

# THIS IS A **WEB DEVELOPMENT** COURSE

PROF. SANTORO FEDERICO FAUSTO

UNIVERSITY OF CATANIA  
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE



# HELLO!

## I am Santoro Federico Fausto

I am here because I love divulging what I learnt during my working experience.  
You can find me at [@fedyfausto](https://twitter.com/fedyfausto)

# INSTRUCTIONS FOR USE

## GOOGLE IS YOUR FRIEND

Before asking for questions, it's a good habit to look for a solution with your search engine (I always do that while looking for the best solution to solve a problem).

## STACKOVERFLOW IS NOT ALWAYS THE BEST SOLUTION

Please do not choose the first answer that you find, but try to understand why the solution works.

## WORK TOGETHER

Cooperation is vital in this work, it could save you a lot of time sometimes.

## PRACTICE, PRACTICE AND PRACTICE

It is a good practice to learn how to use the introduced technologies and to continue practicing outside the course.

## DO NOT BE AFRAID TO ASK FOR HELP

# COURSE PROGRAM (FRONT-END)

## Network Application Layer

- HTTP Introduction
- Client - Server
- URLs and Verbs
- Status Codes

## HTML Introduction

- Web Browsers and HTML Documents
- Block and inline Elements
- Text Elements
- Attributes
- List and Tables
- Forms

## CSS Introduction

- Simple Selectors
- Selector Combinators
- Pseudo-classes and elements
- Box Model
- CSS Style rules
- Cols System
- Flexbox System
- Grid System

## Javascript Introduction

- Variables
- Data Types
- Expressions
- Statements
- Conditionals
- Arrays
- Functions
- Arrow Functions
- Scope and Closures
- Objects and Notations
- Classes
- DOM
- Events
- Asynchronous
- Callbacks
- Promises
- Async/Await
- Ajax and Fetch

## JQuery Introduction

- Selectors
- Elements
- DOM Traversing

## TypeScript Introduction

- Differences and Story
- The compiler
- Workflow and Configuration
- Objects and Arrays
- Explicit and Dynamic Types
- Functions and Aliases
- DOM and Casting
- Classes
- Modules
- Interfaces
- Generics, Enums and Tuples

## UI / UX For Developers (by Prof. Mulà D.)

- Terminology
- Wireframe, Mockup and Prototype
- Basics: Typography, Colors and Images
- User Centered Design (UCD)
- User Experience Design
- Common use and Dark Patterns
- Usability and Accessibility
- Introduction to Figma
- Figma handoff

# COURSE PROGRAM (BACK-END)

## Introduction to Back-End Development

- HTTP & Client - Server Re-Introduction
- Server and Services
- SOA and Microservices
- Workflow of a Server
- Database Types
- Back-end Languages
- Back-end Technologies
- Back-end Systems
- SOAP Protocol
- REST Architecture
- XML and JSON
- Security Aspects

## Introduction to NodeJS

- History of NodeJS
- Installation
- Difference between Browser and NodeJS
- The V8 Engine and other engines
- Hello World!
- Environment Variables
- Application Arguments
- Exports
- The NPM
- The package.json
- Dependencies
- The NPX runner
- The Event Loop
- Event Emitter

## NodeJS - Development Introduction

- Simple HTTP Server and request
- Files operations
- Buffers
- Streams
- Error handling

## NodeJS - Express Introduction

- Hello Express
- Route Handlers
- Middlewares
- Static Files
- Error Handling
- Render Views
- The Generator
- Controllers
- Forms

