JAVASCRIPT

GOF – Design Patterns
HTTP protocol
NodeJs – Introduction

Bruno Oliveira: bmo@estg.ipp.pt
Marco Gomes: mfg@estg.ipp.pt
Miguel Andrade: mja@estg.ipp.pt

Design pattern

- A design pattern is a general, reusable solution to a commonly occurring problem;
- Each pattern has a name and becomes part of a vocabulary when discussing complex design solutions;
- The 23 Gang of Four (GoF) (1) patterns are generally considered the foundation for all other patterns.
- They are categorized in three groups: Creational, Structural, and Behavioral.

(1) - Book: Design Patterns: Elements of Reusable Object-Oriented Software

Creational Patterns

- Creational patterns provide ways to instantiate single objects or groups of related objects:
  - Abstract Factory: The abstract factory pattern is used to provide a client with a set of related or dependent objects;
  - Builder: The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm

Creational Patterns

- Prototype: The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone
- Singleton: The singleton pattern ensures that only one object of a particular class is ever created.
- (…)

Structural Patterns

- Structural patterns provide a manner to define relationships between classes or objects.
  - Adapter: The adapter pattern is used to provide a link between two otherwise incompatible types;
  - Facade: The facade pattern is used to define a simplified interface to a more complex subsystem;
  - (…)

Behavioural Patterns

- Behavioural patterns define manners of communication between classes and objects.
  - Chain of Responsibility: The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler;
  - Iterator: The iterator pattern is used to provide a standard interface for traversing a collection of items without the need to understand its underlying structure;
  - Observer: The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.
  - (…)

Example

- Singleton:

```
var Singleton = (function() {
  var instance;

  function createInstance() {
    var object = new Object("I am the instance");
    return object;
  }

  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

function run() {

  var instance1 = Singleton.getInstance();
  var instance2 = Singleton.getInstance();

  alert("Same instance? " + (instance1 === instance2));
}

run();
```

https://jsfiddle.net/brunobmo/vgo1owvj/

Source: https://joshbedo.github.io/JS-Design-Patterns/

Example

- Prototype Design Pattern:

```
var TeslaModelS = function() {
  this.numWheels = 4;
  this.manufacturer = 'Tesla';
  this.make = 'Model S';
}

TeslaModelS.prototype = function() {

  var go = function() {
    return "GO";
  };

  var stop = function() {
    return "brake";
  };

  return {
    pressBrakePedal: stop,
    pressGasPedal: go
  }

}();
alert(TeslaModelS.prototype.pressGasPedal());
```

https://jsfiddle.net/brunobmo/0vcjf26c/

Source: https://joshbedo.github.io/JS-Design-Patterns/

Example

- Constructor Pattern

```
class Car {
  constructor(opts) {
    this.model = opts.model;
    this.year = opts.year;
    this.miles = opts.miles;
  }

  // Prototype method
  toString() {
    return `${this.model} has driven ${this.miles} miles`;
  }
}

// Usage:
var civic = new Car({
  model: 'Honda',
  year: 2001,
  miles: 50000
});

alert(civic.toString());
```

https://jsfiddle.net/brunobmo/oekLso55/

Source: https://joshbedo.github.io/JS-Design-Patterns/

Further (web) Reading and sources

- https://joshbedo.github.io/JS-Design-Patterns/
- https://github.com/fbeline/Design-Patterns-JS/blob/master/docs.md
- http://loredanacirstea.github.io/es6-design-patterns/#creational-patterns
- https://www.tutorialspoint.com/design_pattern/index.htm

HTTP protocol

- HTTP is the protocol behind the World Wide Web, allowing a web server to send data to a client;
- The web browser understands the address syntax and knows it needs to make an HTTP request to a server;
- The address (e.g. http://google.com) is a URL, representing a specific resource on the web;
- Resources are “things” we interact: Images, pages, files, and videos;

HTTP protocol

- The URL has three parts:
  - http, the part before the ://, is the URL scheme;
    - The scheme describes how to access a particular resource;
  - google.com is the host.
    - This host name tells the browser the name of the computer hosting the resource.
  - `/search?safe=active&dcr=0&source=hp` (…) is the URL path;
    - The host should recognize the specific resource being requested by this path and respond appropriately;

HTTP protocol

- We also have the port number:
  - https://desporto.sapo.pt:80/futebol/primeira-liga/
    - The number 80 represents the port number the host is using to listen for HTTP requests;
  - We also have the query:
    - http://www.bing.com/search?q=broccoli
    - The query, also called the query string, contains information for the destination website to use or interpret;

HTTP protocol

- We can have Name-value pairs in the example:
  - http://www.bing.com/search?q=broccoli
- Fragments:
  - http://server.com?recipe=broccoli#ingredients
    - The fragment is only used on the client and it identifies a particular section of a resource.
- Unsafe characters
  - You can transmit unsafe characters in a URL, but all unsafe characters must be percent-encoded (URL encoded);
  - %20 is the encoding for a space character (where 20 is the hexadecimal value for the US-ASCII space character);
  - These characters include the blank/empty space and " < > # % { } | \ ^ ~ [ ] `
    - Don’t use them!

