

P.PORTO



POLITÉCNICO
DO PORTO
ESMAD

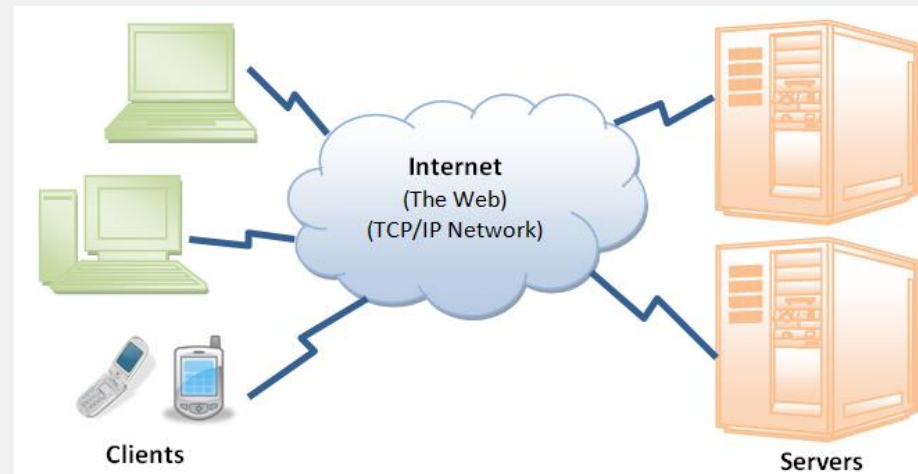
PROGRAMAÇÃO WEB II
TSIW

SUMMARY

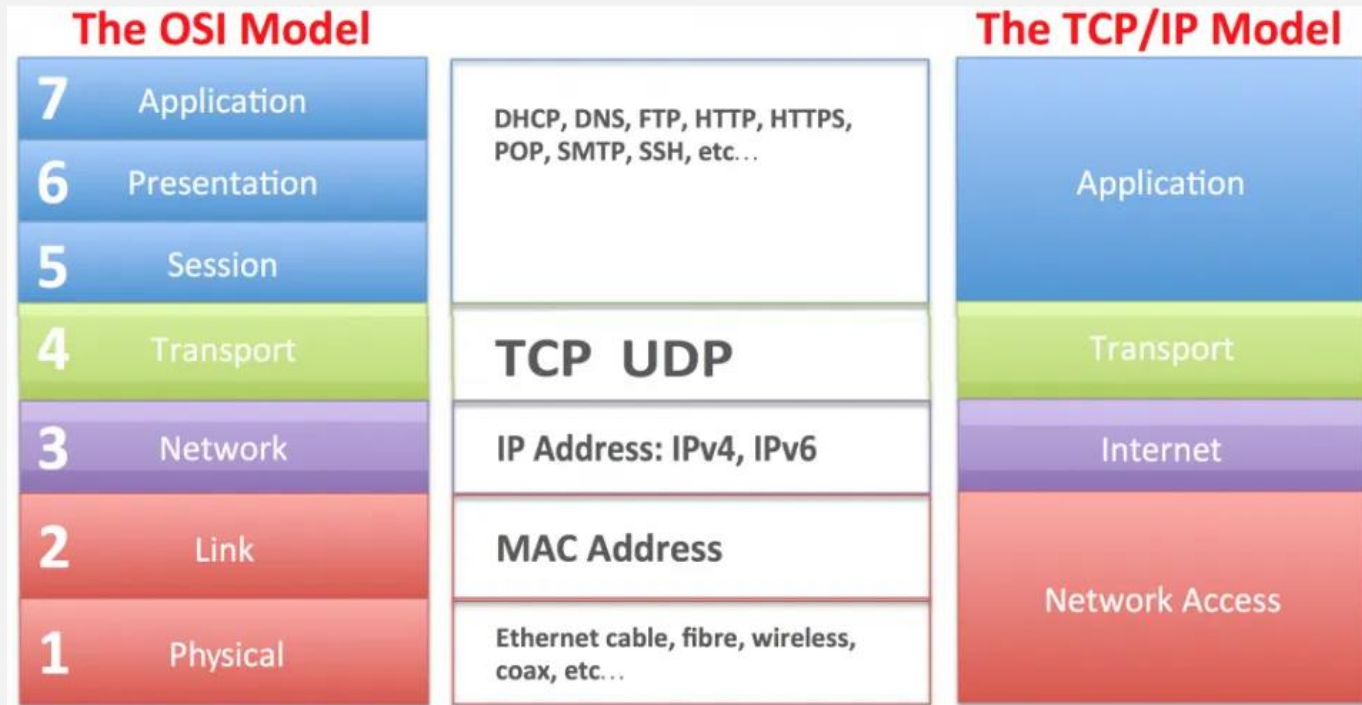
- Web servers
- Client-server architecture
- Server-side web frameworks
- Introduction to Node.js
- NPM and the package.json file
- HTTP and Node modules

The Internet

- Internet is a **massive distributed client/server information system**
- Many applications are running concurrently over the Internet
 - web browsing, e-mail, file transfer, audio/video streaming, ...
- The Internet is a technical infrastructure which allows billions of computers to be connected all together; among those, some computers (**Web servers**) can send messages intelligible to web browsers
- The **Internet** is an infrastructure, whereas the **Web** is a service built on top of the infrastructure



The Internet & the WWW

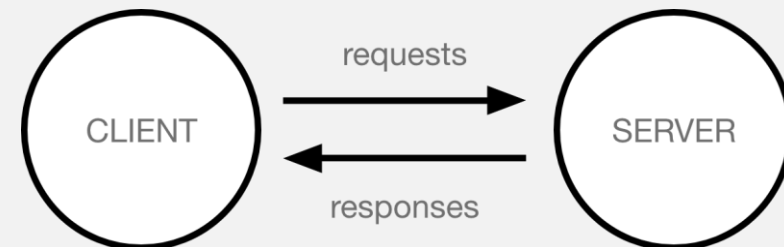


OSI - Open Systems Interconnection model: open standard for all communication systems. Splits the communications between a computing system into 7 different abstraction layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application. Each intermediate layer serves a class of functionality to the layer above it and is served by the layer below it

TCP/IP protocol suite: Internet uses TCP/IP protocol suite, also known as Internet model, which contains four layered architecture

How the Web works

- Computers (and other devices) connected to the web can be named as **clients** or **servers**
- A simplified way of interacting with each other can be outlined in this way:



- Clients are the typical **web user's internet-connected devices** (for example, a computer connected to a Wi-Fi network or a phone connected to a mobile network), with **web-accessing software** available on those devices (usually a web browser)
- Servers are computers that store web pages, websites or applications; when a client device wants to access a web page, a copy is downloaded from the server onto the client machine to be displayed in the user's web browser

How the Web works

- Clients and servers don't tell the whole story about how the internet works
- Imagine that the web is a road, with the client at home and the server as a shop the client wants to buy something from
 - **Internet connection**: allows sending and receiving data on the web; it's basically like the street between the client's house and the shop
 - **TCP/IP (*Transmission Control Protocol/Internet Protocol*)**: communication protocols that define how data should travel across the internet; is like the transport mechanisms in the example (car or bicycle)



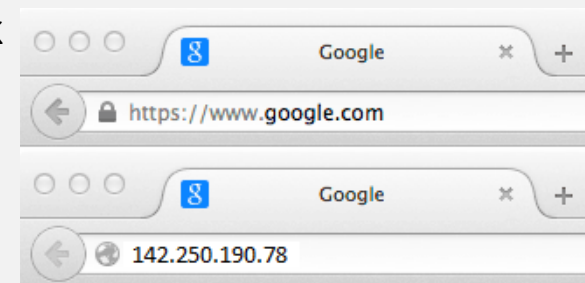
How the Web works

- Imagine that the web is a road, with the client at home and the server as a shop the client wants to buy something from



- **DNS (Domain Name Server)**: are like an address book for websites; when the user types a web address in a browser, the browser looks at the DNS to find the website's real address, so it can send HTTP messages to the right place; it is like looking up the address of the shop

- **HTTP (Hypertext Transfer Protocol)**: application protocol that defines a language for clients and servers to speak to each other; it is like the language used to order the goods from the shop



How the Web works

- Imagine that the web is a road, with the client at home and the server as a shop the client wants to buy something from



- **Component files:** a website can be made up of many different files, which are like the different parts of the goods the client buy from the shop; these files come in two main types
 - **Code files:** websites are built primarily from HTML, CSS, and JavaScript (among others)
 - **Assets:** collective name for all the other stuff that makes up a website, such as images, music, video, PDF files...

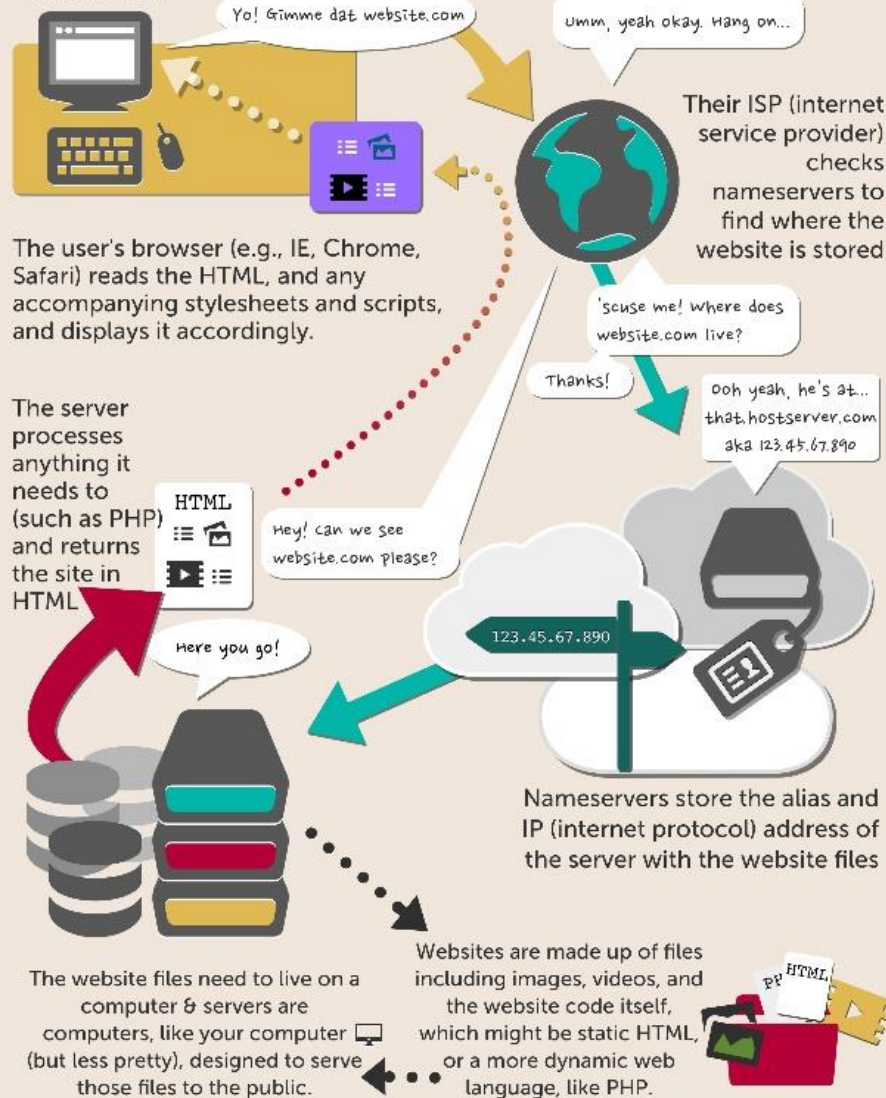


How the Web works

- When someone types a web address into a browser:
 1. The browser goes to the **DNS server**, and finds the real address of the server that the website lives on
 2. The browser sends an **HTTP request message** to the server, asking it to send a copy of the website to the client. This message, and all other data sent between the client and the server, is sent across the internet connection using **TCP/IP**
 3. If the server approves the client's request, the server sends the client a "200 OK" message, which means "Of course you can look at that website! Here it is", and then starts sending the website's files to the browser as a series of small chunks called data packets
 4. The browser assembles the small chunks into a complete web page and displays it

How do websites work?

