

Building the Web Api

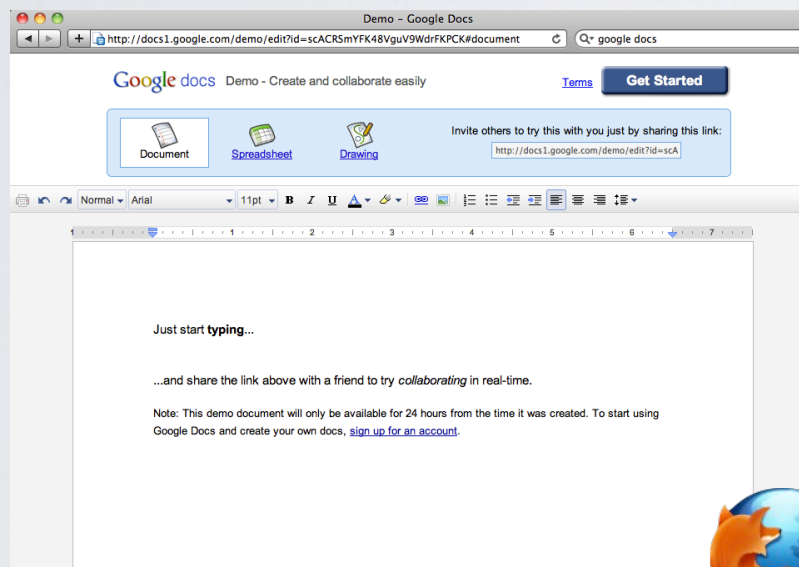
Thierry Sans

The HTTP protocol

Anatomy of a Web Application

Client Side

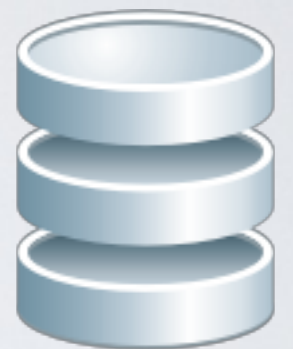
Server Side



Web Browser



Web Server



Database

The HTTP protocol

Network protocol for requesting/receiving data on the Web

- Standard TCP protocol on **port 80** (by default)
- **URI/URL** specifies what resource is being accessed
- The **request method** specified with a command

Let's look at what a web server does

telnet to a web server



```
> telnet www.utsc.utoronto.ca 80  
GET /
```



enter HTTP requests

Anatomy of a URL



HTTP Request Methods

- **POST** - add an unidentified resource
- **PUT** - add a an identified resource
- **GET** - get a resource
- **PATCH** - update a resource
- **DELETE** - delete a resource
- and others HEAD, TRACE, CONNECT, OPTIONS

HTTP Request

- **Method** - POST, PUT, GET, PATCH, DELETE ...
- **Query String**
- **Headers** - key/value pairs
- [optional] **Body** - data

Using the command **curl**

```
$ curl options url
```

```
-v verbose
```

```
--request request_method
```

```
--data request_body
```

```
--header header
```

HTTP response

- **Status code**
- **Headers** - key/value pairs
- [optional] **Body** - data

HTTP response status codes

- 1xx - information
- 2xx - success
- 3xx - redirection
- 4xx - client error
- 5xx - server errors

Method properties

An HTTP request/response

- may have a request body
 - may have a response body
 - may not have side effects a.k.a safe
 - may have the same result when called multiple times a.k.a idempotent
- ➡ the choice is left to the programmer

What the standard recommends

Method	Request Body	Response Body	Safe	Idempotent
POST				
PUT				
GET				
PATCH				
DELETE				

Building an HTTP server with Node.js

Node.js

- Runs on Chrome V8 Javascript engine
- Non blocking-IO (a.k.a asynchronous, a.k.a event-driven)
- No restrictions (unlike when js is running on the browser)

Example

src/node/readfile.js

```
import {readfile} from 'fs'

readFile('helloworld.txt', 'utf8', function(err, data) {
  if (err) console.log(err)
  return console.log("output 1")
});

console.log("output 2")
```

console

```
$ node example.js
output 2
output 1
```

Building an HTTP server with Node.js

src/node/httpserver.js

```
import { http } from 'http'
const PORT = 3000

function handler(req, res) {
  console.log("Method:", req.method)
  console.log("Url:", req.url)
  console.log("Headers:", req.headers)
  res.end('hello world!')
};

createServer(handler).listen(PORT, function (err) {
  if (err) console.log(err)
  else console.log("HTTP server on http://localhost:%s", PORT)
});
```