HTTP: Resources and Media Types

- There are thousands of different resource types on the web;
- When a host responds to an HTTP request, it returns a resource and also specifies the content type (also known as the media type);
- To specify content types, HTTP relies on the Multipurpose Internet Mail Extensions (MIME) standards;
- For example, when a client requests an HTML webpage, the host can respond to the HTTP request with some HTML that it labels as "text/html".
- The "text" part is the primary media type, and the "html" is the media subtype.

HTTP: Messages

- HTTP is a request and response protocol.
- A client sends an HTTP request to a server using a formatted message and the server responds by sending an HTTP response;
- The request and the response are two different message types that are exchanged in a single HTTP transaction;
- The HTTP standards define what goes into these request and response messages so that everyone understands exchange resources.

HTTP Request example

- Source: https://wiki.wireshark.org/Hyper_Text_Transfer_Protocol

```
HTTP/1.1 200 OK
Date: Fri, 13 May 2005 05:51:12 GMT
Server: Apache/1.3.x LaHonda (Unix)
Last-Modified: Fri, 13 May 2005 05:25:02 GMT
ETag: "26f725-8286-42843a2e"
Accept-Ranges: bytes
Content-Length: 33414
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
```

HTTP Request Methods

- The GET is one of the primary HTTP methods;
- Every request message must include one of the HTTP methods, and the method tells the server what the request wants to do;
- An HTTP GET wants to get, fetch, and retrieve a resource;
- A list of common HTTP operators is shown in the following:
  - GET - Retrieve a resource
  - PUT - Store a resource
  - DELETE - Remove a resource
  - POST - Update a resource
  - HEAD - Retrieve the headers for a resource

HTTP Request Methods

- A web browser issues a GET request when it wants to retrieve a resource;
- GET requests are the most common type of request.
- A web browser sends a POST request when it has data to send to the server;
- POST requests are typically generated by a <form> on a webpage, like the form you fill out with <input> elements for address and credit card information.

HTTP Request Methods

- Web browsers typically treat GET and POST differently since GET is safe and POST is unsafe;
- It's OK to refresh a webpage retrieved by a GET request: the web browser will just reissue the last GET request and render whatever the server sends back;
- However, with HTTP POST request, the browser will warn us if we try to refresh the page;
- After a user clicks a button to POST information to a server, the server will process the information and respond with an HTTP redirect telling the browser to GET some other resource;
- The browser will issue the GET request, the server will respond with a "thank you for the order" resource, and then the user can refresh or print the page safely as many times as he or she would like.
- This is a common web design pattern known as the POST/Redirect/GET pattern.

HTTP Request Headers

- A full HTTP request message:

```
[method] [URL] [version]
[headers]
[body]
```

- The middle section contains one or more HTTP headers;
- Headers contain useful information that can help a server process a request;
- For example, if the client wants to see a resource in French:

```
GET http://odetocode.com/Articles/741.aspx HTTP/
1.1
Host: odetocode.com
Accept-Language: fr-FR
```

HTTP Request Headers

- Some of the more popular request headers:
  - User-Agent: Information about the user agent (the software) making the request;
  - Accept: Describes the media types the user agent is willing to accept;
  - Cookie: Contains cookie information. Cookie information generally helps a server track or identify a user.

The Response

- An HTTP response has a similar structure to an HTTP request. The sections of a response are:

```
[version] [status] [reason]
[headers]
[body]
```

- HTTP response:

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html; charset=utf-8
(…)
Date: Sat, 14 Jan 2012 04:00:08 GMT
Connection: close
Content-Length: 17151
<html>
(…)
</html>
```

Response Status Codes

- The status code is a number defined by the HTTP specification:
  - 100–199 - Informational
  - 200–299 - Successful
  - 300–399 - Redirection
  - 400–499 - Client Error
  - 500–599 - Server Error

Response Status Codes

- The HTTP status code indicates what is happening at the HTTP level;
- It doesn't necessarily reflect what happened inside your application.
- Some common examples:

Code	Reason	Description
200	OK	The status code everyone wants to se e. A 200 code in the response means e verything worked!
400	Bad Request	The server could not understand the re quest. The request probably used inco rrect syntax.
403	Forbidden	The server refused access to the reso urce.
404	Not Found	A popular code meaning the resource was not found.

- From an application perspective the request can be a failure, but from an HTTP perspective the request can be successfully processed.

Response Headers

- A response includes header information that gives a client metadata it can use to process the response;
- For example, the following content type is HTML, and the character set used to encode the type is UTF-8:

```
Content-Type: text/html; charset=utf-8
```

- The headers can also contain information about the server, like the name of the software and the version.

Connections

- Network communication, like many applications, consists of layers;
- Each layer in a communication stack is responsible for a specific and limited number of responsibilities;

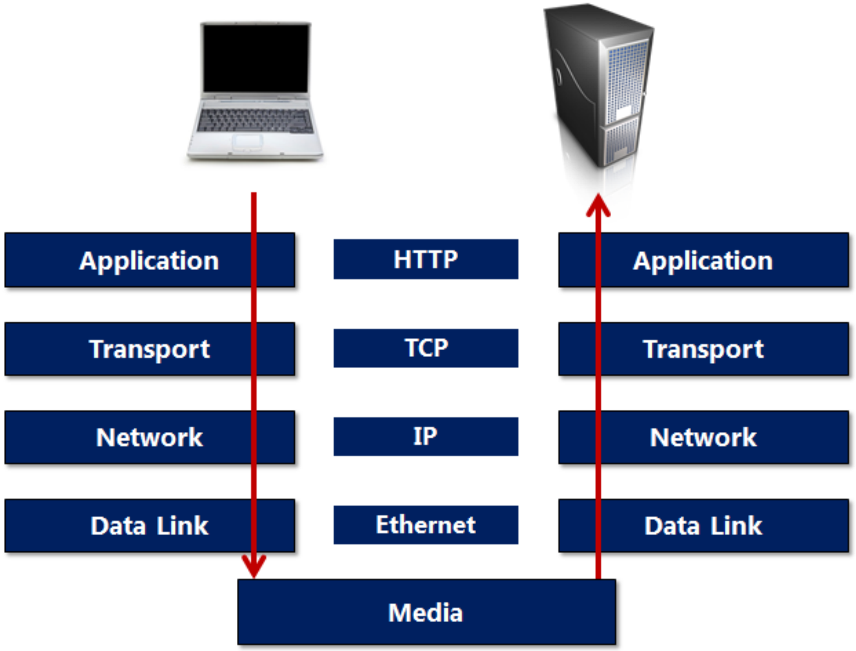


Image source: HTTP Succinctly - https://www.syncfusion.com/ebooks/http

State and Security

- Websites that want to track users will often turn to cookies;
- When a user first visits a website, the site can give the user's browser a cookie using an HTTP header;
- The browser then knows to send the cookie in the headers of every additional request it sends to the site;
- Assuming the website has placed some sort of unique identifier into the cookie, then the site can now track a user.

State and Security

- Cookies can identify users but cookies do not authenticate users;
- An authenticated user has proved his or her identity usually by providing credentials;
- Since cookies can track what a user is doing, they raise privacy concerns;
- Some users can disable cookies in their browsers, meaning the browser will reject any cookies a server sends in a response.

State and Security

- Disabled cookies present a problem for sites that need to track users;
- For example, one approach to place the user identifier into the URL;
- When a website wants to give a user a cookie, it uses a Set-Cookie header in an HTTP response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: fname=Scott$lname=Allen;
domain=.mywebsite.com; path=/
```

Setting Cookies

- A website can put any information into a cookie with a size limitation of 4 KB;
- However, many websites only put in a unique identifier for a user;
- A server can never trust anything stored on the client unless it is cryptographically secured;
- The browser will send the cookie to the server in every subsequent HTTP request:

```
GET ... HTTP/1.1
Cookie: GUID=00a48b7f6a4946a8adf593373e53347c;
...
```

Setting Cookies

- When the ID arrives, the server software can quickly look up any associated user data;
- One security concern around session identifiers is how they can open up the possibility of someone hijacking another user's session;
- We can guess that some user already has a SessionID of 11, and create an HTTP request with that ID just to see if I can steal or view the HTML intended for some other user.

Setting Cookies

- To avoid that, most web applications will use large random numbers as identifiers;
- Another security concern is how vulnerable cookies are to a cross-site scripting attack (XSS);
- In an XSS attack, a malicious user injects malevolent JavaScript code into someone else's website;
- The HttpOnly flag tells the user agent to not allow script code to access the cookie;
- Browsers that implement HttpOnly will not allow JavaScript to read or write the cookie on the client.

Authentication

- The process of authentication forces a user to prove her or his identity by entering credentials;
- At the network level, authentication typically follows a challenge/response format;
- The client needs to send a request and include authentication credentials for the server to validate;
- If the credentials are right, the request will succeed.

