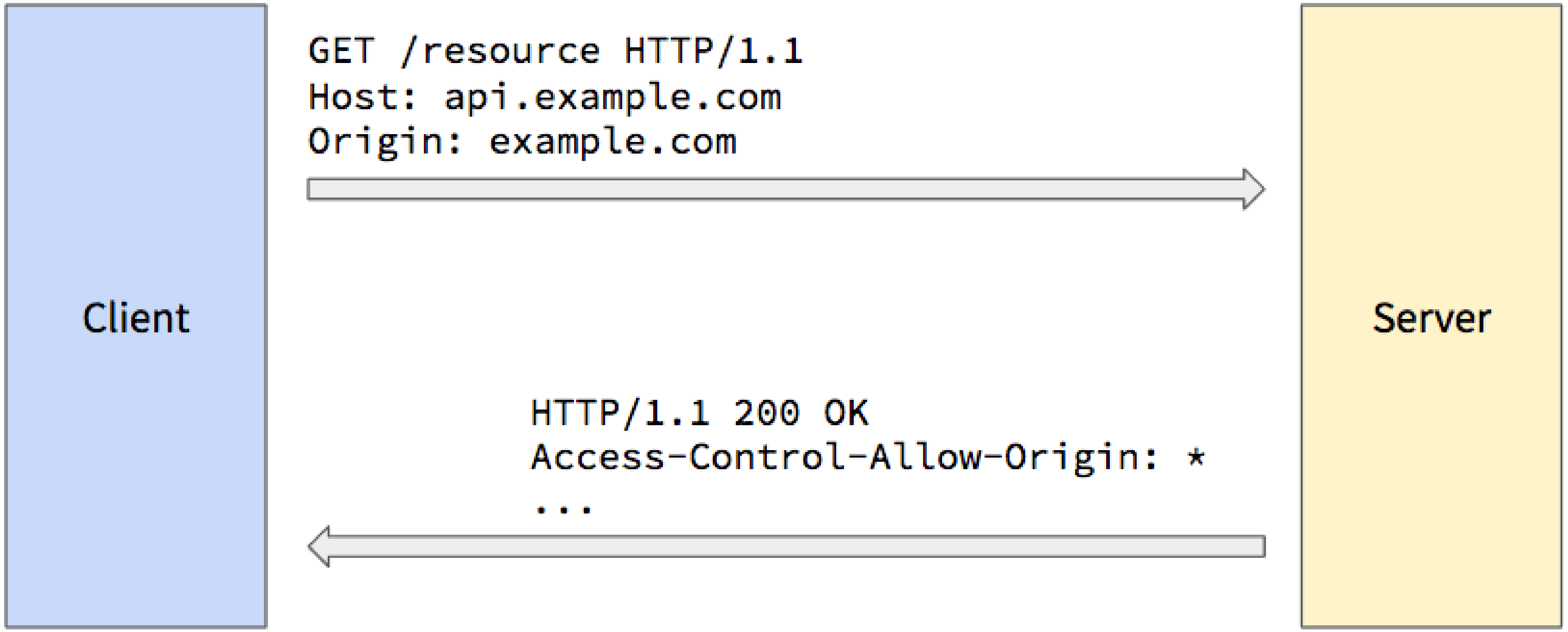
Cross Origin Resource

- JSONP is a hack
  - Masquerade the result as JavaScript
- Only works with GET
  - Result may be cached
- Cross origin resource allows clients to make cross origin request
  - Using JavaScript
- REST servers must opt-in
  - By adding extra headers in the response
- CORS is only enforced on the client by the browser
  - Not enforce if request is from server to server
  - Eg. using `request` to make REST invocation in Node

Types of CORS Request

- Simple request are REST request made with the following
  - Methods – `GET`, `POST`, `HEAD`
  - Request headers
    - `Accept`
    - `Content-Type` – `text/plain`, `application/x-www-form-urlencoded`, `multipart/form-data`
- Request with preflight
  - Any request that is not a simple request
    - Eg. `Content-Type: application/json`
  - Request 2 request
    - First request OPTIONS
    - The actual request