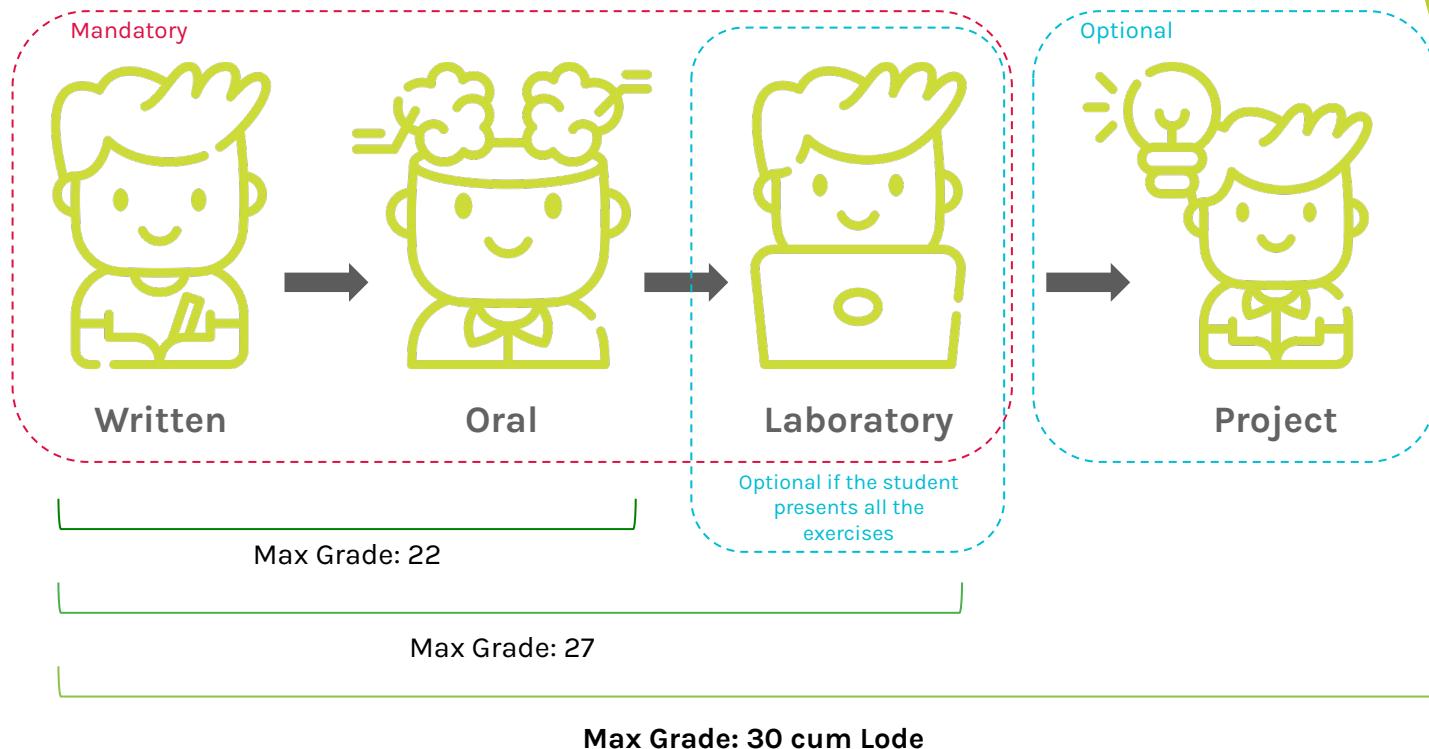
## NodeJS - WebSockets Introduction

- HTTP Polling and Streaming
- Handshake
- Messaging Transmission

## Node JS - SocketIO Introduction

- Difference between WebSockets and SocketIO
- Simple Server Example
- Simple Client Example
- Events and Rooms

# COURSE EXAM



# 1.

## INTRODUCTION

[HTTP://START.HERE](http://start.here)

# WHAT IS HTTP ?

HTTP was developed by Tim Berners-Lee and his team in 1989, HTTP has gone through many changes that have helped maintain its simplicity while shaping its flexibility.

HTTP is a stateless, application-layer protocol to let communication possible between distributed systems, and is the foundation of the modern web.

# WHAT IS HTTP ?

HTTP allows communication between a variety of hosts and clients, and supports a mixture of network configurations.

HTTP evolved from a protocol designed to exchange files in a semi-trusted laboratory environment into a modern internet maze that carries images and videos in high resolution and 3D.

# HOW DOES IT WORK ?



Communication between a host and a client occurs, via a request/response pair.

The client initiates an **HTTP request message**, which is serviced through a **HTTP response message** in return.

# THE URLs

The heart of web communications is the request message, which is sent via Uniform Resource Locators (**URLs**).

[PROTOCOL]://[DOMAIN]:[PORT]/[RESOURCE]?[QUERY][#FRAGMENT]

E.G.

HTTP://PICSUM.PHOTOS/200/300/?RANDOM

DEFAULT PORT: 80 (HTTP) OR 443 (HTTPS)

# THE VERBs

URLs reveal the identity of a particular host with whom we want to communicate, but the action that should be performed on the host is specified via HTTP verbs (or methods).

- ▶ **GET**: fetch an existing resource.
- ▶ **POST**: create a new resource.
- ▶ **PUT**: update an existing resource.
- ▶ **DELETE**: delete an existing resource.

PUT and DELETE are sometimes considered specialized versions of the **POST** verb

# THE STATUS CODES

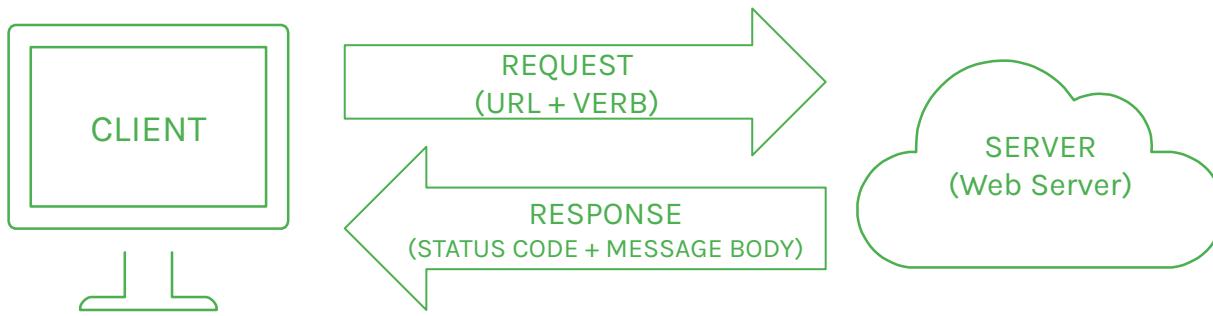
With URLs and verbs, the client can initiate requests to the server.

In return, the server responds with status codes and message payloads.

- ▶ **1XX**: Info messages
- ▶ **2XX**: Successful messages
- ▶ **3XX**: Cache messages
- ▶ **4XX**: Client error messages
- ▶ **5XX**: Server error messages