Authentication

- With basic authentication, the client will first request a resource with a normal HTTP message;
- The browser can send another request to the server, includin an Authorization header.

```
GET http://localhost/html5/ HTTP/1.1
Authorization: Basic bm86aXdvdWxkbnRkb3RoYXQh
```

- The value of the authorization header is the client's credentials in a base 64 encoding;
- Basic authentication is insecure by default, because anyone with a base 64 decoder can view the message;

Authentication

- Forms authentication is the most popular approach and doesn't use WWW-Authenticate or Authorization headers;
- The login page for forms-based authentication is an HTML form with inputs for the user to enter credentials;
- When the user clicks submit, the form values will POST to a destination for validation.

Authentication

- OpenID is an open standard for decentralized authentication;
- With OpenID, a user registers with an OpenID identity provider that stores and validate user credentials;
- There are many OpenID providers around, including Google.

Secure HTTP

- Previously we showed how anyone can read a message and understand what's inside, representing a security problem;
- Secure HTTP solves this problem by encrypting messages before the messages start traveling across the network;
- Secure HTTP is also known as HTTPS;
- The default port for HTTP is port 80, and the default port for HTTPS is port 443.

Secure HTTP

- HTTPS works by using an additional security layer in the network protocol stack;
- HTTPS requires a server to have a cryptographic certificate.

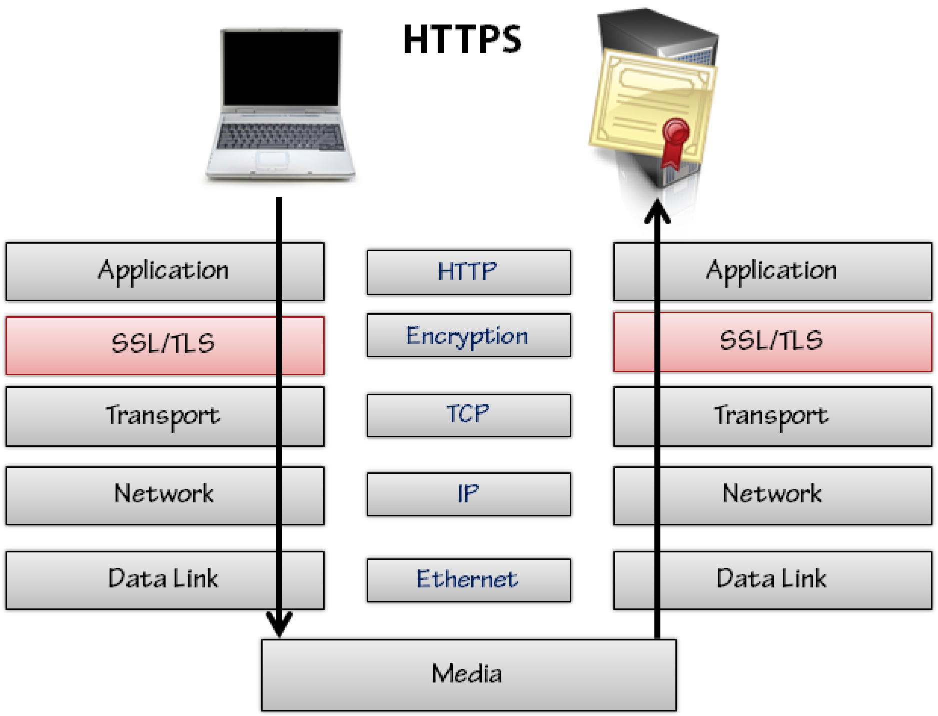


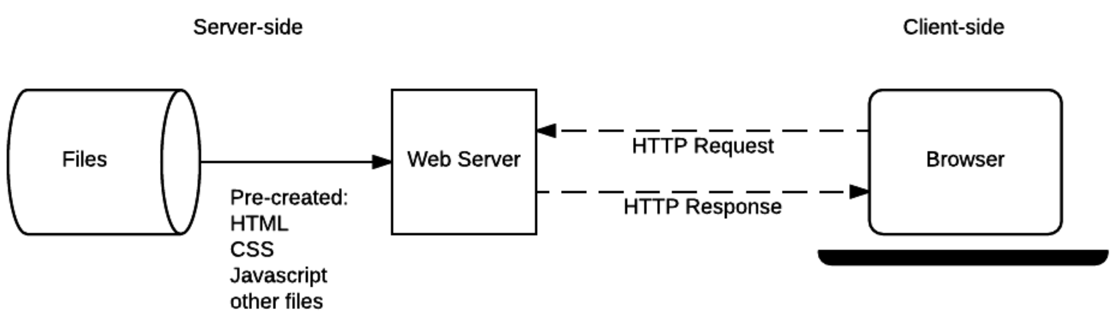
Image source: HTTP Succinctly - https://www.syncfusion.com/ebooks/http