A user connects to the internet and asks for an address



How The Web Works - The Big Picture

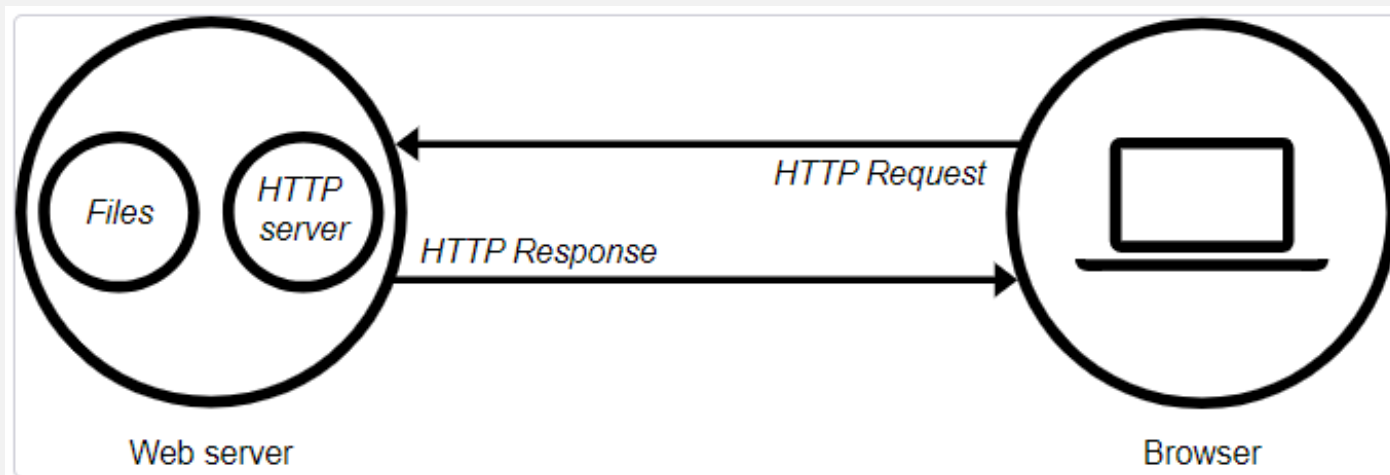
<https://www.youtube.com/watch?v=hjHvdBlSxug>

Web servers

- The term web server can refer to both hardware or software:
 - HW: a web server is a computer that stores web server **software and a website's component files**; a web server connects to the Internet and supports physical data interchange with other devices connected to the web
 - SW: a web server includes several parts that control how web users access hosted files; at a minimum, this is an HTTP server, that understands **URLs – Uniform Resource Locators** (web addresses) and **HTTP – HyperText Transfer Protocol** (the protocol a browser uses to view webpages)

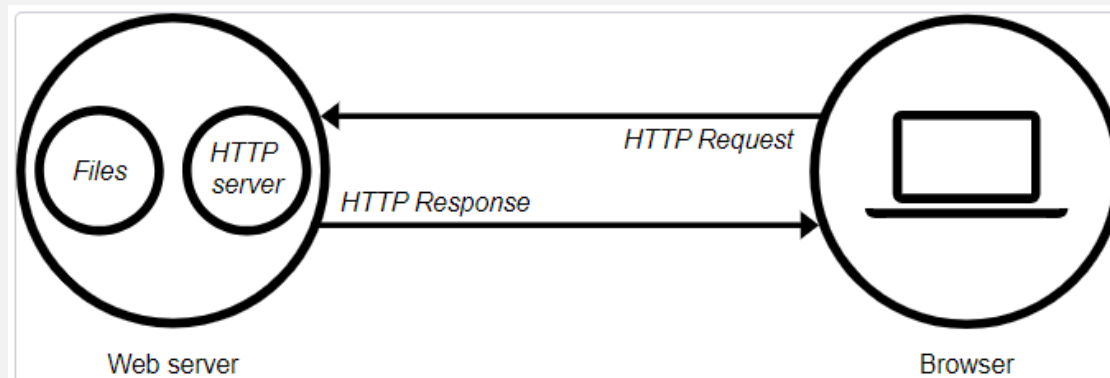
Web servers

- Web browsers communicate with web servers using **HTTP**
 - When we click a link on a web page, submit a form, or run a search, an **HTTP request** is sent from the browser to the target server
 - The request includes a **URL** to the affected resource and a **method** that defines the required action (for example, to get, delete, or post the resource), and may include additional information encoded in URL parameters, body data or in cookies



Web servers

- Web servers wait for client request messages, process them when they arrive, and reply to the web browser with an **HTTP response** message
 - A response contains a status line indicating whether or not the request succeeded (e.g., "200 OK" for success or "404 Not found" for error)
 - The body of a successful response would contain the requested resource (e.g. a new HTML page, or an image, etc...), which could then be displayed by the web browser

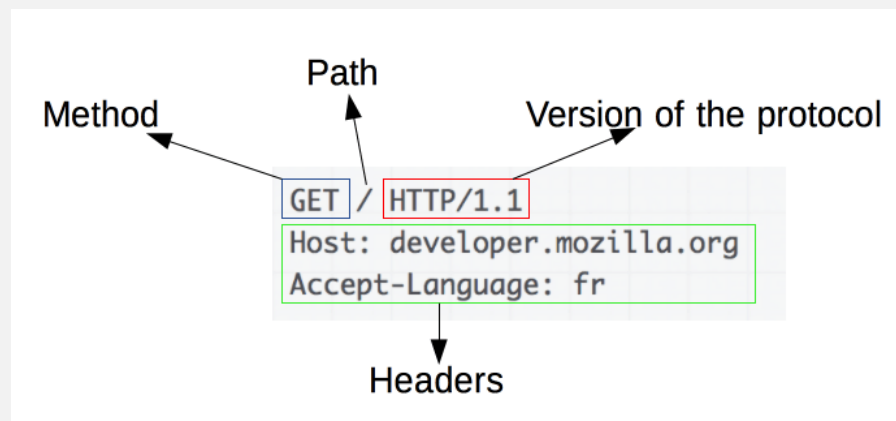


Communicating through HTTP

- The **HTTP** is an **application-layer protocol** (set of rules) that specifies how to transfer hypertext (linked web documents) between two computers, following the classical client-server model; it is:
 - Textual: all commands are plain-text and human-readable
 - Stateless: neither the server nor the client remember previous communications
E.g: relying on HTTP alone, a server can't remember a password a user typed or remember its progress on an incomplete transaction; the server application is required for tasks like that

Communicating through HTTP

- An **HTTP request** contains:
 - HTTP **method**: GET, POST, ...
 - **URI**: URL of the resource stripped from elements that are obvious from the context, like protocol (HTTP), domain (host) or TCP port
 - HTTP **headers**: pass additional information
 - **Body** (or **payload** – only for some methods)



Communicating through HTTP

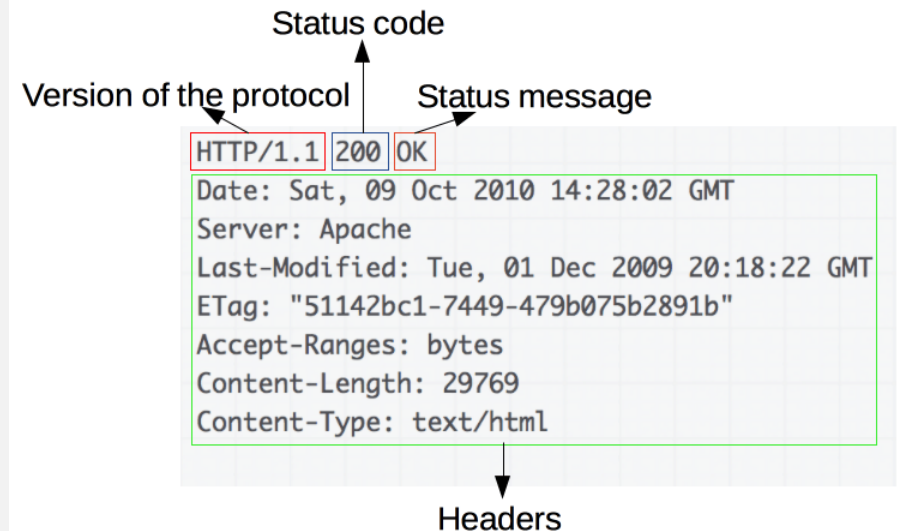
- HTTP methods (or verbs/nouns):
 - **GET**: used to get a resource; requests using GET should only retrieve data
 - **POST**: used to create a resource (data sent in the request body or payload);
 - **PUT**: used to edit the specified resource (data sent in the request body);
 - **DELETE**: used to delete the specified resource;
 - **PATCH**: used to partially edit a feature on the specified resource;
 - **OPTIONS**: used to receive all HTTP methods that can be used in a resource
 - There are other verbs like **HEAD**, **CONNECT** and **TRACE**

Communicating through HTTP

- An HTTP method is:
 - **safe**: the request using this method doesn't alter the state of the server (read-only operation) - GET, HEAD, OPTIONS, TRACE
 - **idempotent**: if multiple identical requests using this method will have the same effect on the server as that of a single request of that same method - GET, HEAD, OPTIONS, TRACE, PUT, DELETE
 - **cacheable**: if the response to a request using this method is allowed to be stored for future use, i.e. cached - GET, HEAD, POST (sometimes)

Communicating through HTTP

- An **HTTP response** contains:
 - **Status code**: indicates if the request was successful or not, and why
 - **Status message**: short description of the status code
 - HTTP **headers**: act like in the requests
 - **Body**: optionally, containing the fetched resource