Routing HTTP requests

Process HTTP requests and execute different actions based on

- the request method
 - the url path
 - whether the user is authenticated
 - ect ...
- ⦿ A router can be written from scratch (but it is tedious)
 - ⦿ Use the backend framework **Express.js**

Express.js - HTTP Methods

src/express-examples/01_httpmethods.js

```
import { createServer } from 'http'
import express from 'express'
const app = express()
const PORT = 3000

// curl localhost:3000/
app.get('/', function (req, res, next) {
  res.end("Hello Get!")
});

// curl -X POST localhost:3000/
app.post('/', function (req, res, next) {
  res.end("Hello Post!")
})

createServer(app).listen(PORT, function (err) {
  if (err) console.log(err)
  else console.log("HTTP server on http://localhost:%s", PORT)
});
```

Express.js - Routing based on the path

src/express-examples/02_routing.js

```
// curl localhost:3000/  
app.get('/', function (req, res, next) {  
    res.end(req.path + ": the root")  
});  
  
// curl localhost:3000/messages/  
app.get('/messages/', function (req, res, next) {  
    res.end(req.path + ": get all messages")  
});  
  
// curl localhost:3000/messages/1234/  
app.get('/messages/:id/', function (req, res, next) {  
    res.end(req.path + ": get the message " + req.params.id)  
});
```

Express.js - body encoding

The body of HTTP request and response is a string

➔ **Problem:** how to send data structure between the frontend and backend?

➔ **Solution:** encode them either using:

- ✓ URI encoding (sometimes used)

see `src/express-examples/04_body-uri-encoded.js`

- ✓ XML encoding (rarely used these days)

- ✓ JSON encoding (very frequently used these days)

see `src/express-examples/05_body-json-encoded.js`

JSON

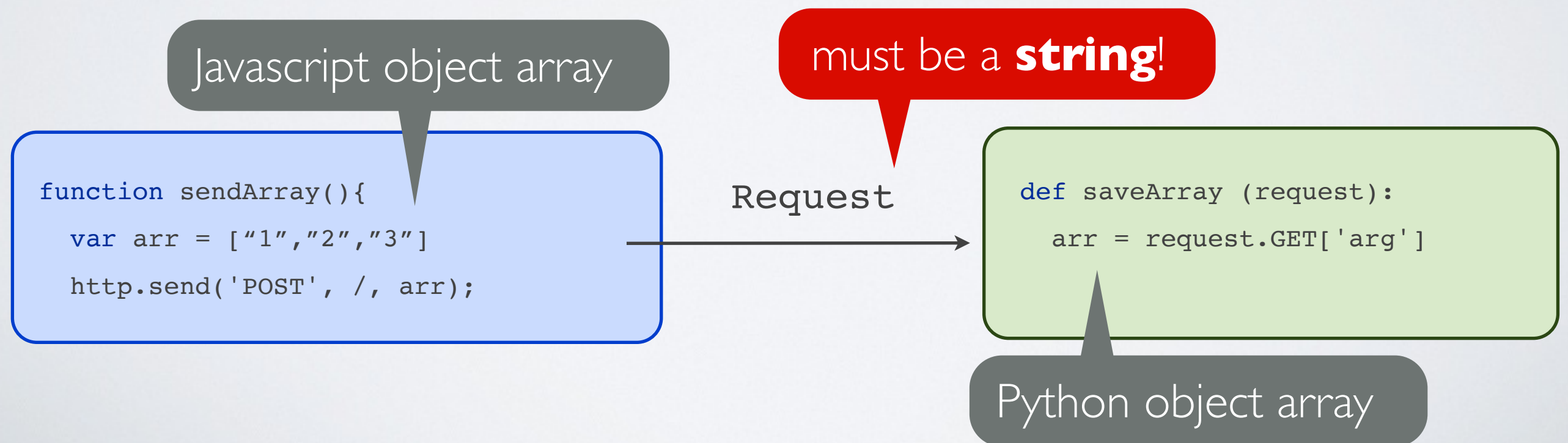
JavaScript Object Notation

Sending structured data

How to send a structured data (arrays or dictionaries) through an HTTP request or response?