References/Sources

- Source: HTTP Succinctly - https://www.syncfusion.com/ebooks/http
- https://wiki.wireshark.org/Hyper_Text_Transfer_Protocol
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

Server-side programming

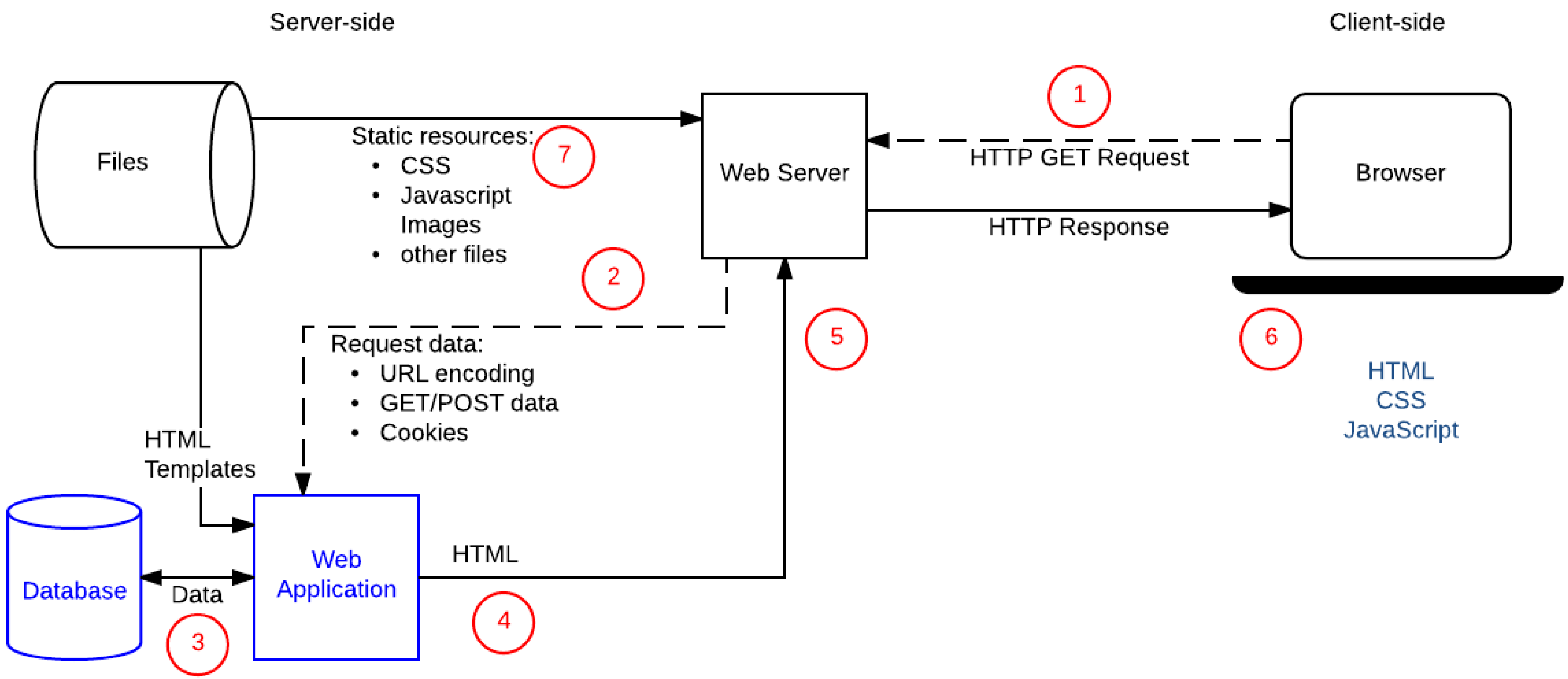
- In the previous slides we learn that Web browsers communicate with web servers using HTTP;
- There are two types of websites:
  - *static site*