Example Simple CORS Request

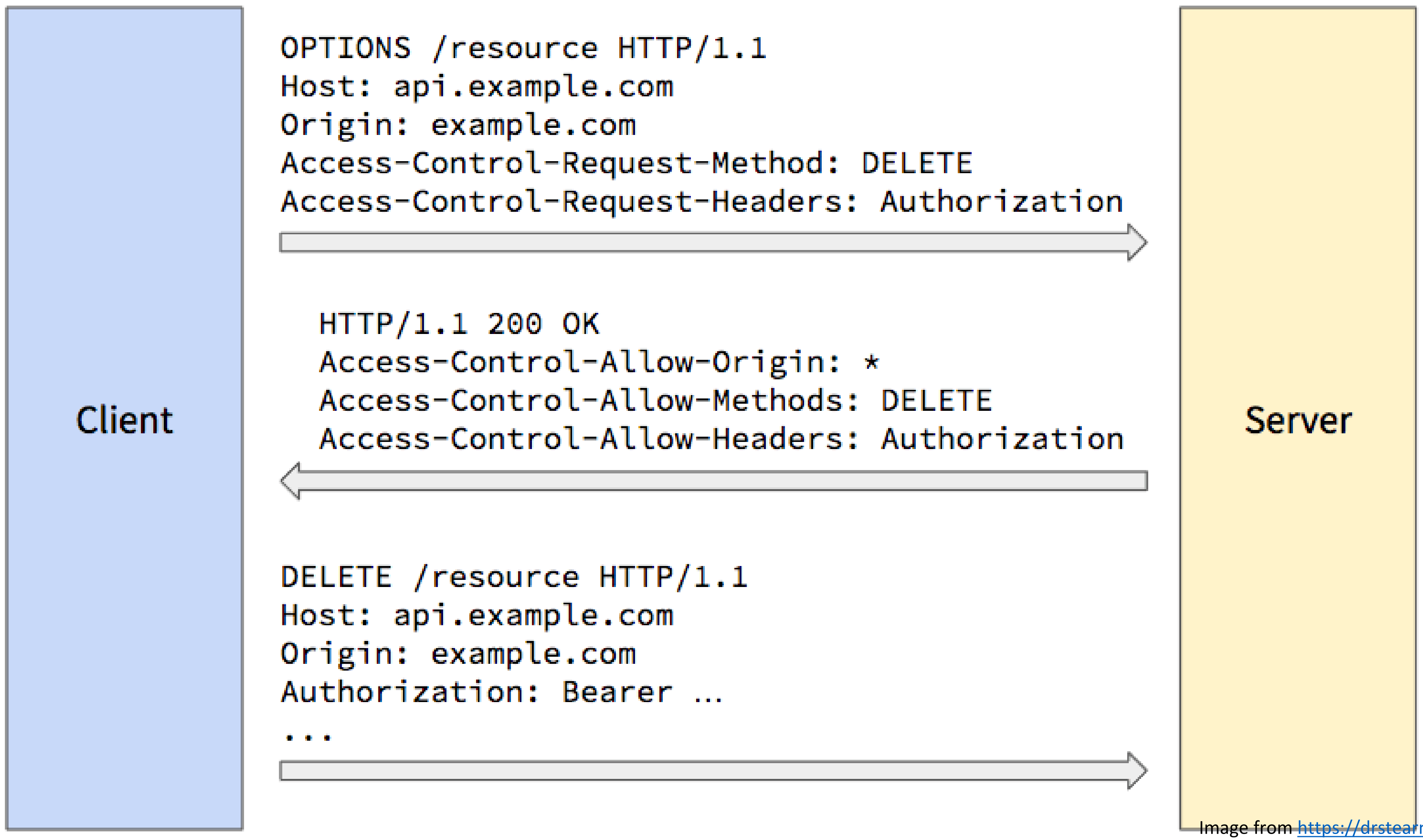


```
GET /resource HTTP/1.1
Host: api.example.com
Origin: example.com
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
...
```

Image from https://drstearns.github.io/tutorials/cors/

Example CORS Request with Preflight



```
OPTIONS /resource HTTP/1.1
Host: api.example.com
Origin: example.com
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: Authorization
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: DELETE
Access-Control-Allow-Headers: Authorization
```

```
DELETE /resource HTTP/1.1
Host: api.example.com
Origin: example.com
Authorization: Bearer ...
...
```

Image from https://drstearns.github.io/tutorials/cors/

Example CORS

```
this.http.get('http://example.com/api/customers')
   .toPromise()
   .then(result => {
     //do something with the result
   })
```

```
GET /api/customers
Origin: acme.com
Accept: application/json
```

```
OK 200
Content-Type: application/json
Access-Control-Allow-Origins: *
```

```
const cors = require('cors');

app.use(cors());
app.get('/api/customers', ...)
```

CORS Express middleware. All request after that route will have CORS headers in their response.

Versioning API

- API changes over time
  - Product deprecated
  - New attributes in the response
  - Redesign an old API
- Need to maintain backward compatibility as API evolve
- Have a deprecation strategy
  - Will you continue to support old APIs
  - Publish a timeline to remove APIs

Versioning APIs

- There are 3 versioning strategy
- Version via URL segment

```
/api/v2/products
```

- Version via content negotiation

```
Accept: application/json;version=1
```

- Version via custom header

```
X-API-Version: v3
```

Version via URL Segment

```
//Version 1 of the API
const v1 = express.Router();
v1.use(...)

//Version 2 of the API
const v2 = express.Router();
v2.use(...)

const app = express();
app.use('/api/v1', v1);
app.use('/api/v2', v2);
```

Each Router object represents a version of the API

Mount different router under different resource

Versioning via Content Negotiation

- Set the version as a parameter of the representation

```
GET /customer/1
Accept: application/json;version=2
```

Parameter

- Can be interpreted as
  - Use version 2 of `/customer/1 API`
    - Multiple API version each producing a single result
  - Return version 2 of the representation
    - Single API producing multiple versions of representation

Example Versioning via Content Negotiation

Express

```
const contentType = require('content-type');

app.get('/api/customers', (req, resp) => {

   const header = req.get('Accept');
   const obj = contentType.parse(header);
   const ver = 'version' in obj.parameters? obj.version || 'v1';

   switch (ver) {
      case 'v1':
      default:
         //Do v1
      case 'v2':
         ....
   }
});
```

Parse the `Accept` header for version information

Return the version if present in the `Accept` header, otherwise default to v1

```
Accept: application/json; version=v2
```

List of Modules

- body-parser - https://www.npmjs.com/package/body-parser
- multer - https://www.npmjs.com/package/multer
- compression - https://www.npmjs.com/package/compression
- cors - https://www.npmjs.com/package/cors
- content-type - https://www.npmjs.com/package/content-type
- express-range - https://github.com/purposeindustries/express-range
- http-range - https://github.com/clns/node-http-range
- pako - https://www.npmjs.com/package/pako