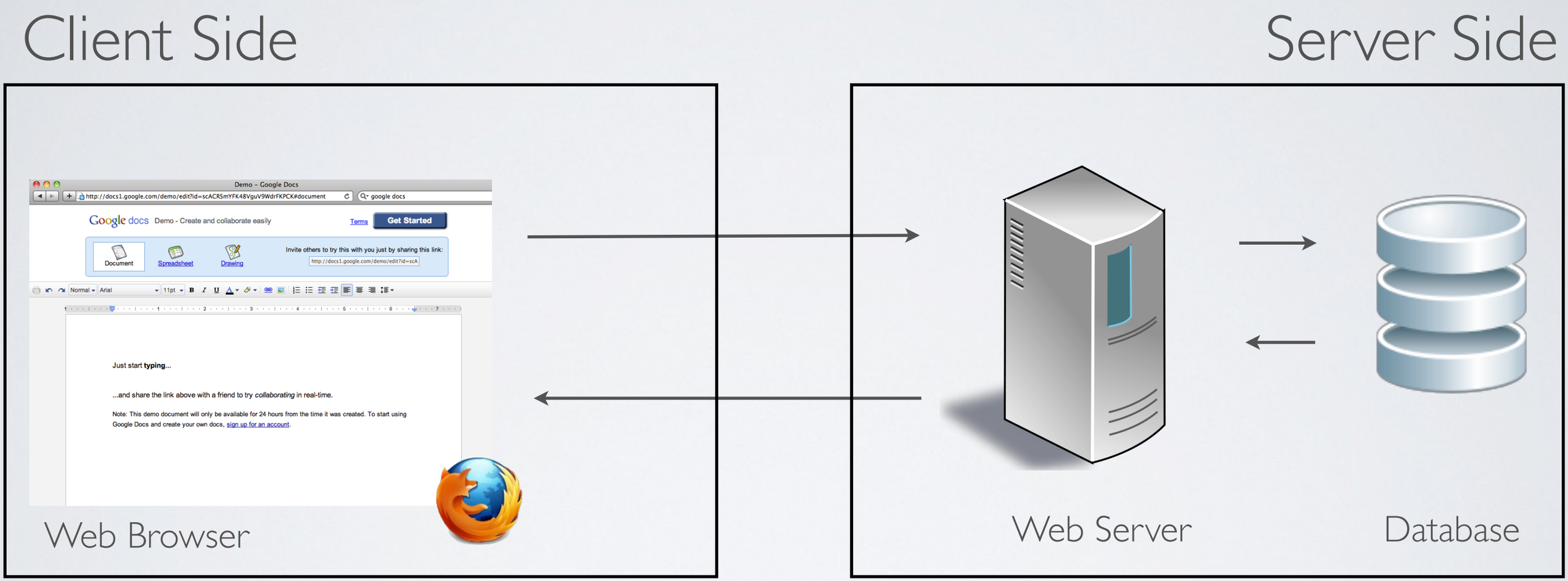
Building the Web Api

Thierry Sans

The HTTP protocol

Anatomy of a Web Application



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

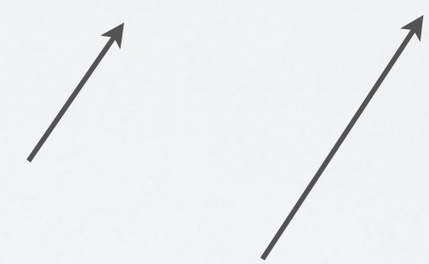
```
> telnet www.utsc.utoronto.ca 80
GET /
```

enter HTTP requests

Anatomy of a URL

protocol server query string

`http://www.utsc.utoronto.ca/~registrar/index.php?course2=CSCC09H3`



path

resource

parameters

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 <u>request body</u>
- may have a <u>response body</u>
- may not have side effects a.k.a <u>safe</u>
- may have the same result when called multiple times a.k.a <u>idempotent</u>

➡ 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.ka asynchronous, a. k.a event-driven)
- No restrictions (unlike when js is running on the browser)

Example

src/node/readfile.js

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

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

```
const http = require('http');
const PORT = 3000;

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

http.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

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

// 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!");
});

const http = require('http');
const PORT = 3000;

http.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

Thierry Sans

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

Javascript object array