Never seen the error **404**? Go to <http://alexgorbatchev.com/wiki/>

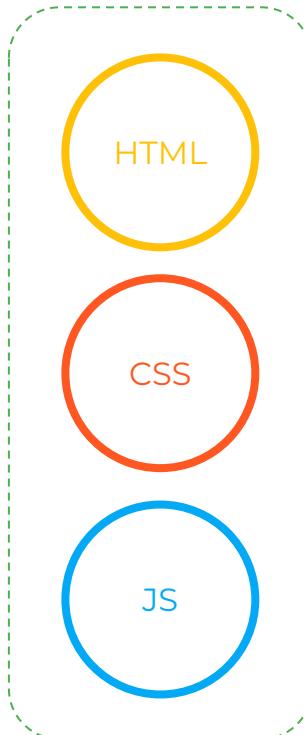
# REQUEST AND RESPONSE FORMATS



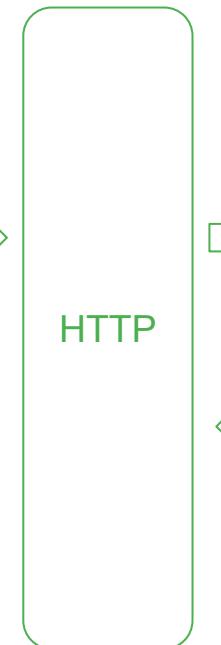
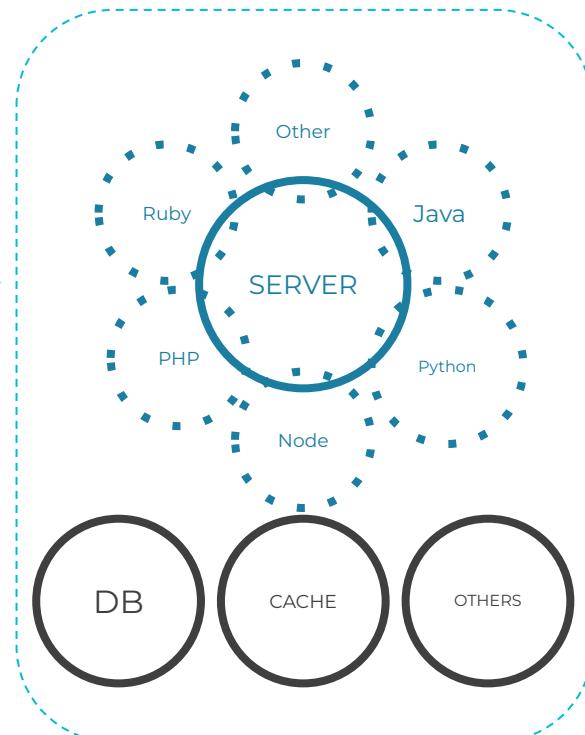
We have seen that URLs, verbs and status codes make up the fundamental pieces of an HTTP request/response pair.

# THE FULL STACK

Frontend Stack



Backend Stack



# 2.

# HTML

```
<p>  
The Markup  
language  
</p>
```

# WHAT IS HTML ?

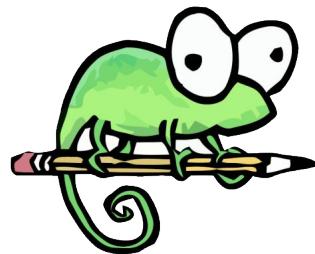
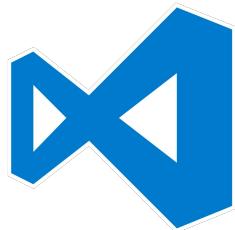
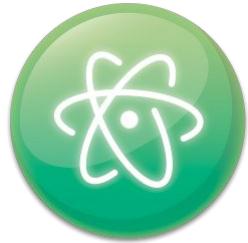
HTML is the standard markup language to create Web pages.

It **is not** a programming language!

It does not have the proper constructs of programming, such as the "conditional" mechanisms.

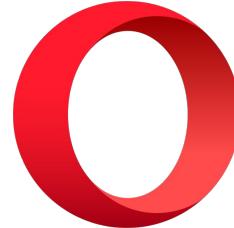
It describes **the structure** of Web pages with elements represented by <tags>.

# HTML EDITORS



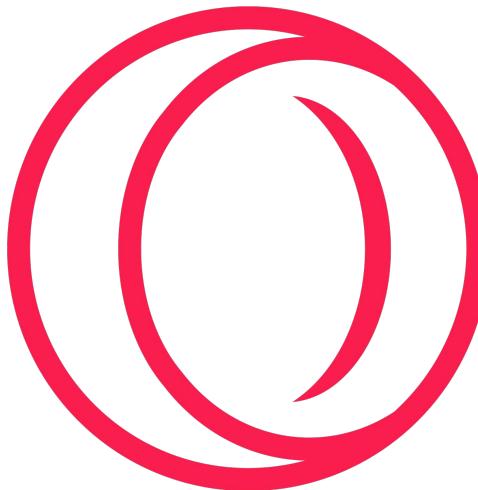
# WEB BROWSERS

The purpose of a **web browser** is to read HTML documents and display them.



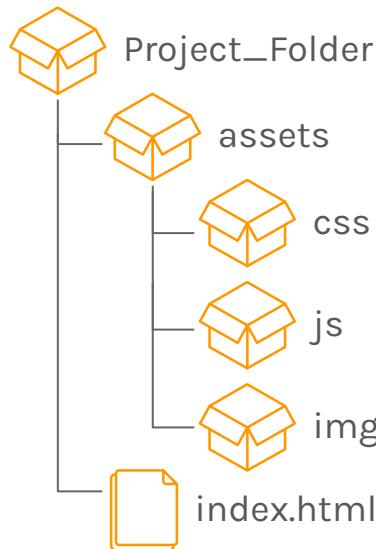
# WEB BROWSERs (in 2022)

The purpose of a **web browser** is to read HTML documents and display them.



# PROJECT FOLDER

An HTML **project folder** has this structure



The name of **index.html** file means it will be loaded as the main file.

This is a web server configuration that can be customized,

# THE DOCTYPE

The `<!DOCTYPE>` declaration represents the document type, help browsers displaying web pages correctly (like the magic number).