➔ Only strings are send back and forth

✓ Have a string representation of a complex data structure



Why do we need JSON?

Original idea: using XML

✓ **In practice:** JSON is used for its simplicity

The JSON standard (RFC 4627)

- Lightweight open format to interchange data
- Human readable
- Used for serializing and transmitting structured data over a network connection (HTTP mostly)
- Since 2009 browsers support JSON natively

Anatomy of JSON

- A JSON data structure is either

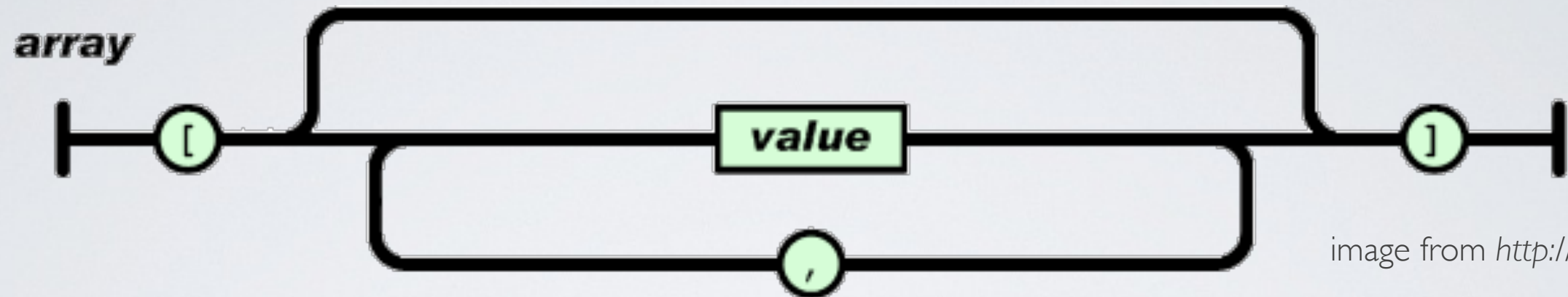
`array` (indexed array)

`object` (associative array)

- JSON values are

`string` – `number` – `true` – `false` – `null`

JSON Array

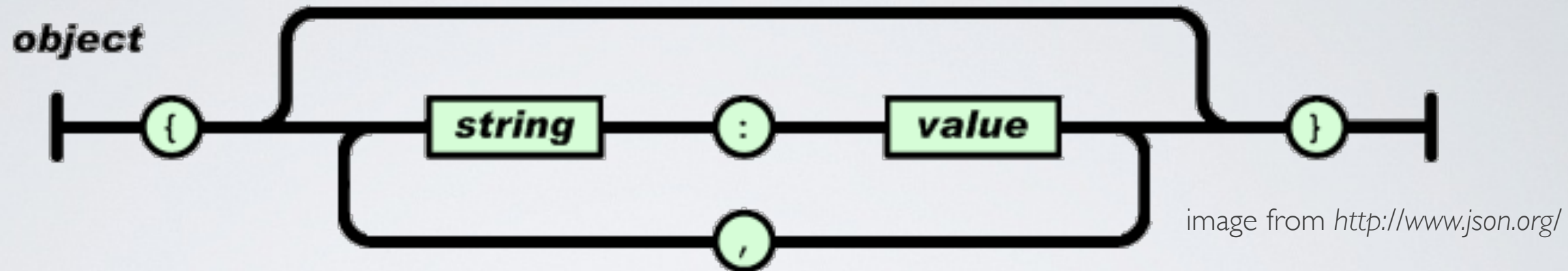


```
[  
  {"name": "Thierry"},  
  {"name": "Jeff"},  
  {"name": "Bill"},  
  {"name": "Mark"},  
]
```

or

```
[1, 2, 3, 4, 5]
```

JSON Object



```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "age": 25,  
  "male": true  
  "address":  
  {  
    "streetAddress": "21 2nd Street",  
    "additionalAddress": null  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021"  
  }  
}
```


JSON in Javascript (natively supported)