```
function sendArray(){
  var arr = ["1","2","3"]
  http.send('POST', /, arr);
```

must be a **string**!

Request

```
def saveArray (request):
  arr = request.GET['arg']
```

Python object array

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

source *http://en.wikipedia.org/wiki/JSON*

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

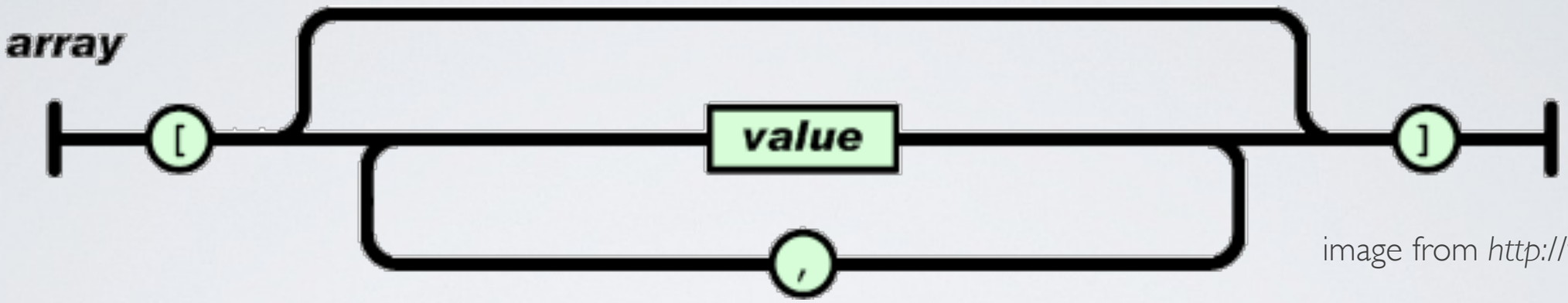


image from *http://www.json.org/*

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

or

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

JSON Object

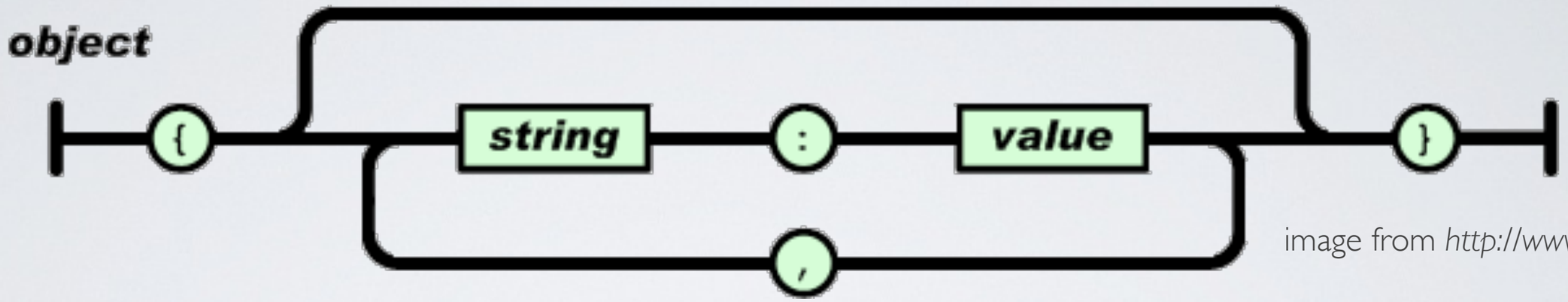


image from *http://www.json.org/*

```
{
    "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)