In HTML 4.01

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
```

In HTML 5

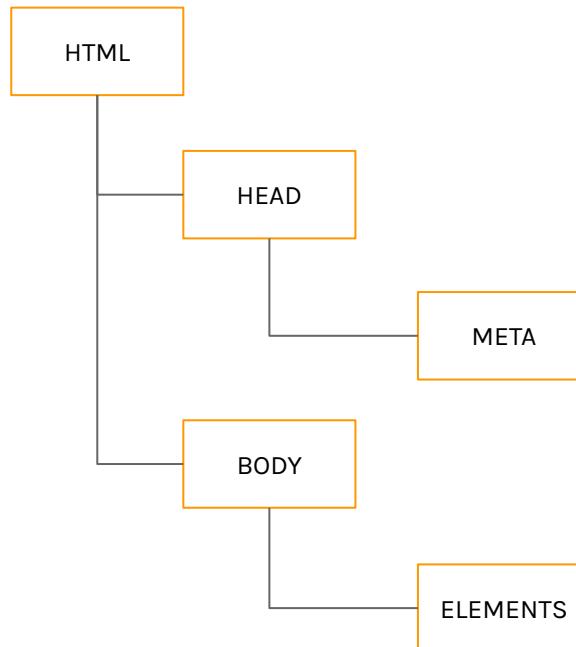
```
<!DOCTYPE html>
```

# HTML PAGE STRUCTURE

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.      <title>Title here!</title>
5.      <!-- META INFO -->
6.  </head>
7.
8.  <body>
9.
10.     <!-- ELEMENTS HERE -->
11.
12.  </body>
13. </html>
```

# HTML TREE

An **HTML structure** is defined by a n-ary tree.



All nodes of tree are  
HTML elements.

# HTML ELEMENTS

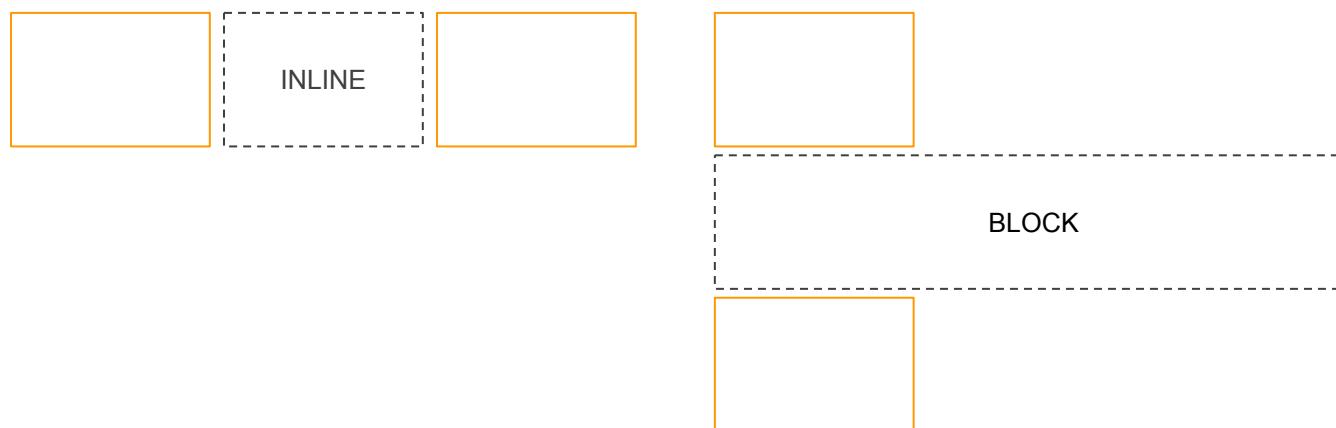
An **HTML element** is defined by a starting `<tag>`.

If the element contains other content, it ends with a closing tag, where the element name is preceded by a forward slash.

1. `<p>`
2.     This is paragraph content.
3. `</p>`

# BLOCK AND INLINE ELEMENTS

An **HTML element** may have a block or inline behavior



# SOME INLINE ELEMENTS

1. `<span>some text</span>`
2. `<b>Bold text</b>`
3. `<small>Small text</small>`
4. `<a>Link</a>`
5. `<i>Italic!</i>`

E.g.

1. `<body>`
2.   `<span> Hello <b><i>World!</i></b> </span>`
3. `</body>`

# SOME BLOCK ELEMENTS

1. `<p>some text</p>`
2. `<div>Container</div>`
3. `<h6>Small text</h6>`
4. `<h2>Some text</h2>`
5. `<ul>Italic!</ul>`

E.g.

1. `<body>`
2.   `<p> Hello </p>`
3.   `<p> <b><i>World!</i></b> </p>`
4. `</body>`

# DISPLAY TEXT

1. some text
2. `<p> some text </p>`
3. `<span> some text </span>`
4. `<b> bold text </b>`
5. `<i> italic text </i>`
6. `<small> small text </small>`
7. `<big> big text </big>`
8. `<h1> h1 text </h1>`
9. `<h2> h2 text</h2>`
10. `<h3> h3 text </h3>`
11. `<h4> h4 text </h4>`
12. `<h5> h5 text </h5>`
13. `<h6> h6 text </h6>`
14. `<center> centered text </center>`



**EXCERCISE  
TIME!**

**HELLO  
WORLD!**

# EXERCISE TIME

1. Create your own project folder
2. Create your first HTML file!
3. Create the main structure
4. Assign a title
5. Show a simple “Hello World”

# ELEMENT ATTRIBUTES

An attribute is used to define the characteristics of an HTML element and is placed inside the opening tag of the element.

All attributes are made up of two parts:

**KEY=**"VALUE"

It's like a normal object var.

# NATIVE ATTRIBUTES

There are four attributes that can be used in all HTML elements .

Other elements have their own attributes.

- ▶ **ID**

Unique HTML element identifier

- ▶ **CLASS**

Group HTML element identifier

- ▶ **STYLE**

Define some style inline

- ▶ **TITLE**

Description, this will display in a tooltip

# VOID ELEMENTS

There are some HTML elements which don't need to be closed.