Communicating through HTTP



Opera Browser

CLIENT REQUEST

GET	/	HTTP/1.1
-----	---	----------

Host:	www.opera.com
-------	---------------



Opera Server

SERVER RESPONSE

HTTP/1.1	200	OK
----------	-----	----

Date:	Wed, 23 Nov 2011 19:41:37 GMT
-------	-------------------------------

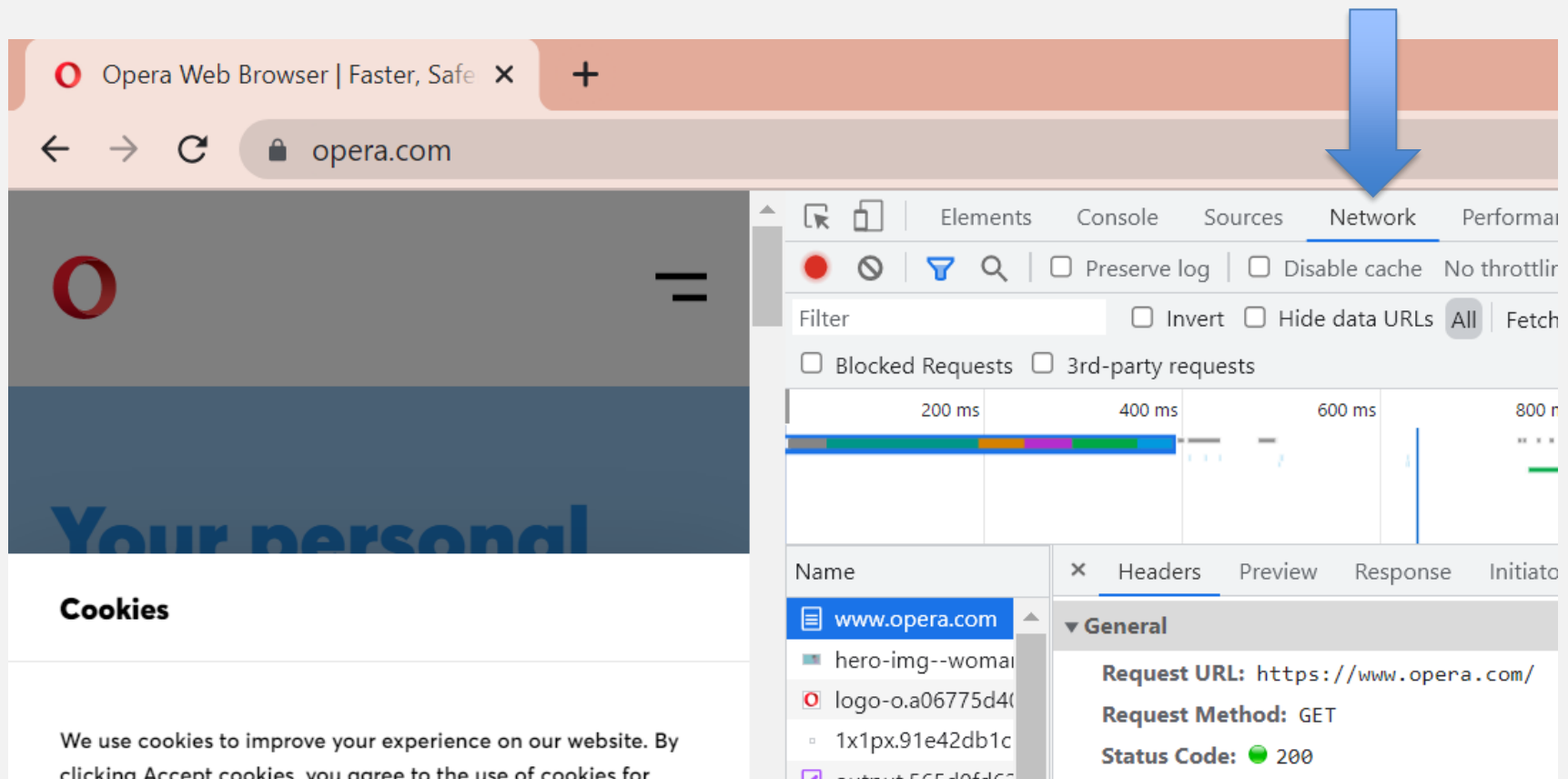
Content-Type:	text/html; charset=utf-8
---------------	--------------------------

Server:	Apache
---------	--------

```
<!DOCTYPE html>  
<html lang="en">
```

...

Communicating through HTTP



Opera Web Browser | Faster, Safe x +

← → ↻ opera.com

Network

Filter ☐ Invert ☐ Hide data URLs ☒ All Fetch

☐ Blocked Requests ☐ 3rd-party requests

200 ms 400 ms 600 ms 800 ms

Name

- www.opera.com
- hero-img--woma
- logo-o.a06775d4
- 1x1px.91e42db1c
- output.555d0f4c

Headers Preview Response Initiato

General

Request URL: <https://www.opera.com/>

Request Method: GET

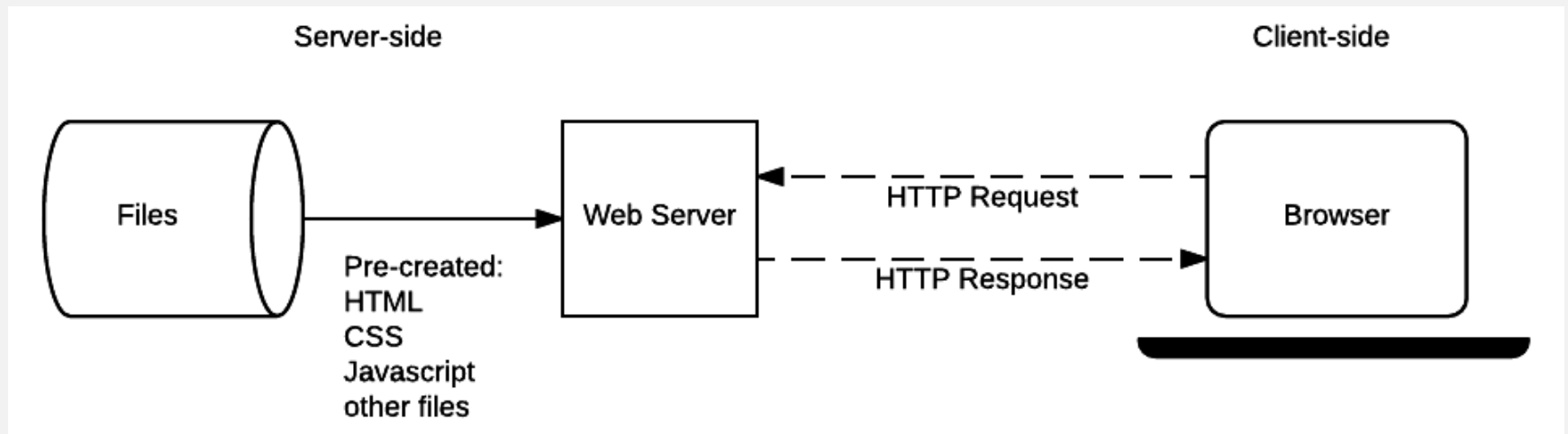
Status Code: 200

Cookies

We use cookies to improve your experience on our website. By clicking Accept cookies, you agree to the use of cookies for

Server-side programming

- Static websites

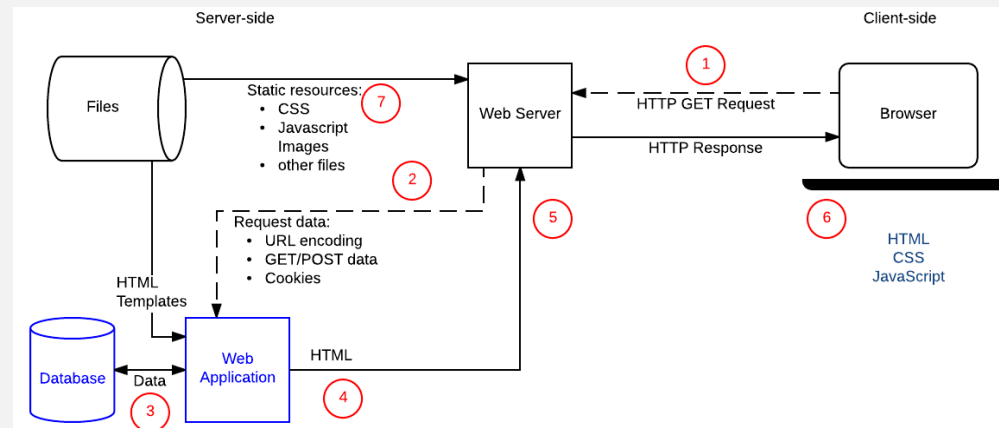


- A static website returns the same hard-coded content from the server whenever a particular resource is requested

Server-side programming

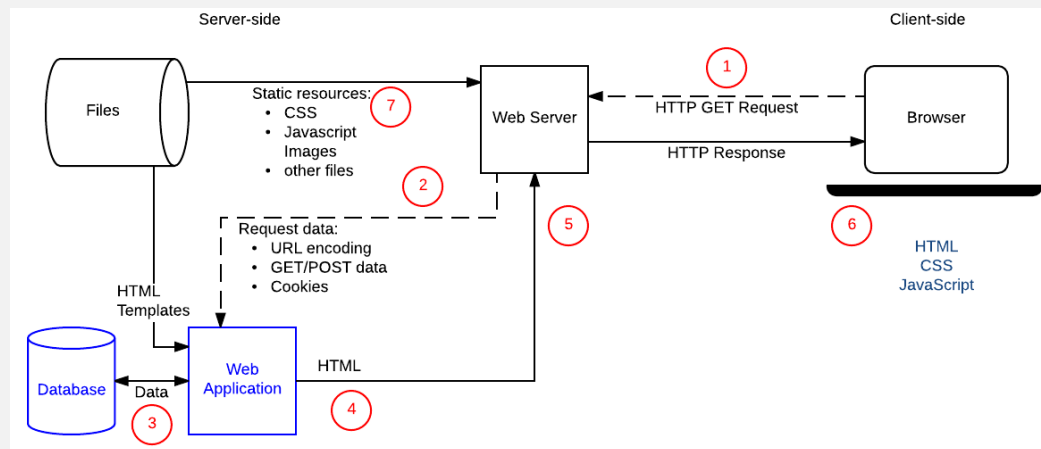
- **Dynamic websites**

- HTML pages are normally created by **inserting data from a database into placeholders in HTML templates** (a much more efficient way of storing large amounts of content than using static websites)
- A dynamic site can return **different data** for a URL based on information provided by the user or stored preferences and can perform other operations as part of returning a response (e.g. sending notifications)
- Most of the code to support a dynamic website must run on the server; creating this code is known as **"server-side programming"** (or "backend programming")



Server-side programming

- **Dynamic websites:** in a dynamic website some of the response content is generated dynamically, only when needed



- Requests for static resources (7) are handled in the same way as for static sites (static resources are any files that don't change)
- Requests for dynamic resources are forwarded to server-side code (2); the server interprets the request, reads required information from the database (3), combines the retrieved data with HTML templates (4) and sends back a response containing the generated HTML (5,6)

Server-side programming

What is it? **How does it differ from the client-side?** Why it is so useful?

- **Client-side** code and is primarily concerned with **improving the appearance and behavior** of a rendered web page
 - selecting and styling UI components, creating layouts, navigation, form validation, etc.
- **Server-side** website programming mostly involves **choosing which content is returned** to the browser in response to requests
 - The server-side code handles tasks like validating submitted data and requests, using databases to store and retrieve data and sending the correct data to the client as required

Server-side programming

- Client-side code is written using HTML, CSS, and JavaScript, it is run inside a web browser and has little or no access to the underlying operating system (including limited access to the file system)
 - part of the challenge of client-side programming is handling differences in browser support gracefully
- Server-side code can be written in many number of programming languages (PHP, Python, Ruby, C#, NodeJS - JS) and has full access to the server operating system

Server-side programming

- Web developers typically use web frameworks
 - Web frameworks are collections of functions, objects, rules and other code constructs designed to solve common problems, speed up development, and simplify the different types of tasks
- While both client and server-side code use frameworks, the domains are very different
 - Client-side frameworks simplify layout and presentation tasks
 - Server-side frameworks provide support for sessions, user authentication, easy database access, templating libraries, etc.