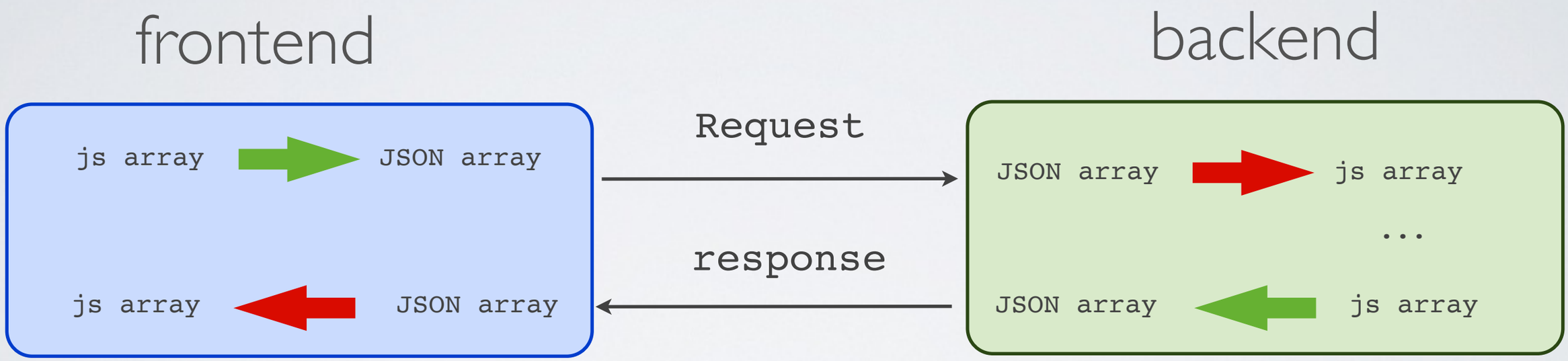
Serialization Javascript → JSON

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

Deserialization Javascript ← JSON

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

Serialization - Deserialization



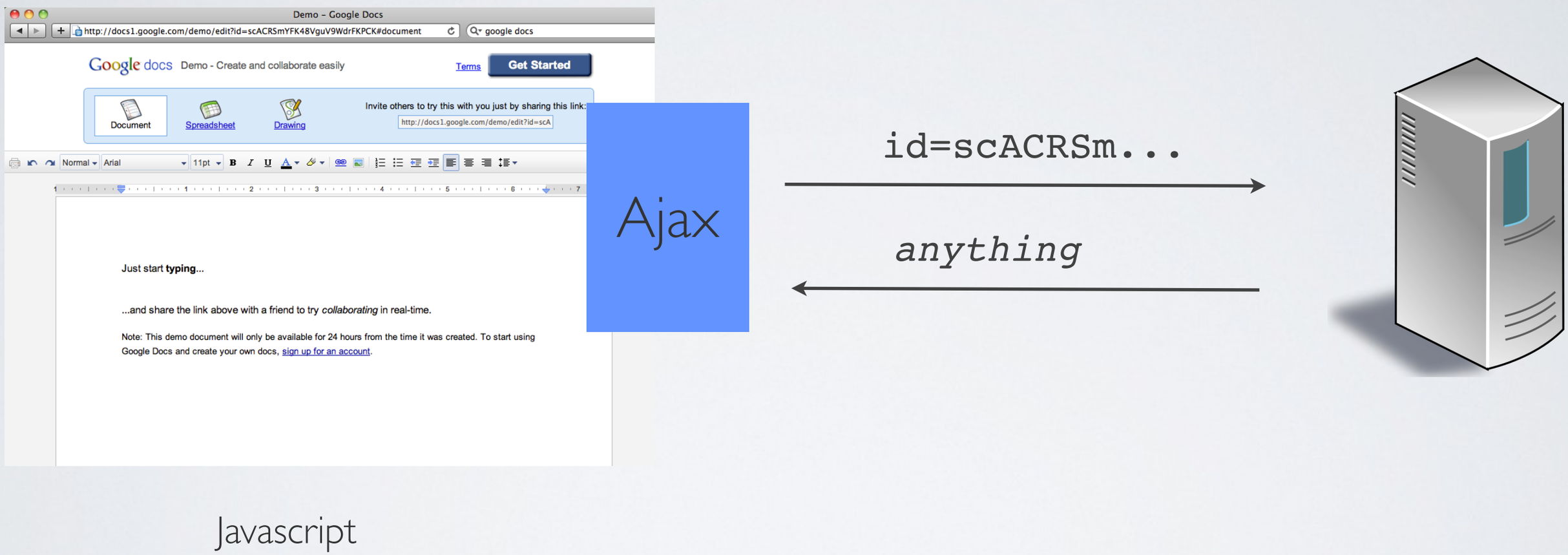
Serialization

Deserialization

AJAX

Asynchronous Javascript ~~and XML~~ and JSON

Ajax - fetching data without refreshing the page



Why do we need Ajax?

So far, 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?