Serialization

Javascript → JSON

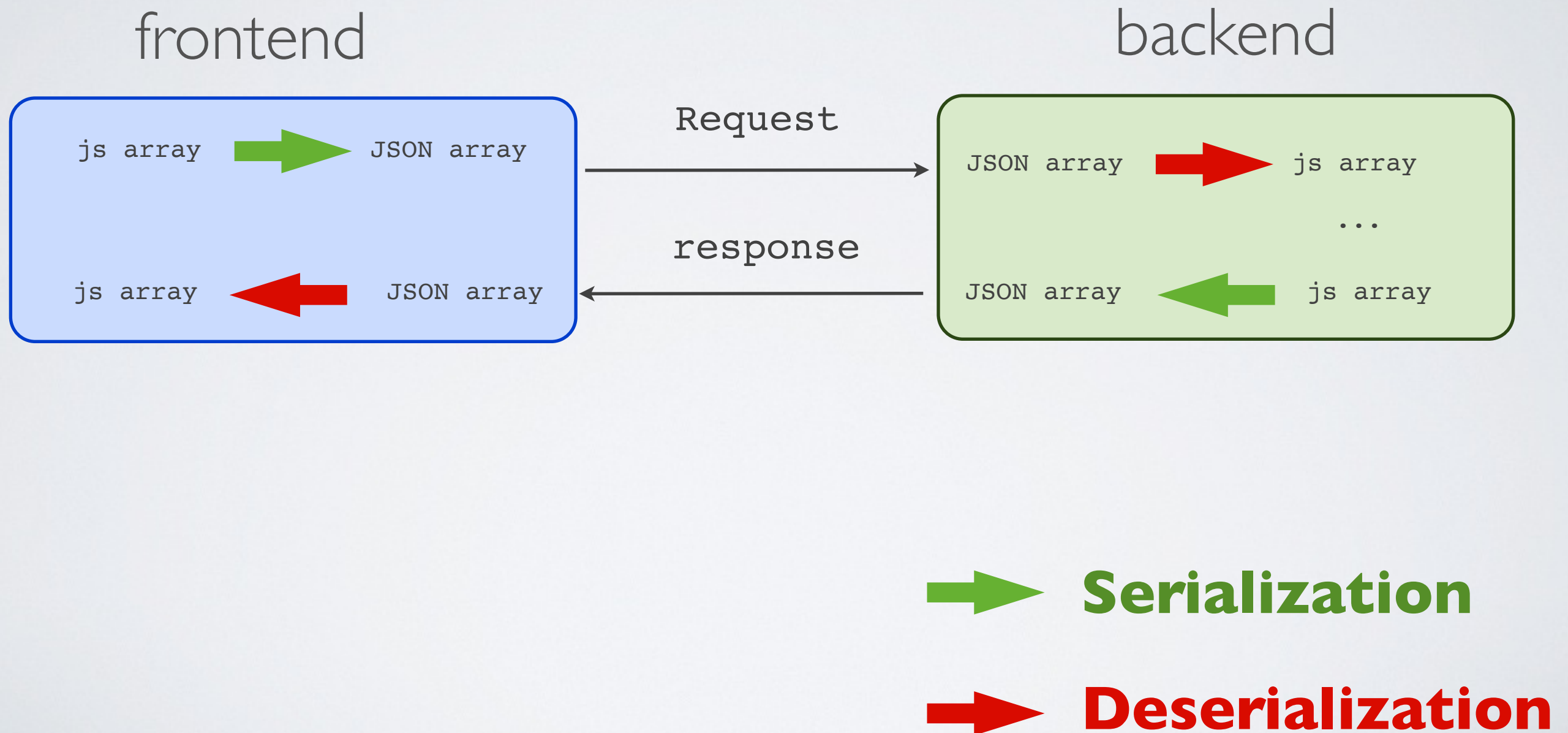
```
const myJSONText = JSON.stringify(myObject);
```