Server-side programming

- Client-side frameworks are often used to help and speed up development, but one can also choose to write all the code by hand
 - in fact, writing the code by hand can be quicker and more efficient if one only needs a small and simple UI
- In contrast, one would almost never consider writing the server-side component without a framework
 - implementing a vital feature like an HTTP server is really hard to do from scratch

Server-side programming

What is it? How does it differ from the client-side? **Why it is so useful?**

- Efficiency in storing and delivering information
 - How many products are available on Amazon or how many posts have been written on Facebook? Creating a separate static page for each product or post would be completely impractical!
 - Server-side programming allows to instead store the information in a database and **dynamically construct** and return HTML and other types of files
 - It is also possible to **return data** (JSON, XML, etc.) for rendering by appropriate client-side web frameworks (this reduces the processing burden on the server and the amount of data that needs to be sent)

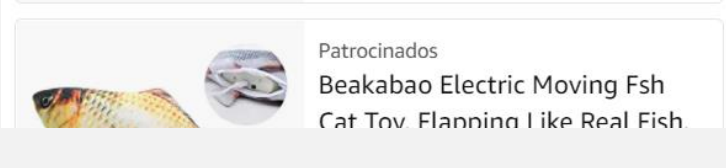
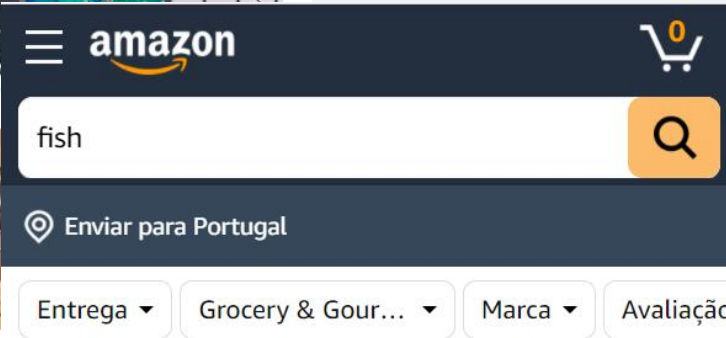
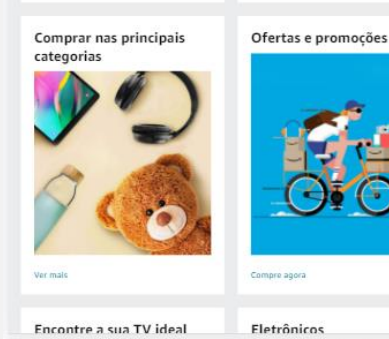
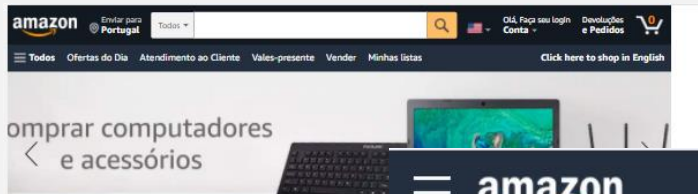
Server-side programming

Note: Your imagination doesn't have to work hard to see the benefit of server-side code for efficient storage and delivery of information:

1. Go to [Amazon](#) or some other e-commerce site.
2. Search for a number of keywords and note how the page structure doesn't change, even though the results do.
3. Open two or three different products. Note again how they have a common structure and layout, but the content for different products has been pulled from the database.

For a common search term ("fish", say) you can see literally millions of returned values. Using a database allows these to be stored and shared efficiently, and it allows the presentation of the information to be controlled in just one place.

Server-side programming



Server-side programming

- Controlled access to content
 - Server-side programming allows sites to **restrict access to authorized users** and serve only the information that **a user is permitted to see**
- Store session / state information
 - Use of sessions in backend programming, a mechanism that allows a server to **store information** and send different responses based on that information

Server-side programming

- Notifications and communication
 - Servers can send general or user-specific **notifications** through the website itself or via email, SMS, instant messaging, video conversations, or other communications services
- Data analysis
 - A website may **collect a lot of data about users**: what they search for, what they buy, what they recommend, how long they stay on each page; server-side programming can be used to **refine responses based on analysis of this data**

Server-side frameworks

- A.k.a “*web application frameworks*”, make it easier to write, maintain and scale web applications, because they provide tools and libraries that simplify common web development tasks
- Provide a simplified syntax to generate **server-side code to work with the HTTP requests and responses** between server and browser
- Most sites will provide several different resources, accessible through distinct URLs; web frameworks provide simple **mechanisms to map URL patterns** to specific handler functions
 - Benefits in terms of maintenance, because you can change the URL used to deliver a particular feature without having to change the underlying code

Server-side frameworks

- **Data can be encoded in an HTTP request** in a few ways; web frameworks provide programming-language-appropriate mechanisms to access this information
- Websites use databases to store information both to be shared with users, and about users; frameworks often provide a **database layer that abstracts database read, write, query, and delete operations**
 - This abstraction layer is referred to as an ORM (*Object Relational Mapper*) or ODM (*Object Document Mapper*)
- Frameworks often provide **templating systems**, specifying the structure of an output document, by using placeholders for data that will be added when a page is generated

Server-side frameworks

- There are so many, almost as the many as programming languages:
 - Django (Python)
 - Flask (Python)
 - Express (Node.js / Javascript)
 - Deno (Javascript)
 - Ruby on Rails (Ruby)
 - Laravel (PHP)
 - ASP.NET (C#, Visual Basic,...)
 - Spring Boot (Java)
- How to choose?

Server-side frameworks

- Factors to consider when choosing:
 - **Effort to learn**: familiarity with the underlying programming language, consistency of its API, quality of its documentation, and size and activity of its community
 - **Productivity**: measure of how quickly one can create new features once is familiar with the framework, and includes both the effort to write and maintain code
 - **Performance**: usually "speed" is not the biggest factor in selection because even relatively slow runtimes are more than "good enough" for mid-sized sites running on moderate hardware
 - **Suporte a cache**: caching is an optimization where you store all or part of a web response so that it does not have to be recalculated on subsequent requests; returning a cached response is much faster

Server-side frameworks

- Factors to consider when choosing :
 - **Scalability**: on a website with many accesses the benefits of caching will easily be exhausted and even reach the limits of vertical scaling (running the web application on more powerful hardware); at this point one may need to scale horizontally (share the load by distributing the site across a number of web servers and databases) or scale "geographically" (some of the clients will be located too far from the original server)
 - **Security**: some web frameworks provide better support for handling common web attacks (Cross-Site Scripting (XSS), SQL injection, Cross-Site Request Forgery (CSRF),...)

Node.js: introduction



- [Node.js](#): JavaScript interpreter that works on the server side
- Designed to optimize **performance** and **scalability** in web applications and is a good solution for many common development problems (e.g., real-time applications)
- The code is written in "plain old **JavaScript**", which means that less time is spent dealing with "changing context" between languages for those who write code on the client side and on the server side
 - JavaScript is a relatively new programming language and benefits from **language design improvements** when compared to other more traditional web server languages (e.g. Java, PHP, etc.)

Node.js: introduction

- The **NPM package manager** provides access to hundreds of thousands of reusable packages, facilitates dependency resolution and automates most of the tool chain in building an application
- Consists of a command line client, also called **npm**, and an online database of packages, called the npm registry
- Well supported by many web hosting providers, who often provide specific infrastructure and documentation for hosting Node websites
- Very active developer and user community

Node.js: introduction

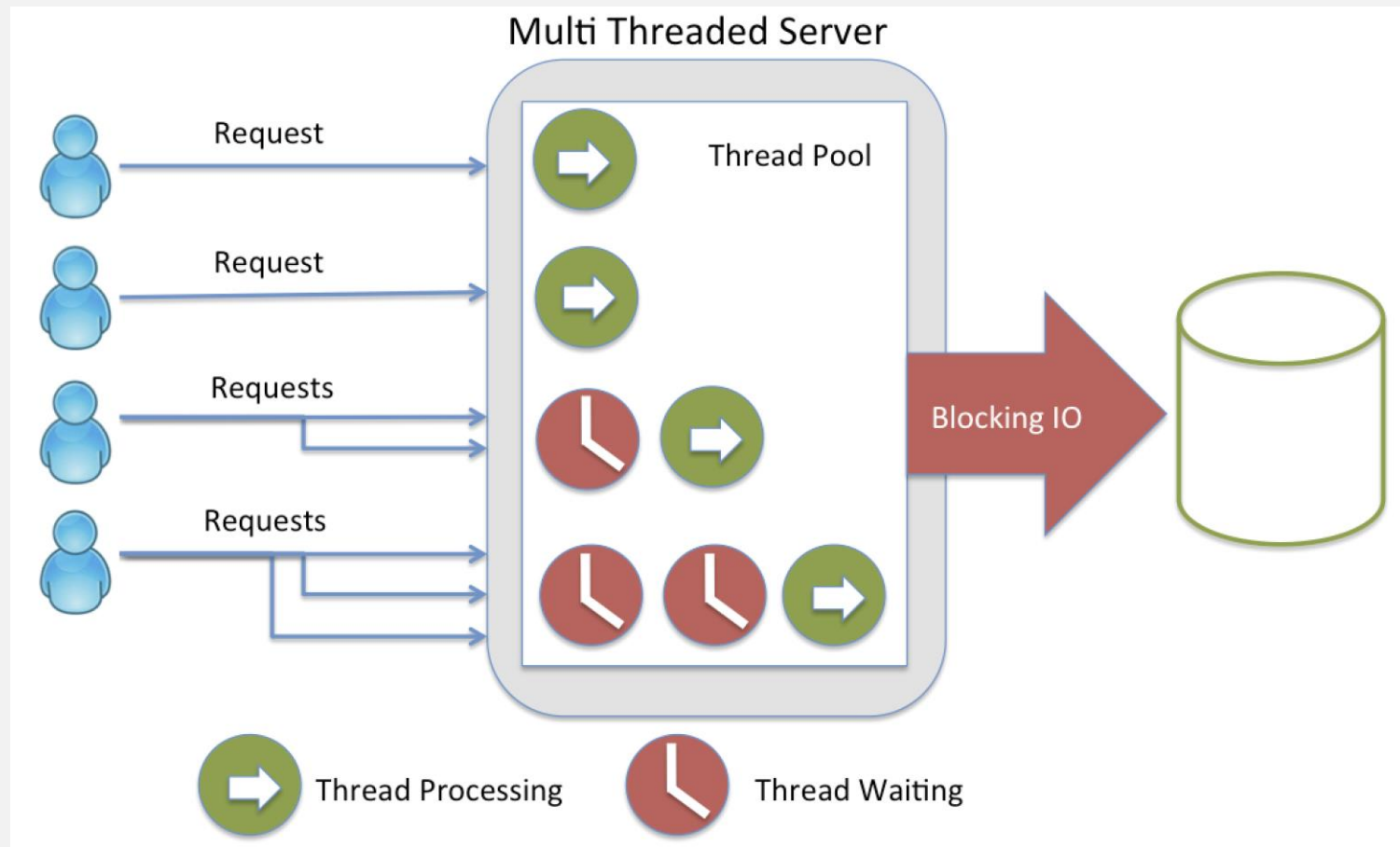
- **Single-thread**

- a Node.js app runs in a single process (thread), without creating a new thread for every request
- as it uses only one thread, it is not recommended to use it to deal with complex CPU-intensive algorithms (image editing, video rendering, heavy mathematical calculations) as this prevents other actions from being performed until processing is complete

- **Event-loop / Non blocking I/O**

- Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JS code from blocking
- generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm

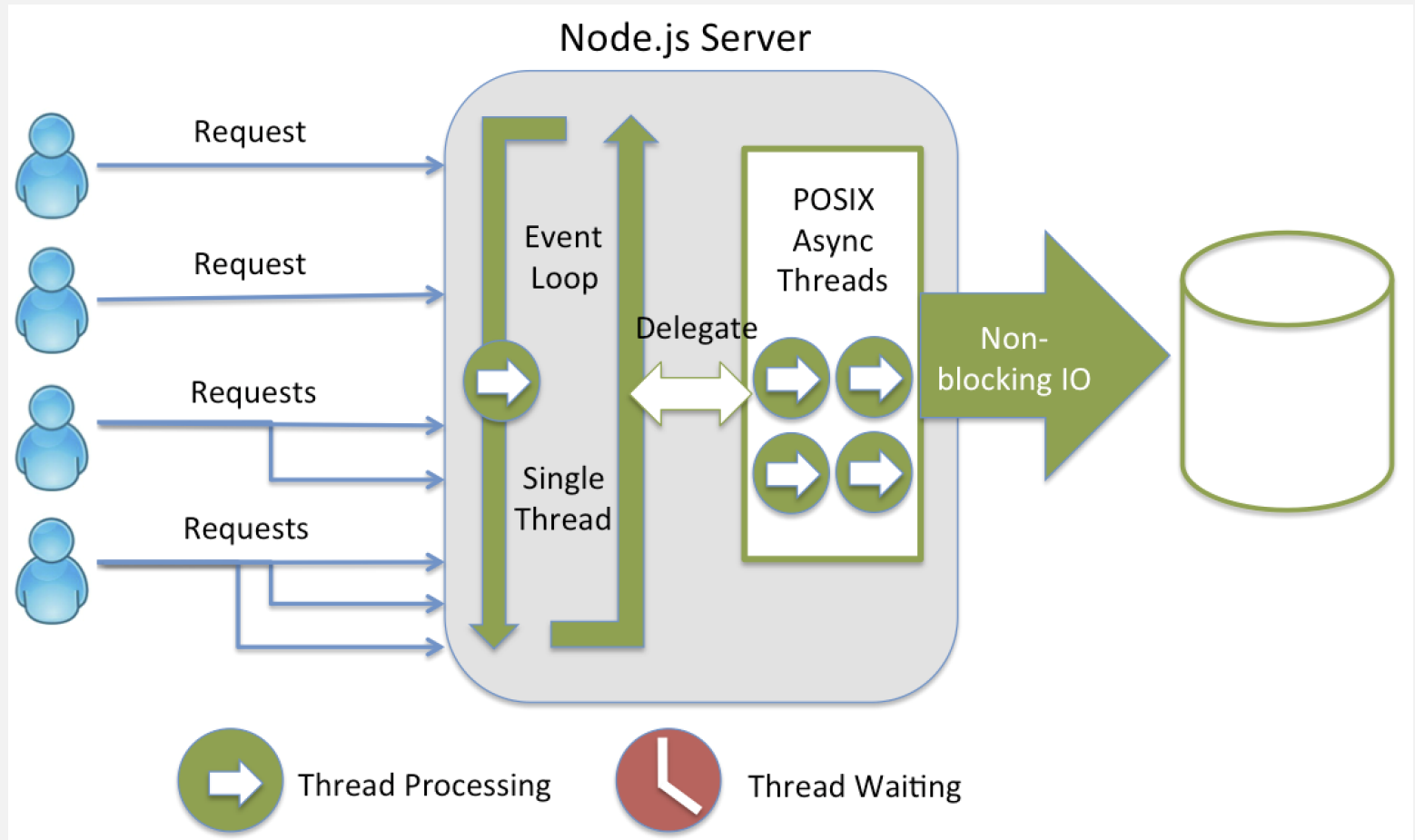
Node.js: introduction



Multi-thread programming

(source: <https://strongloop.com/strongblog/node-js-is-faster-than-java/>)

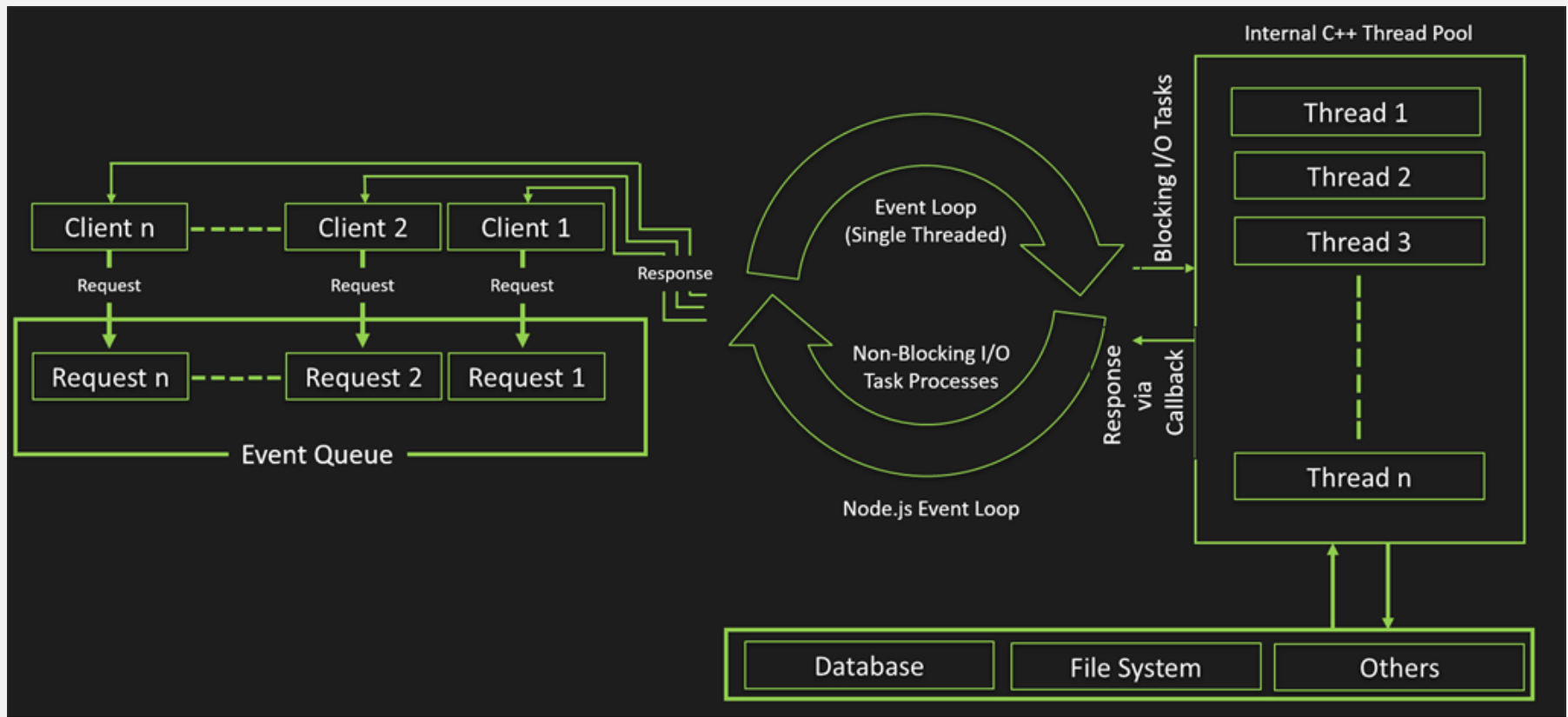
Node.js: introduction



Single-thread programming

(source: <https://strongloop.com/strongblog/node-js-is-faster-than-java/>)

Node.js: introduction

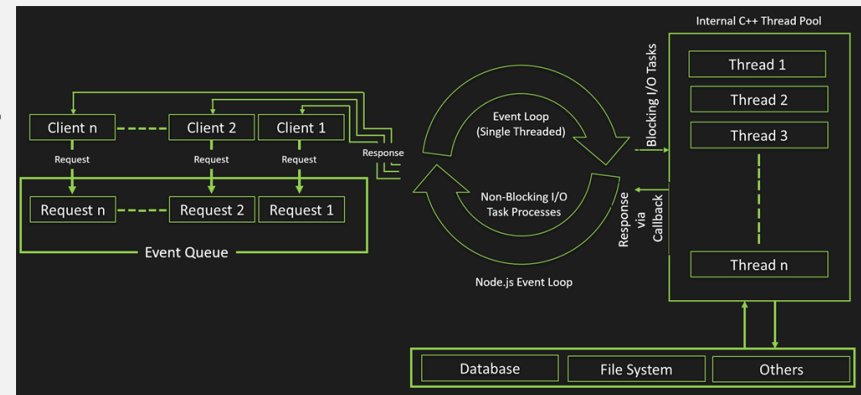


Node.js event processing model

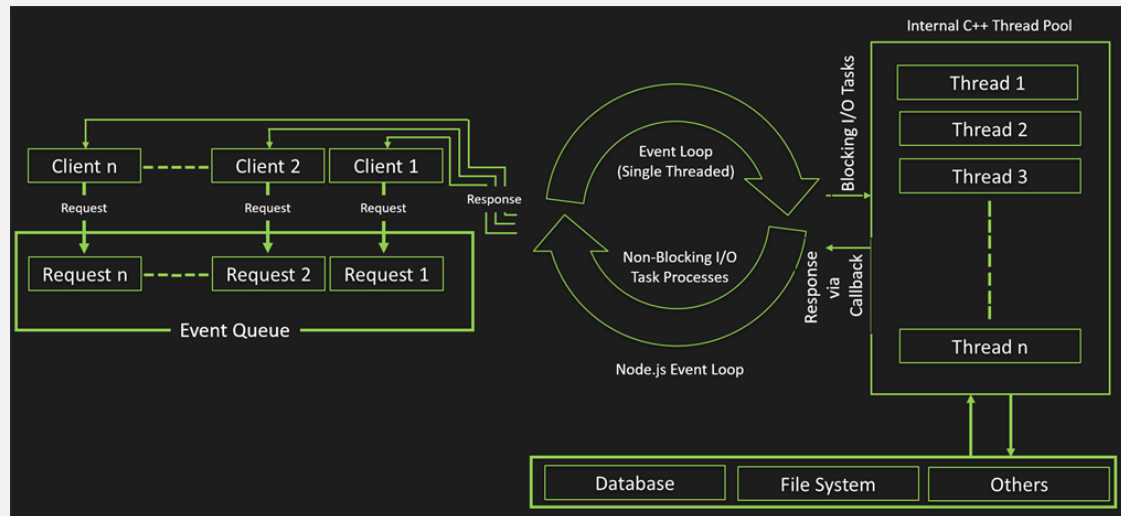
(source: <https://www.c-sharpcorner.com/article/node-js-event-loop/>)

Node.js: execution process

- Whenever a request comes to Node.js API, that incoming request is added to the **event queue**
- The Node.js **event loop** checks if any event or request is available in event queue or not: if any requests are there, then according to the "First Come, First Served" property of queue, the requests will be served
- The event loop is **single threaded** and performs **non blocking I/O** tasks (like database request, file request,...), so it sends requests to **C++ internal thread pool** where lots of threads can be run, therefore handling multiple requests



Node.js: execution process



...

- The event loop checks again and again if any event is there in the event queue: if there is any, then it serves to the thread pool if the blocking process is there
- Whenever any thread completes that task, the callback function calls and sends the response back to the event loop
- Now, event loop sends back the response to the client whose request is completed

Node.js: installation

- Download and run the installation file (.msi for Windows OS)
 - attention, if the PC is old, it may be necessary to install the version for 32-bit systems

The screenshot shows the Node.js Downloads page. At the top, the Node.js logo is centered, with a navigation bar containing links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below the navigation bar, the 'Downloads' section is displayed. It features the text 'Latest LTS Version: 14.16.0 (includes npm 6.14.11)' with a red box around the version number and a red exclamation mark to its right. Below this, a message states: 'Download the Node.js source code or a pre-built installer for your platform, and start developing today.' The main content area is divided into two columns: 'LTS Recommended For Most Users' (dark green) and 'Current Latest Features' (light green). Under the 'LTS' column, there is a 'Windows Installer' button with a Windows logo icon, labeled 'node-v14.16.0-x64.msi'. A large red arrow points to this button. Under the 'Current' column, there are two buttons: 'macOS Installer' with an Apple logo icon, labeled 'node-v14.16.0.pkg', and 'Source Code' with a cube icon, labeled 'node-v14.16.0.tar.gz'. At the bottom, there is a row of three buttons: 'Windows Installer (.msi)', '32-bit', and '64-bit'.

node

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | CERTIFICATION | NEWS

Downloads

Latest LTS Version: 14.16.0 (includes npm 6.14.11) !