https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

Server-side programming

- dynamic *site*



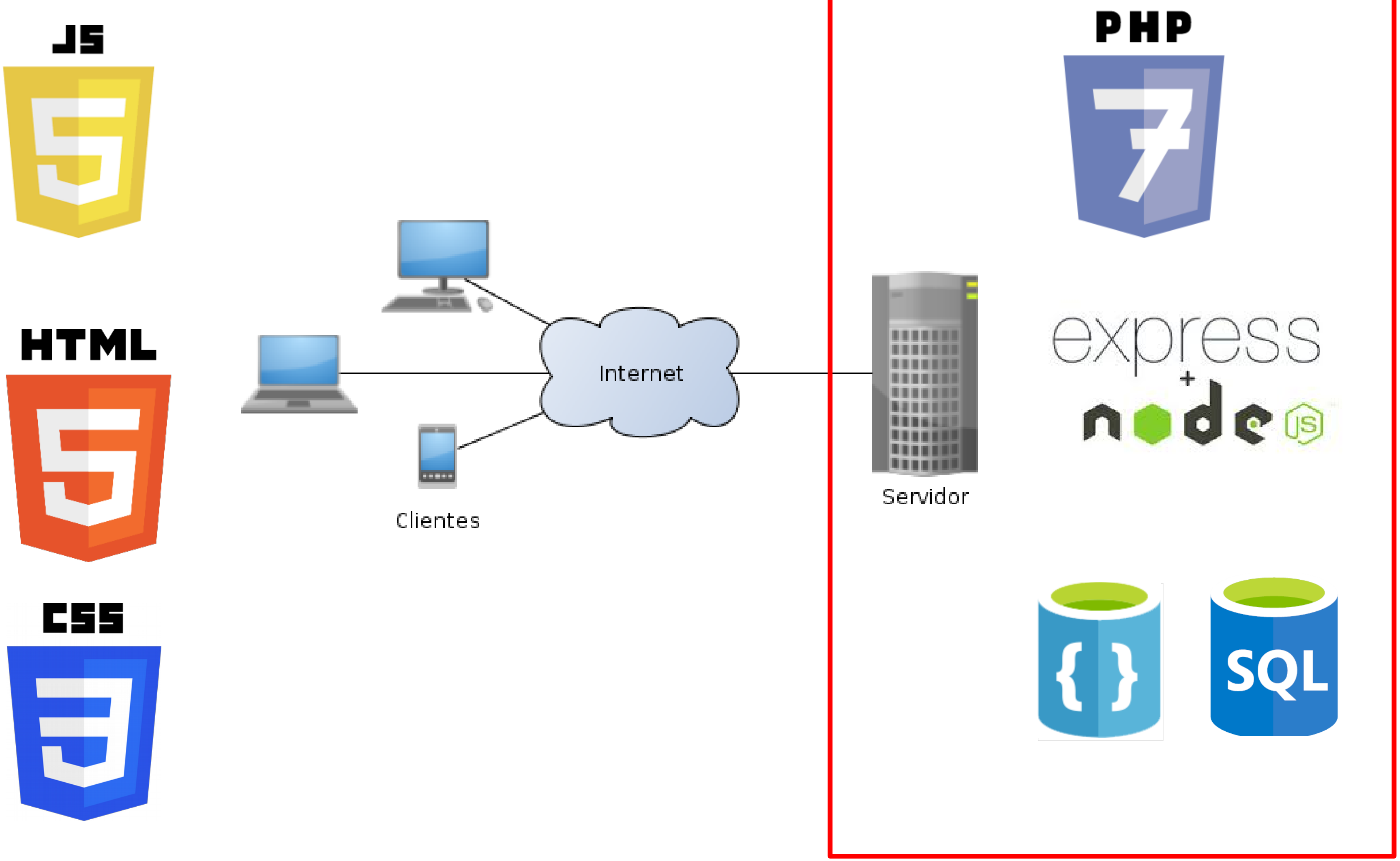
https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

Server-side programming

- Code running in the browser is known as client-side code and is primarily concerned with improving the appearance and behavior of a rendered web page: HTML, CSS and JavaScript;
- Server-side code can be written in any number of programming languages — examples of popular server-side web languages include PHP, Python, Ruby, and C#;
- The server-side code has full access to the server operating system and the developer can choose what programming language to use.

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

Server-side programming



Server-side programming

- What can you do on the server-side?
  - Efficient storage and delivery of information
  - Customised user experience
  - Controlled access to content
  - Store session/state information
  - Notifications and communication
  - Data analysis

NodeJs

- Node is an open-source, cross-platform, runtime environment that allows developers to create all kinds of server-side tools and applications using JavaScript;
- Node.js was born in 2009 from an idea of Ryan Dahl, who was searching for a way to track the time needed to upload a file from a browser without continuously asking the server "how much of the file is uploaded?";
- He explored non-blocking requests using the Google Chrome V8 engine as a base for writing a basic event loop and a low-level I/O API.

Instalation

- Download and install LTS version:
  - https://nodejs.org/en/
- To test, you can type node in the command line to enter the REPL interface. Once inside, you can try to execute some JavaScript code:

Node.js

```
> console.log("I am alive");
I am alive
```

REPL stands for Read Evaluate Print Loop. It is often used to learn a library or just to try some code. It consists of a command line executable (node in this case). You can run it by typing node in the terminal, which will give you access to the REPL. Here, you type any valid JavaScript statement and the REPL evaluates it for you immediately.