Deserialization

Javascript ← JSON

```
const myObject = JSON.parse(myJSONtext)
```

Serialization - Deserialization



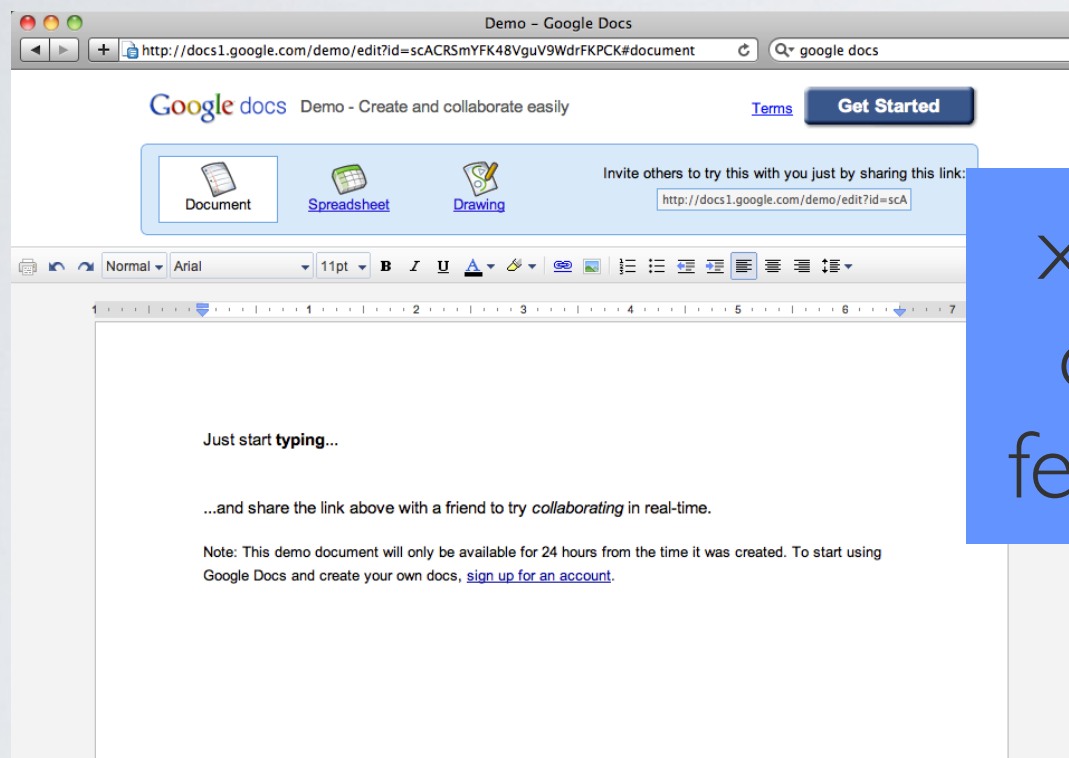
Asynchronous HTTP Requests

Why do we need to fetching resources asynchronously ?

In the past, when we wanted to

- send data to the server
- or retrieve data from the server
- we had to refresh the entire page
(i.e reloading HTML, CSS, JS and all media files)
- ✓ But, why not using Javascript to process the data and perform the necessary page changes?

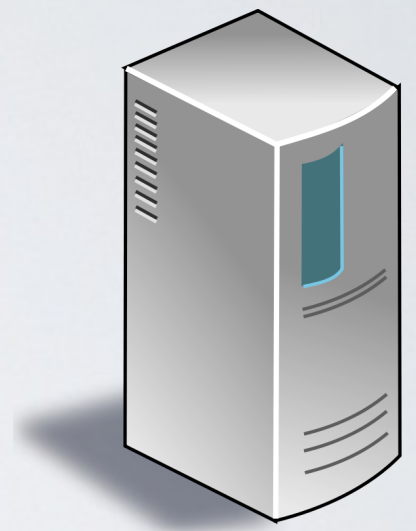
HTTP requests without refreshing the page



xhr
or
fetch

id=scACRSm...

anything



Javascript

Making asynchronous HTTP requests

`XMLHttpRequest`

(a.k.a AJAX - Asynchronous Javascript and XML)

- Invented by *Microsoft* (2000) and adopted by all other browsers
- Asynchronism is managed with a *callback*

`Fetch`

- Came with *Ecmascript 6* (2015)
- Asynchronism is managed with a *promise*

Ajax has revolutionized the Web

✓ Started with Gmail and Google Maps

- Advantages
 - Low latency
 - Rich interactions
- Consequences
 - Webapp center of gravity moved to the client side
 - Javascript engine performance race