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
 Windows Installer node-v14.16.0-x64.msi	 macOS Installer node-v14.16.0.pkg	 Source Code node-v14.16.0.tar.gz

Windows Installer (.msi) 32-bit 64-bit

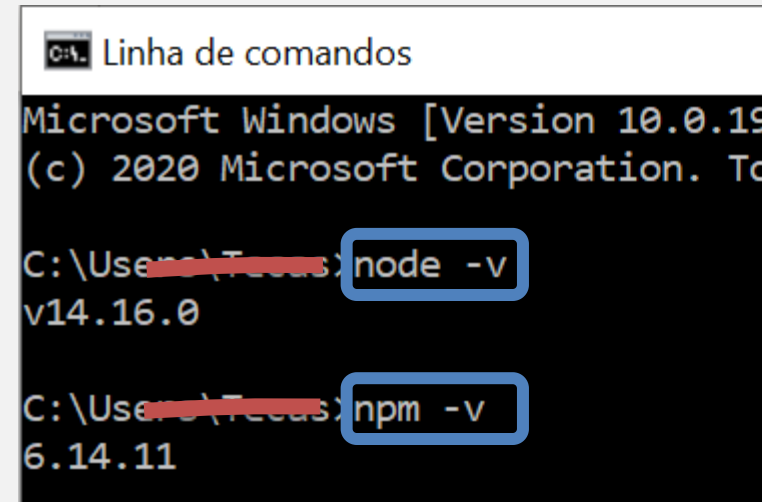
Node.js: installation

- Verify that the installation was successful by obtaining the installed version:

➤ **node -v**

➤ **npm -v**

- If the system does not recognize the commands **node** or **npm**, it is necessary to add the node executable into the environment variables (and then restart the computer)
 - <https://love2dev.com/blog/node-is-not-recognized-as-an-internal-or-external-command/>



```
C:\> Linha de comandos


Microsoft Windows [Version 10.0.19H2.0]
(c) 2020 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Tomas> node -v
v14.16.0

C:\Users\Tomas> npm -v
6.14.11
```

Node.js: hello world

- **First test:** create a Javascript file, add a `console.log` command instruction (or use the `echo` command) and execute it using Node



```
CA. Linha de comandos
C:\Users\...>mkdir nodeFirstTest
C:\Users\...>cd nodeFirstTest
C:\Users\...>echo console.log("Node is installed!"); > HelloNode.js
C:\Users\...>node HelloNode.js
Node is installed!
```

- Attention: commands can differ between operating systems
 - List a directory: `dir` (DOS), `ls` (Unix, MAC)
 - Command help: `help` (DOS), `man` (Unix, MAC)
 - ...

Node.js: hello world

- Any number of **arguments** can be passed when invoking a Node.js application
- Arguments are retrieved by using the **process** object
 - It exposes an **argv** property, which is an array that contains **all the command line invocation arguments**
 - Open the created file with Visual Studio Code and add the following instruction `console.log(process.argv.slice(2));` test it again

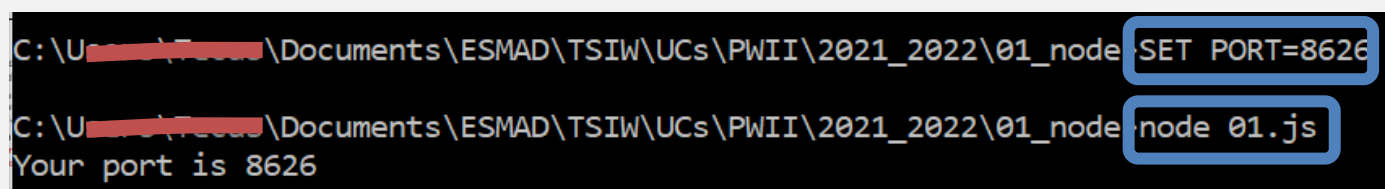
```
C:\Users\...s\Documents\nodeFirstTest>node HelloNode.js
Node is installed!
[]

C:\Users\...s\Documents\nodeFirstTest>node HelloNode.js hello world
Node is installed!
[ 'hello', 'world' ]
```

<https://nodejs.dev/learn/node-is-accept-arguments-from-the-command-line>

Node.js: hello world

- The **process** core module of Node.js provides the **env** property which hosts all the **environment variables**
 - Environment variables are useful when the code changes based on the environment: HTTP port to listen on, path and folder the files are located in, or to point to a development, staging, test, or production stage
- Create a Node program with the following instructions:
`const PORT = process.env.PORT;`
`console.log(`Your port is ${PORT}`);`
- On Windows, run it by typing `SET PORT=8626`, then press `Enter`, then run your program

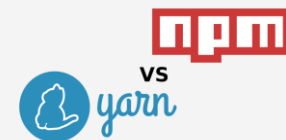


A screenshot of a Windows command prompt window. The first line shows the command `SET PORT=8626` being entered. The second line shows the command `node 01.js` being entered. The output of the program is `Your port is 8626`.

<https://nodejs.dev/learn/how-to-read-environment-variables-from-nodejs>

NPM: *Node Package Manager*

- Node.js module or package manager
- **Modules** are publicly available - reusable components
 - List of available modules: <https://www.npmjs.com/>
 - They can also be installed via the CLI (*Command Line Interface*) installed with Node.js
- It has also become popular as a module manager for other JavaScript frameworks (Vue, jQuery, Gulp...)
- **npm** included in the Node.js installer



[yarn](#) is an alternative to NPM!
Read this [article](#) for more informations.

NPM: main commands

`npm init`

Displays a mini questionnaire to help create a project's `package.json` file

`npm install`

If a project has a `package.json` file, it installs all dependencies in a `node_modules` project subfolder

`npm install module_name`

Installs the module in the project (in the `node_modules` subfolder of the project)

- If the `package.json` file exists, the module name is added within the `dependencies` attribute; it is the same as `npm install module_name --save`
- `--save-dev`: adds the module name within the `devDependencies` attribute
- The difference between `devDependencies` and `dependencies` is that the former should contain just the development tools (like `nodemon` or a testing library) while the latter is bundled with the app dependencies

NPM: main commands

`npm install -g module_name`

Install the module **globally**, at location given by `npm root -g` (won't install the package under the project's local `node_modules` folder)

`npm list`

Lists all modules that have been installed in the project

`npm uninstall module_name`

Uninstall a project module

- `npm uninstall -g module_name`: uninstall a module installed globally
- `npm uninstall module_name --save`: also removes it from the `dependencies` attribute in `package.json`
- `npm uninstall module_name --save-dev`: also removes it from the `devDependencies` attribute in `package.json`

NPM: main commands

`npm update [module_name]`

Updates the version of all modules or the specified module

NOTE: when a module is installed, the latest available version is downloaded and put in the `node_modules` folder, and a corresponding entry is added to the `package.json` and `package-lock.json` files; the goal of `package-lock.json` is to keep track of the exact version of every module that is installed so that a product is 100% reproducible in the same way even if modules are updated by their maintainers

`npm run nome_script`

Executes the command specified in the script in `package.json`

`npm help`

Displays all commands

package.json

- Who have worked with JavaScript front-end frameworks, certainly know what the **package.json** file is
- It is a **module descriptor file**
- In addition to documenting the project and facilitating its sharing, this file indicates its dependencies

Module/project name (mandatory)

Version (mandatory): *major.minor.patch*

Dependencies

```
{
  "name": "meu-primeiro-node-app",
  "description": "Meu primeiro app Node.js",
  "author": "User <user@email.com>",
  "version": "1.2.3",
  "private": true,
  "dependencies": {
    "modulo-1": "1.0.0",
    "modulo-2": "~1.0.0",
    "modulo-3": ">=1.0.0"
  },
  "devDependencies": {
    "modulo-4": "*"
  }
}
```

package.json

- One can automate tasks using: **npm run *command_name***
- New commands must be declared in the **script** attribute of **package.json**

New command: executed using **npm run start**
or **npm start** in the command line

New command: **npm run clean**
(in the example, run a global command that
deletes contents of the *node_modules*
folder)

```
{
  "name": "meu-primeiro-node-app",
  "description": "Meu primeiro app Node.js",
  "author": "User <user@email.com>",
  "version": "1.2.3",
  "private": true,
  "scripts": {
    "start": "node app.js",
    "clean": "rm -rf node_modules",
    "test": "node test.js"
  },
  "dependencies": {
    "modulo-1": "1.0.0",
    "modulo-2": "~1.0.0",
    "modulo-3": ">=1.0.0"
  },
  "devDependencies": {
    "modulo-4": "*"
  }
}
```

package.json

```
C:\Users\Teresa\Documents\nodeFirstTest>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (nodefirsttest) HelloNode
Sorry, name can no longer contain capital letters.
package name: (nodefirsttest) hellonode
version: (1.0.0)
description: Simple Node Web Server
entry point: (HelloNode.js)
test command:
git repository:
keywords: Node Server
author: Teresa Terroso
license: (ISC)
```

package.json

Added later to
automate
application
execution

```
{ package.json > ...
1  {
2    "name": "hellonode",
3    "version": "1.0.0",
4    "description": "Simple Node Web Server",
5    "main": "HelloNode.js",
6    "scripts": {
7      "start": "node HelloNode.js",
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "keywords": [
11     "Node",
12     "Server"
13   ],
14   "author": "Teresa Terroso",
15   "license": "ISC"
16 }
```

```
C:\Users\Teresa\Documents\nodeFirstTest>npm start
```

```
> hellonode@1.0.0 start C:\Users\Teresa\Documents\nodeFirstTest
> node HelloNode.js
```

```
Servidor Node.js a executar em http://127.0.0.1:3000/
```

More about package.json:

<https://docs.npmjs.com/creating-a-package-json-file>

<https://nodejs.dev/learn/the-package-json-guide>

Node.js: modules

- Modern browsers have started to support module functionality natively
- Node.js has had this ability for a long time
- In Node.js a module is loaded into a main application via the statement

```
const library = require(module_name)
```

- The instruction starts searching for the module in the native modules or in the installed modules using the NPM (node_modules folder)
- One can create our own modules, exporting any wanted functions

```
const library = require('./module_name')
```

- the prefix `'/'` indicates the absolute path to the file
- the prefix `'./'` indicates the relative path to the file

Node.js: own modules

- `module.exports` is a special Node.js API for **creating modules**
- When a variable, function, class or object is assigned as a new `exports` property, that is the thing that's being exposed, and as such, it can be imported

File (module) `car.js`

```
const car = {  
  brand: 'Ford',  
  model: 'Fiesta'  
}  
  
module.exports = car
```

File `index.js`

```
const myCar = require('./car.js');  
console.log(typeof myCar) // object  
console.log(`My car is a ${myCar.model}`);  
  
const {model} = require('./car.js');  
console.log(typeof model) // string
```

One can import only a set of exportable features, defining them in a comma-separated list wrapped in curly braces

Node.js: own modules

- Exporting own modules (classes):

File (module) `square.js`

```
// module: as a variable exposes the object it points to
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
  }
};
```

File `bar.js`

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

```
C:\Users\T... \Documents\nodeFirstTest>node bar.js
The area of mySquare is 4
```

Node.js: own modules

- **Named export:** another way to export is, instead of assigning the whole `module.exports` to a value, assign **individual properties** of the default `module.exports` object to values (or simply using `exports`)

Option A) File (module) `circle.js`

```
exports.area = (r) => Math.PI * r ** 2;  
exports.circumference = (r) => 2 * Math.PI * r;
```

One can export only a set of features, defining them in a comma-separated list wrapped in curly braces

Option B) File (module) `circle.js`

```
const area = (r) => Math.PI * r ** 2;  
function circumference(r){  
  return 2 * Math.PI * r;  
}  
module.exports = {  
  area, circumference  
};
```

```
const {area} = require('./circle.js');  
console.log(`The area of a circle with radius 2 is ${area(2)}`);
```

File `index.js`

Node: native modules

Core Module	Description
<code>http</code>	<code>http</code> module includes classes, methods and events to create Node.js http server.
<code>url</code>	<code>url</code> module includes methods for URL resolution and parsing.
<code>querystring</code>	<code>querystring</code> module includes methods to deal with query string.
<code>path</code>	<code>path</code> module includes methods to deal with file paths.
<code>fs</code>	<code>fs</code> module includes classes, methods, and events to work with file I/O.
<code>util</code>	<code>util</code> module includes utility functions useful for programmers.