1. `<br />`
2. `<hr />`
3. `<img />`
4. `<input />`

# HTML COMMENTS

Comment is a piece of code which is ignored by any web browser.

It is a good practice to add comments into your HTML code, especially in complex documents, to indicate sections of a document, and any other notes to anyone looking at the code.

```
<!-- COMMENT HERE -->
```

# HTML HYPERLINKS

Hyperlinks allow visitors to navigate between Web sites by clicking on words, phrases, and images.

It is inline element.

```
<a [attributes]>....</a>
```

Attributes:

- ▶ **Href:**

Define the URL of the document

- ▶ **Target:**

This attribute is used to specify the location where linked document is opened.

```
<a href="http://google.it" target="_blank">Click</a>
```

# HTML ANCHORS

You can create a link to a particular section of a given webpage by using ID or NAME attribute.

```
<a href="#test">Click here</a>
```

...

...

...

```
<p id="test">Hello!</p>
```

# HTML IMAGES

In HTML, images are defined with the `<img>` tag.

The `<img>` tag is void, IT only contains attributes and does not have a closing tag.

`<img [ATTRIBUTES] />`

Attributes:

- ▶ **Src:**  
Define the URL of the image (PNG,JPG,GIF, SVG) or Base64 String
- ▶ **Alt:**  
The alt attribute provides an alternate text for an image
- ▶ **Width and Height**  
Specify the width and height of an image.



**EXCERCISE  
TIME!**

**LINKS!**

# EXERCISE TIME

1. Create two <p> with different ID
2. Create a Link to google
3. Create a Link to google and initialize the search parameter with a personal word
4. Create a Link to another page of your project
5. Create a Link with a image
6. Create a Link with a image and assign a tooltip
7. Try to make some links like a horizontal menu

# HTML LISTS

Lists allow you to list various items. These are natively block elements.

There are several types of lists, the most used are the ordered (`<ol>`) and unordered (`<ul>`) lists.

Any element in the list is a block element `<li>`.

1. `<ol>`

2. `<li>First</li>`

3. `<li>Second</li>`

4. `<li>Third</li>`

5. `</ol>`

1. `<ul>`

2. `<li>Coffee</li>`

3. `<li>Sugar</li>`

4. `<li>Milk</li>`

5. `</ul>`

# HTML TABLES

The HTML tables were designed to represent data, later used to manage the web page structure (no longer used).

```
1.  <table>
2.    <tr>
3.      <th>Username</th>
4.      <th>Points</th>
5.    </tr>
6.    <tr>
7.      <td>Fedyfausto</td>
8.      <td>27960</td>
9.    </tr>
10.   ...
11. </table>
```

Username	Points
Fedyfausto	27960

You can use the “border” in the table tag attribute to display in right way your table.

<tr> : Table Row

<th> : Table Header

<td> : Table Data

# HTML TABLES - COLSPAN

To make a cell span more than one column, use the colspan attribute

```
1.  <table>
2.    <tr>
3.      <th>Username</th>
4.      <th colspan="2">
5.        Points
6.      </th>
7.    </tr>
8.    <tr>
9.      <td>Fedyfausto</td>
10.     <td>Max: 27960</td>
11.     <td>Min: 5463</td>
12.   </tr>
13.   ...
14. </table>
```

Username	Points	
Fedyfausto	Max: 27960	Min: 5463

# HTML TABLES - ROWSPAN

To make a cell span more than one row, use the rowspan attribute

```
1.  <table>
2.    <tr>
3.      <th>Username:</th>
4.      <td>Fedyfausto</td>
5.    </tr>
6.    <tr>
7.      <th rowspan="2">
8.          Points:
9.      </th>
10.     <td>Max: 27960</td>
11.   </tr>
12.   <tr>
13.     <td>Min: 5463</td>
14.   </tr>
15. </table>
```

<b>Username:</b>	Fedyfausto
<b>Points:</b>	Max: 27960
	Min: 5463



**EXCERCISE  
TIME!**

**TABLES!**

# EXERCISE TIME

Create a HTML page with this tables

Name:	Mario Rossi
Email:	mario@telecom.com
	mrossi@gmail.com

Name:	Luigi Mario
Email:	luigi@nintendo.it
	ml@nintendo.com

# EXERCISE TIME

Create a HTML page with this tables

Title		
A	C	E
B	D	F

Title	A	D
	B	E
	C	F

# EXERCISE TIME

Create a HTML page with this tables

Title			A	B
	D	E	F	G
C	H	I		J
	K	L	M	
	N		O	

# HTML DIVisor

The div tag defines a division or a section in an HTML document. It is a block element

1. <**div**>
2. <!-- elements -->
3. </**div**>

The <div> element is very often used together with CSS, to layout a web page (instead tables).

# HTML FORMS

The form element defines a form that is used to collect user inputs.

1. `<form [ATTRIBUTES]>`
2.     `<!-- form elements -->`
3. `</form>`

Form elements are different types of input elements, like text fields, checkboxes, radio buttons, submit buttons, and more.

# HTML FORMS - ATTRIBUTES

The action attribute defines the action to be performed when the form is submitted.

Normally, the form data is sent to a web page on the server when the user clicks the submit button.

1. `<form action="http://google.it/search">`
2.       `<!-- form elements -->`
3. `</form>`

# HTML FORMS - ATTRIBUTES

The method attribute specifies the HTTP method (GET or POST) to be used when submitting form datas

1. `<form method="get or post">`
2.     `<!-- form elements -->`
3. `</form>`

# HTML FORMS - GET METHOD

The default method when submitting form datas is GET. However, when GET is used, the submitted form datas will be visible in the address field of the page (and it is limited at 255 chars).