XMLHttpRequest (legacy)

```
const xhr = new XMLHttpRequest();
xhr.onload = function() {
  if (xhr.status !== 200)
    console.error("[ " + xhr.status + " ]" + xhr.responseText);
  else
    console.log(xhr.responseText);
};
xhr.setRequestHeader(key, value);
xhr.open(method, url, true);
xhr.send(body);
```

(always) asynchronous

Fetch (new)

```
fetch(url, {  
  method: method  
  body: body  
}).then(function(res) {  
  if (res.status !== 200) throw new Error(res.status);  
  return res.text();  
}).then(function(text) {  
  console.log(text);  
}).catch(function(error) {  
  console.log(error);  
})
```

Concurrency issue - a typical example

```
var result = ""
```

initialization

```
fetch(url)
```

```
.then(function(response) {  
    return response.text();  
}).then(function(res) {  
    result = res;  
})
```

asynchronous

```
result = res;
```

assignment

```
document.getElementById.innerHTML = result;
```

access

result will be "" because of asynchronism

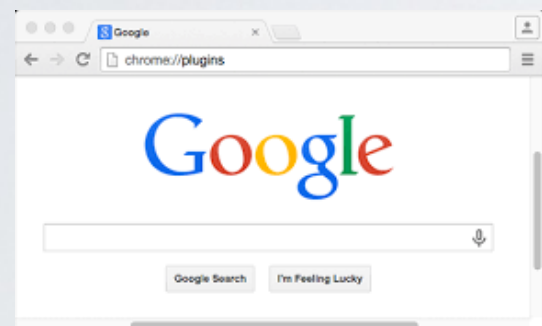
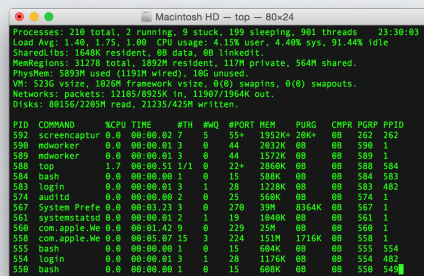
➔ **Race Condition!**

(REST) Web API

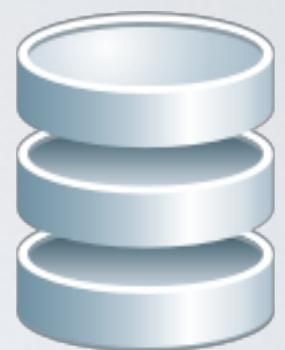
Modern Web Platform

Client Side

Server Side



Web API



Database

➡ The server side is more or less of a **storage system**

REST - Representational State Transfer

Design a remote API for a storage system by using HTTP

- **Function names** : method and URL
- **Function arguments** : URL and request body
- **Returned value** : status code and response body

REST concepts

Mostly storage systems are meant to store

- **Collections** (or resources)
- **Elements** that belongs to one or several collections

Examples

	HTTP request	HTTP response
Create a new message	<code>POST /messages/ "Hello World"</code>	<code>200 "78"</code>
Get all messages	<code>GET /messages/</code>	<code>200 "['Hello world', ...]"</code>
Get a specific messages	<code>GET /messages/78/</code>	<code>200 "Hello World"</code>
Delete a specific messages	<code>DELETE /messages/78/</code>	<code>200 "success"</code>

Relationships


Type	Example
one-to-one	<code>/users/sansthie/profile/firstname/</code>
one-to-many	<code>/users/sansthie/messages/89/</code>
many-to-many	<code>/users/sansthie/teams/8/ /teams/8/users/sansthie/</code>

CRUD - manipulating data

Basic functions of persistent storage

- **C**reate
- **R**ead
- **U**ppdate
- **D**eleate

Query methods

CRUD	HTTP	Collection	Element
Create	POST		Create a new element
	PUT	Replace the entire collection	Create (or replace if exists) a specific element
Read	GET	List all elements	Retrieve a specific element
Update	PATCH	Update some attributes of some elements	Update some attributes of a specific element
Delete	DELETE	Delete the entire collection	Delete a specific element

Status codes

<http://www.restapitutorial.com/lessons/httpmethods.html>

Use of attributes

Query a subset of a collection : filter, page, range ...

```
GET /messages/?from=67&to=99
```


Alternative to REST for data exchange

- SOAP (legacy) with XML
- **(consistency) GraphQL with JSON**
- **(performances) gRPC with Protocol Buffers**