Node.js: basic web server

- Copy the following code into a HelloNode.js file and run it using Node:

```
// import Node.js core module HTTP
const http = require('http');
const HOST = process.env.HOSTNAME || 'localhost' // local server
const PORT = process.env.PORT || 3000; // determine the port to listen to by checking the
PORT variable first and providing it with a default value, if undefined

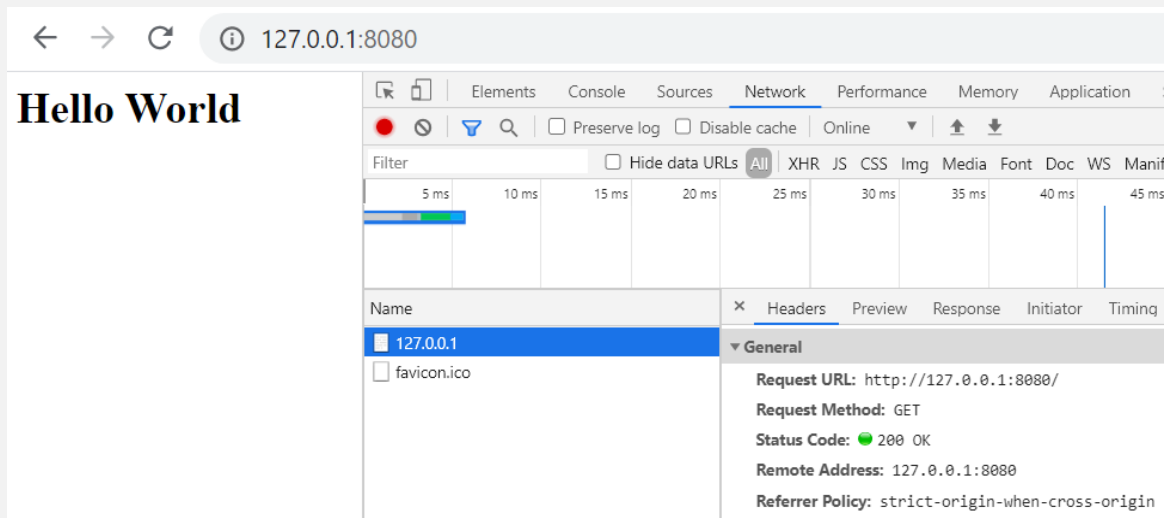
// create a web server instance
const server = http.createServer((request, response) => {
  response.statusCode = 200 // set HTTP status code for OK - SUCCESS
  response.setHeader('Content-Type', 'text/html'); //set HTTP
response header parameters
  response.end("<h1>Hello World</h1>"); // send a response back to client, adding the
content as an argument
});

// listen for any incoming requests
server.listen(PORT, HOST, () => {
  // print message in server terminal
  console.log(`Node server running on http://${HOST}:${PORT}/`);
});
```

Node.js: basic web server

```
C:\Users\T... \Documents\nodeFirstTest>node HelloNode.js
Servidor Node.js a executar em http://127.0.0.1:3000 /
^C
C:\Users\T... \Documents\nodeFirstTest>SET PORT=8080 && node HelloNode.js
Servidor Node.js a executar em http://127.0.0.1:8080 /
^C
```

- This is a classic and simple example of a **web server** running on some defined port, responding in the root route "/" a result in html format with the message **Hello World**
- To end the application, use the command **ctrl+C** on the console



Node: HTTP module

- <https://nodejs.org/api/http.html>
- Module to create a Web server and use the HTTP protocol (to communicate with clients and transfer data on the Web)
- Native module: just invoke it using `require`

```
// Loads native module HTTP of Node.js  
const http = require('http');
```

- The `http` variable is an object, on which we can invoke methods that are in the module

Node: HTTP module

- **`http.createServer([requestHandler(request, response)])`**: creates a web server object - returns an instance of class `http.Server`
 - Turns the computer into na **HTTP server**
 - The function that's passed into **`createServer`** is **called once for every HTTP request** that's made against that server, so it's called the **request handler**
 - When an HTTP request hits the server, Node calls the request handler function with a two handy objects for dealing with the transaction
 - `request`** (instance of the class `http.IncomingMessage`)
 - `response`** (instance of the class `http.ServerResponse`)

```
// create web server
const server = http.createServer((request, response) => {
  // REQUEST HANDLER - handle every single client request with this callback
});
```

Node: HTTP module

- Class **http.Server**: defines methods and events for a Web server
 - **server.listen(...)**: places the server “listening” for client requests through a port or domain of the server
 - In order to actually serve requests, the **listen** method needs to be called on the server object. In most cases, all you'll need to pass to listen is the host domain name and port number you want the server to listen on

```
// create web server
const server = http.createServer((request, response) => {
  ...
});
// listen for any incoming requests
server.listen(port, hostname, () => {
  // print message in server terminal
  console.log(`Node server running on http://${hostname}:${port}/`);
});
```


Node: HTTP module

- Class **http.IncomingMessage**: used to access request status, headers and data
 - when handling a request, the first thing to do is to look at the **method** and **URL**, so that appropriate actions can be taken
 - the **method** will always be a normal HTTP method/verb
 - the **url** is the full URL without the server, protocol or port

```
// create web server
const server = http.createServer((request, response) => {
  const { method, url, headers } = request; // gets the client request properties
  const host = headers['host']; // extracts the host header from request
  console.log(method, url, host);

  if (request.url === '/node') { // reads the HTTP request URL
    response.write('<h1>Hello, Node.js!</h1>');
  }
  response.end();
});
```

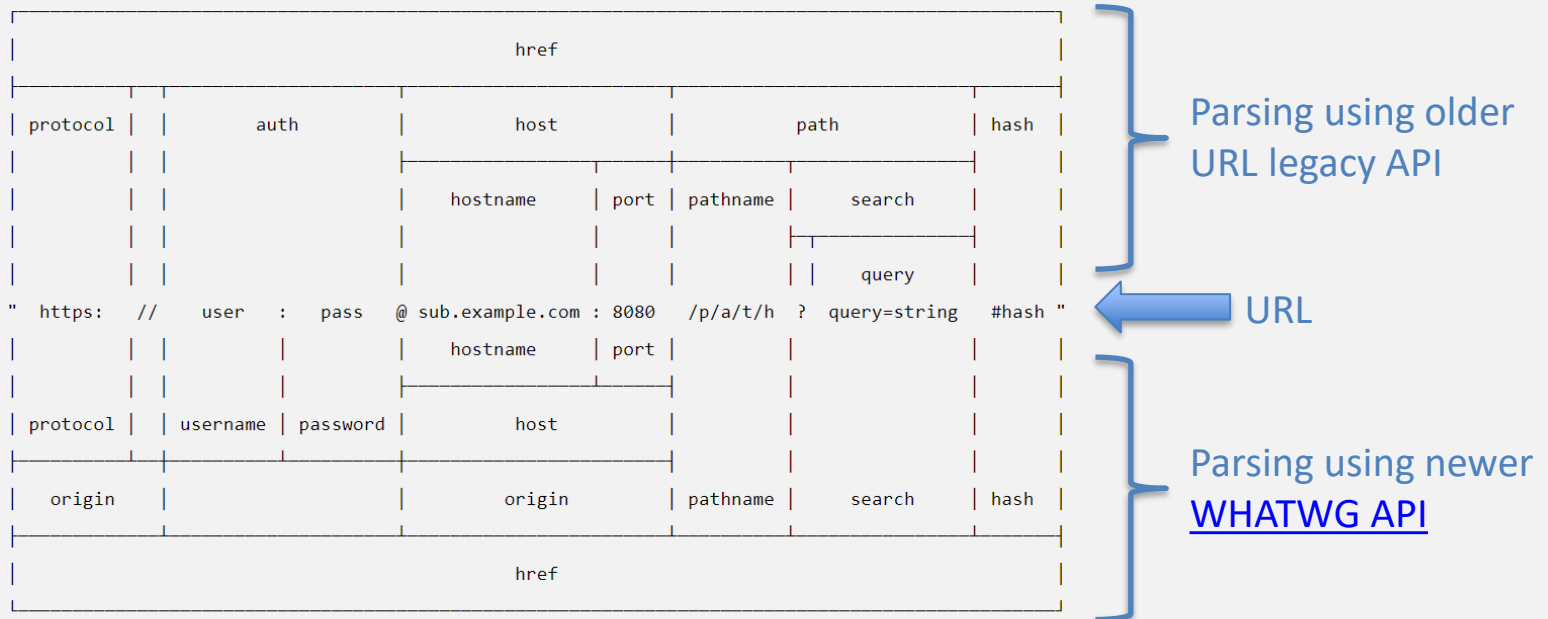
Node: HTTP module

- Class **http.serverResponse**: defines the HTTP response from the server to the client
 - **response.setHeader(name, value)**: sets the headers
 - **response.write(chunk[, encoding][, callback])**: send parts or the entire body of the HTTP response
 - **response.end([data][, encoding][, callback])**: mandatory for each answer; tells the server to send the response

```
// create web server
const server = http.createServer((request, response) => {
  // set response header
  response.setHeader('Content-Type', 'text/html');
  // set response content
  response.write("<html><body><h1>Hello World</h1></body></html>");
  // sends response to the client
  response.end();
});
```

Node: HTTP module

- Reading the request URL is performed through the **request.url** parameter of the server
 - HTTP protocol URL addresses can have, among others, patterns like **query strings** (*?name=john*) or **pathnames** (*/dir/subdir*)




(All spaces in the "" line should be ignored. They are purely for formatting.)

Node: URL module

- How to parse query strings from request's URLs?

```
...  
const url = require('url'); //import URL module  
...  
const server = http.createServer(function (req, res) {  
  //consider a request with a query ?file=string  
  const parse_q = url.parse(req.url, true);  
  console.log(parse_q.query.file); //outputs 'string'  
  ...  
});
```

```
...  
const server = http.createServer(function (req, res) {  
  //consider a request with a query ?file=string  
  const parse_q = new URL(req.url, `http://${req.headers.host}`);  
  console.log(parse_q.searchParams.get('file')); //outputs 'string'  
  ...  
});
```



Node: nodemon module

- Let's change the server response, depending on the URL of the request
- **nodemon** is used to allow changing the code without finishing and restarting the server
 - Installation: `npm install -g nodemon`
 - Test if it was well installed: `nodemon -v`
 - Executing a server using `nodemon [server_filename]` instead of `node [server_filename]` allows it to be automatically restarted whenever changes are made to the node application

Node: HTTP module

- Let's change the server response, depending on the URL of the request:

```
...
const server = http.createServer(function (req, res) {
  //check the URL of the current request
  if (req.url == '/') {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write("<html><body><h1>This is home Page.</h1></body></html>");
    res.end();
  }
  else if (req.url == "/student") {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write('<html><body><h2>This is student Page.</h2></body></html>');
    res.end();
  }
  ...
  else
    res.end('PAGE NOT FOUND!');
});
// listen for any incoming requests
server.listen(port, hostname, () => {
  console.log(`Node server running on http://${hostname}:${port}/`);
});
```

Node: HTTP module

- Let's change the server response, depending on the URL of the request:

← → ↻ ⓘ 127.0.0.1:3000

This is home Page.