1. `<form action="http://google.it/search" method="get">`
2.       `<!-- form elements -->`
3. `</form>`

GET must NOT be used to send sensitive information!

# HTML FORMS - POST METHOD

Always use POST if form datas contain sensitive or personal informations. POST method does not display the submitted form datas in page address field.

1. `<form action="https://it.search.yahoo.com/search" method="post">`
2. `<!-- form elements -->`
3. `</form>`

POST has no size limitations, and can be used to send large amounts of data.

# HTML INPUTS

The <input> element is the most important form element. The <input> element can be displayed in several ways, depending on the type attribute.

1. `<input type="{TYPES}" name="{NAME OF KEY}" [OTHER ATTRIBUTES] />`

Each input field must have a name attribute to be submitted and it defines the key name (or variable).

If name attribute is omitted, the data of that input field will not be sent at all.

# HTML INPUTS - TEXT TYPES EXAMPLE

```
<input type="text" name="nickname" />
```

This defines a one-line input field for a text input.

```
<input type="password" name="password" />
```

This defines a one-line input field for a text input (hidden).

```
<input type="email" name="personal_email" />
```

This defines a one-line input field for email input (control forced by browser).

```
<input type="number" name="age" />
```

This defines a one-line input field for numeric input (control forced by browser).

# HTML TEXTAREA

This represents a multi-line plain-text editing control.

1. `<textarea name="textarea" rows="10" cols="50">`
2.       Write something here
3. `</textarea>`

All input attributes work with textareas.

The **rows** attribute defines the number of visible text lines for the control.

The **cols** attribute defines the visible width of the text control, in average character width.

# HTML INPUTS - ATTRIBUTES

- ▶ **Value**  
Specifies the initial value for an input field.
- ▶  **Readonly** (void attribute)  
Specifies that the input field is read only (cannot be changed).
- ▶  **Disabled** (void attribute)  
specifies that the input field is disabled. A disabled input field is unusable and un-clickable, and its value will not be sent when submitting the form.
- ▶ **maxlength**  
specifies the maximum allowed length for the input field
- ▶ **placeholder**  
specifies a hint that describes the expected value of an input field
- ▶ **Min and Max**  
specify the minimum and maximum values (number type)

# HTML INPUTS - CHECKBOX

These are rendered by default as square boxes that are checked (ticked) when active, like you might see in an official government paper form. They allow you to select single values for submission in a form (or not).

1. Which of these game consoles do you own? `<br \>`
2. `<input type="checkbox" name="console" value="nintendo" checked>` The one`<br \>`
3. `<input type="checkbox" name="console" value="sony">` 30 fps console`<br \>`
4. `<input type="checkbox" name="console" value="microsoft">` red ring

# HTML INPUTS - RADIOS

These are rendered by default as circle buttons that are checked (ticked) when active, like you might see in an official government paper form. They allow you to select only one value for submission in a form with which same name attribute.

1. Which of these game consoles do you own?   
 The one  
 30 fps console  
 red ring

# HTML SELECT

The select element represents a control that provides a menu of options.

1. <select name="select">
2. <option value="value1">Value 1</option>
3. <option value="value2" selected>Value 2</option>
4. <option value="value3">Value 3</option>
5. </select>

The **multiple** special void attribute indicates that multiple options can be selected in the list, like a group of checkboxes.

# HTML BUTTON

The button tag defines a clickable button.

Inside a button element you can put content, like text or images. This is the difference between this element and other buttons created with the `<input>` element.

```
<button [ATTRIBUTES] >Click me [OR OTHER HTML]</button>
```



**EXCERCISE  
TIME!**

**FORMS!**

# EXERCISE TIME

Registrazione utente

Indirizzo email	Conferma indirizzo e-mail
<input type="text" value="Email"/>	<input type="text" value="Email"/>

Password	Conferma Password
<input type="password" value="Password"/>	<input type="password" value="Password"/>

---

Nome	Secondo Nome	Cognome
<input type="text" value="Nome"/>	<input type="text" value="Secondo nome"/>	<input type="text" value="Cognome"/>

---

Sesso	Città ed indirizzo
<input type="button" value="Maschio"/>	<input type="text" value="Citta e Indirizzo"/>

---

Interessi	Sistema Utilizzato
<input type="checkbox"/> Videogame <input type="checkbox"/> Arte <input checked="" type="checkbox"/> Fotografia <input type="checkbox"/> Alcool	<input type="radio"/> Microsoft Windows <input type="radio"/> Linux <input checked="" type="radio"/> MacOS <input type="radio"/> Microsoft DOS

Accetti le condizioni?

Accetto

Registrati

Create a HTML form like this  
(you can use tables).

Use the browser/system style!

# HTML RECAP!

- ▶ HTML is a markup language
- ▶ HTML page is defined by a tree structure
- ▶ Every node of tree is a HTML element
- ▶ Every element is defined by a tag
- ▶ An element can be container or void
- ▶ An element can be a block or an inline element
- ▶ With HTML you can send data
- ▶ The data can be sent by a form
- ▶ A form is a set of input tags
- ▶ You can send datas via GET or POST method
- ▶ So you can design the base structure of your web application!

You can find other HTML elements and attributes at:  
<https://developer.mozilla.org/en-US/docs/Web/HTML>

# 3. CSS

```
#webdev .css-slide:first {  
    color:#fff;  
    background:orange;  
}
```

# WHAT IS CSS?

CSS stands for "Cascading Style Sheet."

Cascading style sheets are used to format the layout of Web pages.

They can be used to define text styles, table sizes, and other aspects of Web pages that previously could only be defined in a page's HTML.

It uses **selectors** to access to the HTML Tree elements.