NodeJs

- Create the: index.js file and put the previous example inside;
- Run it using node index.js;
- You can create a simple web server to respond to any request using just the Node HTTP package:

```
// Load HTTP module
var http = require("http");

// Create HTTP server and listen on port 8000 for requests
http.createServer(function(request, response) {

   // Set the response HTTP header with HTTP status and Content type
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body "Hello World"
   response.end('Hello World\n');
}).listen(8000);

// Print URL for accessing server
console.log('Server running at http://127.0.0.1:8000/');
```

Just run:

```
node simpleServer.js

Enter the address:
localhost:8000 in your browser!
```

https://github.com/brunobmo/PAW17.18/blob/master/Nodejs/simpleServer/simpleServer.js

NodeJs

- This is a basic web server that responds “Hello World” to any incoming request;
- What it does is require from an external library the http module, create a server using the createServer function, and start the server on the port 8000;
- The incoming requests are managed by the callback of the createServer function;
- The callback receives the request and response objects and writes the header (status code and content type) and the string Hello World on the response object to send it to the client.

NodeJs

- Apart from the simplicity, the interesting part that emerges from the previous code is the asynchronicity;
- The first time the code is executed, the callback is just registered and not executed;
- The program runs from top to bottom and waits for incoming requests;
- The callback is executed every time a request arrives from the clients;
- Node.js is an event-driven, single-thread, non-blocking I/O platform for writing applications;

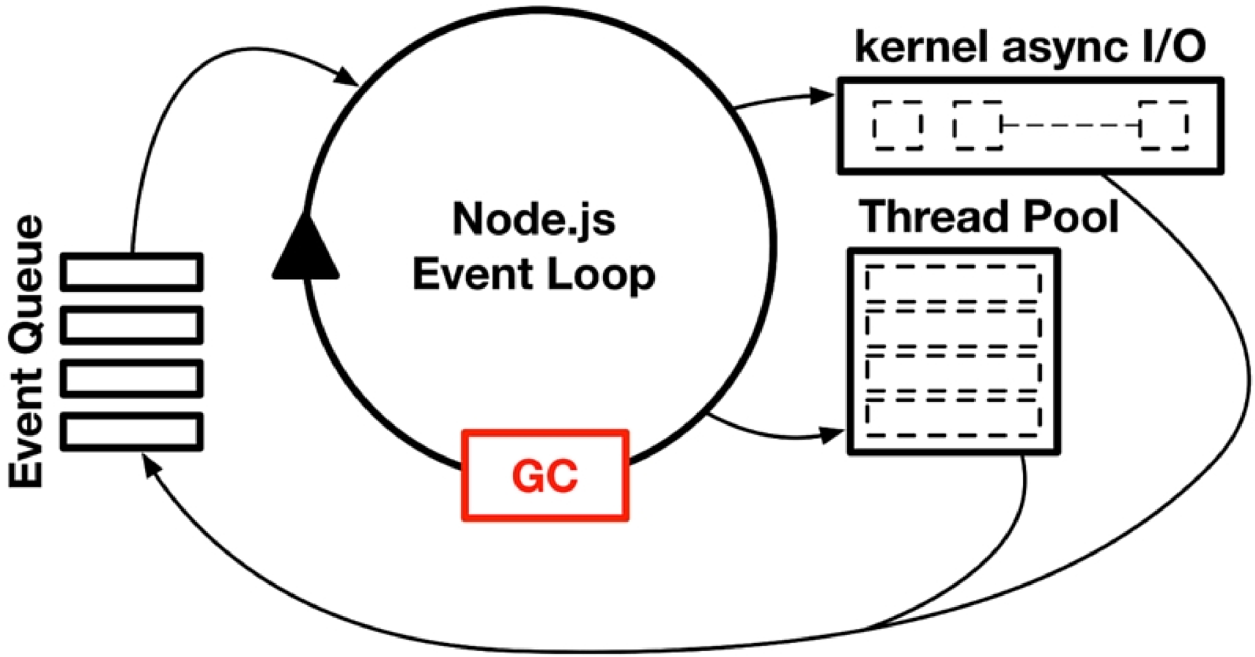
NodeJs

- Event driven
  - When something happens, it executes the code responsible for managing that event, and in the meantime it just waits, leaving the CPU free for other tasks;
- Single thread
  - Node.js is single thread;
  - Developers don’t need to deal with concurrency, cross-thread operations, variable locking, and so on;
- Non-blocking I/O
  - Every I/O request doesn’t block the execution of an application;
  - Every time the application accesses an external resource, for example, to read a file, it doesn’t wait for the file to be completely read;
  - It registers a callback that will be executed when the file is read and in the meantime leaves the execution thread for other tasks.