← → ↻ ⓘ 127.0.0.1:3000/student

This is student Page.

← → ↻ ⓘ 127.0.0.1:3000/admin

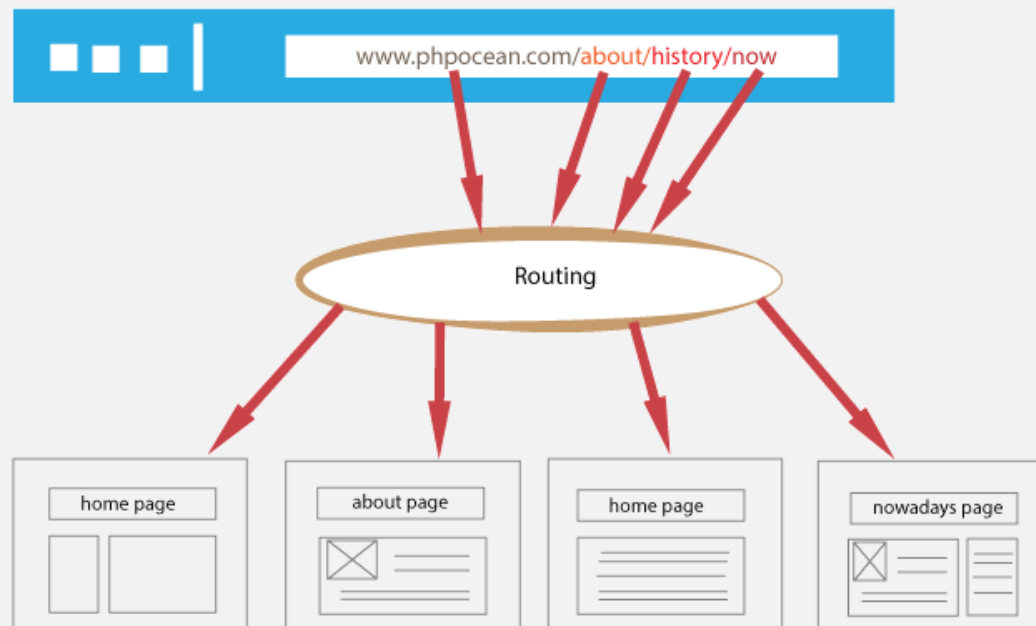
This is admin Page.

← → ↻ ⓘ 127.0.0.1:3000/anyOtherURL

PAGE NOT FOUND!

Node: File System (FS) module

- Typically, web servers provide the HTML pages indicated in the request URL - **routing**:
 - *hostname/index.html*: HTTP GET request of page *index.html*
 - *hostname/dir/file.html*: HTTP GET request of page *file.html* from directory *dir*



Node: FS module

- In Node.js, the **native module FS** ([File System](#)) is used to manage files and directories
 - **`fs.open(path, flags[, mode], callback)`**: opens a file (for reading / writing) in asynchronous mode
 - **`fs.writeFile(filename, data[, options], callback)`**: write to a file (if it already exists, perform an overwrite) in asynchronous mode
 - **`fs.readFile(filename[, options], callback)`**: read file asynchronously
 - **`fs.readdir(path[, options], callback)`**: reads a directory in asynchronous mode, returning an array with the names of the files
 - ...

Node: FS module

- **Example:** display multiple (static) HTML pages by analyzing the URL
 - URL must contain the relative path for the HTML file to be displayed
- The **url** property of the IncomingMessage class does not contain the origin part: only the pathname, search queries and hash
 - The remaining information can be retrieved from the HTTP request headers

If user types the following URL in a browser:

<http://localhost:3000/status?name=ryan>

// read some properties from the Incoming Message class

```
const server = http.createServer(function (req, res) {  
  console.log(req.method);           // 'GET'  
  console.log(req.url);              // 'status?name=ryan'  
  console.log(req.headers.host);     // 'localhost:3000'  
})
```

href									
protocol		auth		host		path		hash	
				hostname		port	pathname	search	
								query	
user:		//		user		:	pass	@ sub.example.com : 8080 /p/a/t/h ? query=string #hash "	
ryan				hostname		port			
protocol		username		password		host			
						origin		pathname	
								search	
								hash	
GET									
'status?name=ryan'									
localhost:3000'									
req.url									
href									

(All spaces in the "" line should be ignored. They are purely for formatting.)

In this example the URL pathname must provide the file path of the HTML page. Therefore, one must only add a “.” to transform it into a file relative path. But remember client can add more to the URL (queries,...)

Node: FS module

The image displays three browser screenshots illustrating file system operations using the Node.js FS module. Each screenshot shows a browser window with a specific URL and a corresponding page content or error message. The browser's developer tools are open, showing the network tab with a table of requests.

Top Left Screenshot: The browser address bar shows `127.0.0.1:3000/index.html`. The page content is **This is home Page.** The network tab shows a single request with a status of 200.

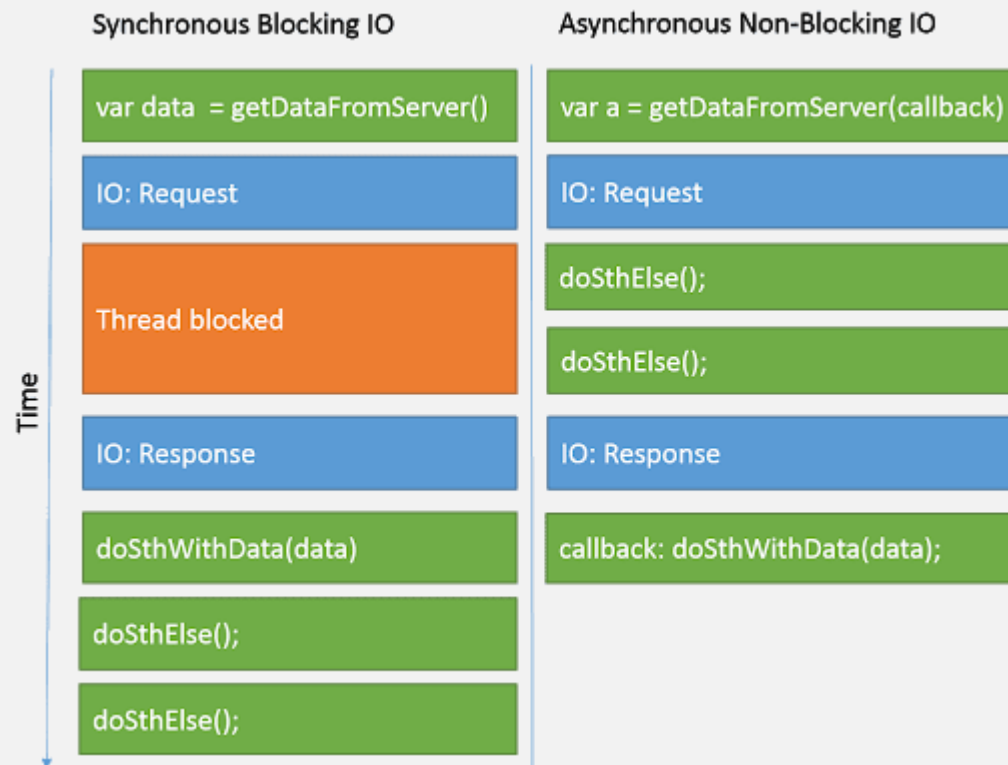
Top Right Screenshot: The browser address bar shows `127.0.0.1:3000/html/admin.html`. The page content is **This is admin Page.** The network tab shows a single request with a status of 200.

Bottom Screenshot: The browser address bar shows `127.0.0.1:3000/someOtherPage.html`. The page content is **404 Not Found**. The network tab shows a single request with a status of 404.

Name	Status
admin.html	200
someOtherP...	404

Node: FS module

- The FS module provides synchronous and asynchronous file manipulation methods



Node: FS module

- Synchronous reading:
 - Create a *readme.txt* text file with the content *"Please read me many times!"*
 - Read it 3 times synchronously
 - Force an error: try to read a file that does not exist

```
const fs = require('fs');  
//forcing error  
try {  
    let data = fs.readFileSync('./noSuchFile.txt');  
    console.log('0', data.toString());  
} catch (err) {  
    console.log(err.message);  
}  
//1st time  
try {  
    let data = fs.readFileSync('./readme.txt');  
    console.log('1', data.toString());  
} catch (err) {  
    console.log(err.message);  
}  
//read file two more times  
console.log('end');
```

```
start  
ENOENT: no such file or directory, open './noSuchFile.txt'  
1 Please read me many times!  
2 Please read me many times!  
3 Please read me many times!  
end
```

Node: FS module

- Asynchronous reading:
 - Read the file again 3 times, but now asynchronously

```
const fs = require('fs');
console.log('start');
//1st time
fs.readFile('./readme.txt', (err, data) => {
  if (err) { throw err; }
  console.log('first', data.toString());
});
//2nd time
fs.readFile('./readme.txt', (err, data) => {
  if (err) { throw err; }
  console.log('second', data.toString());
});
//3rd time
fs.readFile('./readme.txt', (err, data) => {
  if (err) { throw err; }
  console.log('third', data.toString());
});
console.log('end');
```

```
C:\Users\T... \Documents\nodeFirstTest\fs>node asyncFS.js
start
end
first Please read me many times!
third Please read me many times!
second Please read me many times!
C:\Users\T... \Documents\nodeFirstTest\fs>node asyncFS.js
start
end
first Please read me many times!
second Please read me many times!
third Please read me many times!
```

Node: FS module

- Asynchronous reading, but with orderly console return
 - sometimes it is important that there is some sort of ordering in I/O functions

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

console.log('start');
fs.readFile('./readme.txt', (err,data) => {
  if (err) { throw err; }
  console.log('first', data.toString());
  fs.readFile('./readme.txt', (err,data) => {
    if (err) { throw err; }
    console.log('second', data.toString());
    fs.readFile('./readme.txt', (err,data) => {
      if (err) { throw err; }
      console.log('third', data.toString());
    })
  })
})
console.log('end');
```

```
start
end
first Please read me many times!
second Please read me many times!
third Please read me many times!
```


Node: FS module

- Asynchronous reading, but with orderly console return: *callback hell*

➤ Solution: *promises* or *async/await*

<https://nodejs.dev/learn/understanding-javascript-promises>

<https://nodejs.dev/learn/modern-asynchronous-javascript-with-async-and-await>



```
1 function hell(win) {
2   // for listener purpose
3   return function() {
4     loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5       loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6         loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7           loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8             loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9               loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                    loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                      async.eachSeries(SERIALS, function(src, callback) {
14                        loadScript(win, BASE_URL+src, callback);
15                      });
16                    });
17                  });
18                });
19              });
20            });
21          });
22        });
23      });
24    });
25  };
26 }
```

Node: FS module

- From Node version 10, functions of **fs** module can return promises directly, due to its **fs/promises** API.

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

async function readThreeTimes() {
  const data1 = await fs.readFile('./readme.txt');
  console.log("1" + " " + data1.toString());

  const data2 = await fs.readFile('./readme.txt');
  console.log("2" + " " + data2.toString());

  const data3 = await fs.readFile('./readme.txt');
  console.log("3" + " " + data3.toString());
}

readThreeTimes();
```

"**await** can be put in front of any **async** promise-based function to pause the code on that line until the promise fulfills, then return the resulting value"

SOURCE:

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await