# CSS SYNTAX

CSS has a simple syntax and uses a number of English keywords to specify the names of various **style properties**.

A style sheet consists of a **list of rules**. Each rule or rule-set consists of **one or more selectors**, and a **declaration block**.

1. [SELECTORs] /\*declaration block \*/
- 2.
3. /\*list of rules\*/
4. {key}:{value};
5. ...
- 6.
7. }

# CSS SELECTORS

Selectors are used to target the HTML elements on our web pages that we want to style.

There are a wide variety of CSS selectors available, allowing for fine grained precision when selecting elements to style.

- ▶ Element Selector
- ▶ Simple Selector
- ▶ Attribute Selector
- ▶ Pseudo-class Selector
- ▶ Pseudo-element Selector
- ▶ Combinators and Multiple Selectors

# CSS SELECTORS - ELEMENT

This selector is just a **case-insensitive** match between the selector name and a given HTML element name.

This is the **simplest way** to target all elements of a given type.

CSS

```
/* All p elements are red */  
p {  
    color: red;  
}  
  
/* All div elements are blue */  
div {  
    color: blue;  
}
```

HTML

```
<p>What color do you like?</p>  
<div>I like blue.</div>  
<p>I prefer red!</p>
```

OUTPUT

```
What color do you like?  
  
I like blue.  
  
I prefer red!
```

# CSS SELECTORS - SIMPLE (CLASS)

The class selector consists **of a dot, '.', followed by a class name.**

A class name is **any value without spaces put within an HTML class attribute.**

It is up to you to choose a name for the class.

It is also worth knowing that **multiple elements in a document can have the same class value** and a **single element can have multiple class names separated by white space.**

# CSS SELECTORS - SIMPLE (CLASS)

## HTML

```
<ul>
  <li class="first done">Create an HTML document</li>
  <li class="second done">Create a CSS style sheet</li>
  <li class="third">Link them all together</li>
</ul>
```

## CSS

```
/* The element with the class "first" is bolded */
.first {
  font-weight: bold;
}

/* All the elements with the class "done" are strike through */
.done {
  text-decoration: line-through;
}
```

## OUTPUT

- Create an HTML document
- Create a CSS style sheet
- Link them all together

## CSS SELECTORS - SIMPLE (ID)

The ID selector consists of a hash/pound symbol (#), followed by the ID name of a given element.

Any element can have a unique ID name set with the id attribute.

It is up to you what name you choose for the ID.

It's the most efficient way to select a single element.

# CSS SELECTORS - SIMPLE (ID)

HTML

```
<p id="polite"> - "Good morning."</p>  
<p id="rude"> - "Go away!"</p>
```

CSS

```
#polite {  
    color: #f00;  
}  
  
#rude {  
    text-transform: uppercase;  
}
```

OUTPUT

```
- "Good morning."  
- "GO AWAY!"
```

# CSS SELECTORS - SIMPLE (UNIVERSAL)

The **universal selector (\*)** is the ultimate joker. It allows **selecting all elements in a page**. As it is rarely useful to apply a style to every element on a page, it is often **used in combination with other selectors**.

Careful when using the universal selector. As it applies to all elements, **using it in large web pages can have a perceptible impact on performance**: web pages can be displayed slower than expected.

There are not many instances where you'd want to use this selector.

# CSS SELECTORS - SIMPLE (UNIVERSAL)

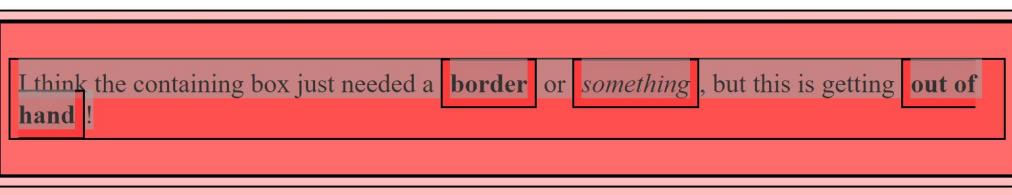
HTML

```
<div>  
  <p>I think the containing box just needed  
    a <b>border</b> or <em>something</em>,  
    but this is getting <b>out of hand</b>!</p>  
</div>
```

CSS

```
* {  
  border: 1px solid black;  
  background: rgba(255,0,0,0.25)  
}
```

OUTPUT



# CSS SELECTORS - COMBINATORS

In CSS, combinator allow you to **combine multiple selectors together**, which allows you to select elements inside other elements, or adjacent to other elements.

- ▶ **The descendant selector – (space) –** allows you to **select an element nested somewhere inside another element** (not necessarily a direct descendant; it could be a grandchild, for example)
- ▶ **The child selector – > –** allows you to select an **element that is an immediate child of another element**.
- ▶ **The adjacent sibling selector – + –** allows you to select an **element that is an immediate sibling of another element** (i.e. right next to it, at the same level in the hierarchy).
- ▶ **The general sibling selector – ~ –** allows you to select any **elements that are siblings of another element** (i.e. at the same level in the hierarchy, but not necessarily right next to it).