Ajax - Asynchronous Javascript And XML

Fetch/push content from/to the server asynchronously
i.e without having to refresh the page

- Ajax is not a language

✓ It is a simple **Javascript command**

History of Ajax

- Patent from Microsoft (filled in 2000, granted in 2006)
  - XMLHTTP ActiveX control (Internet Explorer 5)
- Adopted and adapted by Opera, Mozilla and Apple
  - XMLHttpRequest Javascript object (standard)
- Before / After IE7
  - ◉ Different code for different browser (emergence of the javascript framework *Prototype*)
  - ✓ Javascript Object was adopted by IE7

Ajax 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

Standard Ajax

```
var 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

Concurrency issue in Ajax - a typical example

```
var result = ""
var xhr = new XMLHttpRequest();
xhr.onload = function (){
        result = xhr.responseText;
}
xhr.open(method, url, true );
xhr.send(body);
document.getElementById.innerHTML = result;
```

initialization

asynchronous

assignment

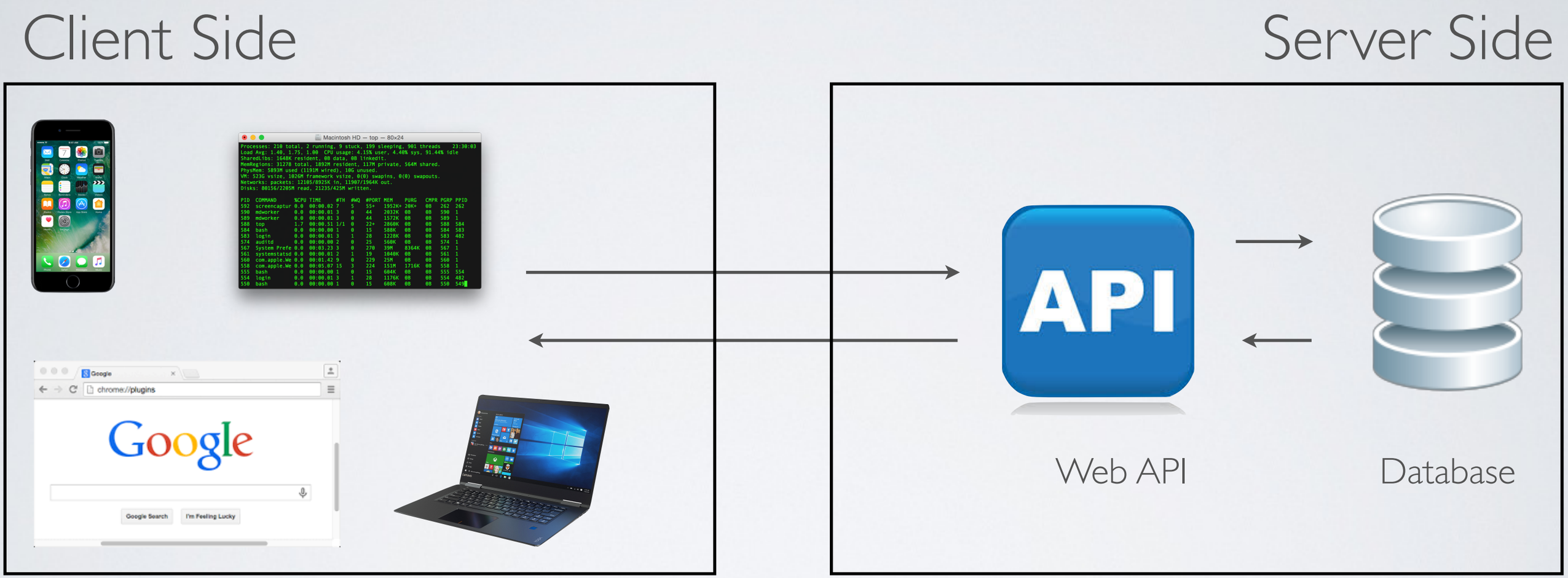
access

`result` will either be `""` or `"Hello world"` depending on the program and the execution context

➡ **Race Condition!**

(REST) Web API

Modern Web Platform



➡ 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 ressources)
- **Elements** that belongs to one or several collections

Examples

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

Relationships

Type	Example
one-to-one	`/users/sansthie/profile/firstname/`
one-to-many	`/users/sansthie/messages/89/`
many-to-many	`/users/sansthie/teams/8/` `/teams/8/users/sansthie/`

CRUD - manipulating data

Basic functions of persistent storage

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

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)
- **GraphQL (consistency)**
- **gRPC (performances)**