NodeJs

- The event loop
  - At the heart of the idea of non-blocking I/O is the event loop;
  - Consider the previous example of a simple web server: What happens when a request arrives before the previous one was served?
  - Remember that Node.js is single thread, so it cannot open a new thread and start to execute the code of the two requests in parallel;
  - It has to wait, or better yet, it puts the event request in a queue and as soon as the previous request is completed it dequeues the next one (whatever it is).
- Actually, the task of the Node engine is to get an event from the queue, execute it as soon as possible, and get another task;
- Every task that requires an external resource is asynchronous, which means that Node puts the callback function on the event queue.

Event loop



https://www.nearform.com/blog/performance-reaching-ludicrous-speed/

Event loop

- Consider another example, a variation of the basic web server that serves a **static file** (an index.html):

```
var http = require('http')
var fs = require('fs')

var server = http.createServer((request, response) => {
    response.writeHead(200, {'Content-Type': 'text/html'});
    fs.readFile('./index.html', (err, file) => {
        response.end(file);
    })
})

server.listen(8000)
```

- When a request arrives to the server, a file must be read from the filesystem;
- The readFile function (like all the async functions) receives a callback with two parameters that will be called when the file is actually read.
- This means that the event “the file is ready to be served” remains in a queue while the execution continues;
- So, even if the file is big and needs time to be read, other requests can be served because the I/O is non-blocking.

Event loop

- Consider another example that executes a callback function every specified number of milliseconds:

```
setInterval(() => console.log('function 1'), 1000)
setInterval(() => console.log('function 2'), 1000)
console.log('starting')
```

- The result is:

```
starting
function 1
function 2
function 1
function 2
function 1
function 2
function 1
function 2
```

https://github.com/brunobmo/PAW17.18/blob/master/Nodejs/examples/example1.js

Event loop

- Now we can try to modify the code:

```
setInterval(() => console.log('function 1'), 1000)
setInterval(() => {
    console.log('function 2')
    while (true) { }
    }, 1000)
console.log('starting')
```

- When it is its turn, function 2 will be executed and it will never release the thread, so the program will remain blocked on the while cycle forever;
- This is why it is very important that our code is fast, because as soon as the current block finishes running, it can extract another task from the queue.
- This problem is solved quite well with asynchronous programming.

https://github.com/brunobmo/PAW17.18/blob/master/Nodejs/examples/example2.js

Video recomendation: https://www.youtube.com/watch?v=O1mx9WO7PAI

How to create asynchronous code?

- Synchronous example:

```
function concat(a, b){
    let r = a + b;
    return r;
}
function upper(a){
    let r = a.toUpperCase();
    return r;
}
let result = upper(concat("Hello", " World"));
console.log(result);

console.log("Next Task");
```

- We can use the **setImmediate** function that, even if it is called immediate, puts the event in the queue and continues with the execution.

https://github.com/brunobmo/PAW17.18/blob/master/Nodejs/examples/example3.js

How to create asynchronous code?

- Asynchronous example:

```
function concat(a, b){
    setImmediate(function(){
        let r = a + b;
        return r;
    });
}
function upper(a){
    setImmediate(function(){
        let r = a.toUpperCase();
        return r;
    });
}
let concatResult=concat("Hello", " World");
let upperResult = upper(concatResult);
console.log(upperResult);

console.log("Next Task");
```

- What happened?

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

https://github.com/brunobmo/PAW17.18/blob/master/Nodejs/examples/example4.js

How to create asynchronous code?

- Use the callback!

```
function concat(a, b, callback){
    setImmediate(function(){
        let r = a + b;
        callback(r);
    });
}
function upper(a, callback){
    setImmediate(function(){
        let r = a.toUpperCase();
        callback(r);
    });
}
concat("Hello", " World", r1 => {
    upper(r1, r2 =>{
        console.log(r2);
    })
});

console.log("Next Task");
```

- Output:

```
Next Task
HELLO WORLD
```

https://github.com/brunobmo/PAW17.18/blob/master/Nodejs/examples/example5.js

References

- https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

- Source: Book – Node Js Succinctly: https://www.syncfusion.com/ebooks/nodejs

- https://www.udemy.com/node-js-training-and-fundamentals/learn/v4/content

- https://www.codecademy.com/learn/learn-express

Next week

- NodeJs - Express Framework (introduction)
- NodeJs - WebSockets
- NodeJs - Jade
- NodeJs - MiddleWare

JAVASCRIPT

GOF – Design Patterns
HTTP protocol
NodeJs – Introduction

Bruno Oliveira: bmo@estg.ipp.pt
Marco Gomes: mfg@estg.ipp.pt
Miguel Andrade: mja@estg.ipp.pt