# CSS SELECTORS - COMBINATORS

## HTML

```
<section>  
  <h2>Heading 1</h2>  
  <p>Paragraph 1</p>  
  <p>Paragraph 2</p>  
  <div>  
    <h2>Heading 2</h2>  
    <p>Paragraph 3</p>  
    <p>Paragraph 4</p>  
  </div>  
</section>
```

## CSS

```
section p {  
  color: blue;  
}  
  
section > p {  
  background-color: yellow;  
}  
  
h2 + p {  
  text-transform: uppercase;  
}  
  
h2 ~ p {  
  border: 1px dashed black;  
}
```

# CSS SELECTORS - COMBINATORS

## Heading 1

PARAGRAPH 1

Paragraph 2

## Heading 2

PARAGRAPH 3

Paragraph 4

# CSS SELECTORS - COMBINATORS

- ▶ `section p` selects all the `<p>` elements – both the first two that are direct children of the `<section>` element, and the second two that are grandchildren of the `<section>` element (they are inside the `<div>` as well). All the paragraph text is therefore colored blue.
- ▶ `section > p` selects only the first two `<p>` elements, which are direct children of the `<section>` element (but not the second two, which are not direct children). Only the first two paragraphs therefore have a yellow background color.
- ▶ `h2 + p` selects only `<p>` elements that come directly after `<h2>` elements on the same hierarchy level – in this case the first and third paragraphs. These ones therefore have text all in uppercase.
- ▶ `h2 ~ p` selects any `<p>` elements on the same hierarchy level as (and coming after) `<h2>` elements – in this case all the paragraphs. All of them therefore have a dashed border.

# CSS SELECTORS - MULTIPLE

You can write **multiple** selectors separated by **commas**, to apply the same rule to multiple sets of selected elements at once.

CSS

```
p, li {  
    font-size: 1.6em;  
}  
  
h1, h2, h3, h4, h5, h6 {  
    font-family: helvetica, 'sans serif';  
}
```

# CSS SELECTORS - ATTRIBUTE

Attribute selectors are a special kind of selector that will match elements based on their attributes and attribute values. Their generic syntax consists of square brackets ([]) containing an attribute name followed by an optional condition to match against the value of the attribute.

Attribute selectors can be divided into two categories depending on the way they match attribute values: **Presence** and value attribute selectors and **Substring** value attribute selectors.

# CSS SELECTORS - ATTR. PRESENCE

These attribute selectors try to match an exact attribute value:

- ▶ **[name\_of\_attr]** - This selector will select all elements with the attribute name\_of\_attr, whatever its value.
- ▶ **[attr=val]** - This selector will select all elements with the attribute attr, but only if its value is val.
- ▶ **[attr~=val]** - This selector will select all elements with the attribute attr, but only if the value val is one of a space-separated list of values contained in attr's value, for example a single class in a space-separated list of classes.

# CSS SELECTORS - ATTR. PRESENCE

## HTML

```
Ingredients for my recipe: <i lang="fr-FR">Poulet basquaise</i>  
<ul>  
  <li data-quantity="1kg" data-vegetable>Tomatoes</li>  
  <li data-quantity="3" data-vegetable>Onions</li>  
  <li data-quantity="3" data-vegetable>Garlic</li>  
  <li data-quantity="700g" data-vegetable="not spicy like chili">Red  
pepper</li>  
  <li data-quantity="2kg" data-meat>Chicken</li>  
  <li data-quantity="optional 150g" data-meat>Bacon bits</li>  
  <li data-quantity="optional 10ml" data-vegetable="liquid">Olive  
oil</li>  
  <li data-quantity="25cl" data-vegetable="liquid">White wine</li>  
</ul>
```

## CSS

```
/* All elements with the attribute  
"data-vegetable" are given green text */  
[data-vegetable] {  
  color: green;  
}  
/* All elements with the attribute  
"data-vegetable" with the exact value  
"liquid" are given a golden  
background color */  
[data-vegetable="liquid"] {  
  background-color: goldenrod;  
}  
/* All elements with the attribute  
"data-vegetable", containing the value  
"spicy", even among others, are given a  
red text color */  
[data-vegetable~="spicy"] {  
  color: red;  
}
```

# CSS SELECTORS - ATTR. PRESENCE

Ingredients for my recipe: *Poulet basquaise*

- Tomatoes
- Onions
- Garlic
- Red pepper
- Chicken
- Bacon bits
- Olive oil
- White wine

# CSS SELECTORS - PSEUDO CLASS

A CSS **pseudo-class** is a keyword preceded by a **colon (:)** that is added on to the end of selectors to specify that you want to **style the selected elements only when they are in certain state.**

For example you might want to style an element only when it is being hovered over by the mouse pointer, or a checkbox when it is disabled or checked, or an element that is the first child of its parent in the HTML tree.

# CSS SELECTORS - PSEUDO CLASS

:active	:read-only
:checked	:required
:disabled	:visited
:empty	.... AND MORE!
:enabled	
:first	
:focus	
:hover	
:not()	
:nth-child()	
:nth-last-child()	

# CSS SELECTORS - PSEUDO CLASS

## CSS

```
/* These styles will style our link in all states */
a {
    color: blue;
    font-weight: bold;
}

/* We want visited links to be the same color as non visited links */
a:visited {
    color: blue;
}

/* We highlight the link when it is hovered (mouse), activated or focused (keyboard) */
a:hover,
a:active,
a:focus {
    color: darkred;
    text-decoration: none;
}
```

## HTML

```
<a href="https://google.it/" target="_blank">Google Search</a>
```

# CSS SELECTORS - PSEUDO ELEMENTS

Pseudo-elements are very much like pseudo-classes, but they have differences.

They are keywords – this time preceded by two colons (::) – that can be added to the end of selectors to select a certain part of an element.

- ▶ ::after
- ▶ ::before
- ▶ ::first-letter
- ▶ ::first-line
- ▶ ::selection
- ▶ ::backdrop

# CSS SELECTORS - PSEUDO ELEMENTS

CSS

```
/* All elements with an attribute "href", which values
   start with "http", will be added an arrow after its
   content (to indicate it's an external link) */
[href^=http]::after {
  content: '↗';
}
```

HTML

```
<ul>
  <li><a href="https://google.it">Google</a> search.</li>
  <li><a href="https://yahoo.it">Yahoo</a> search.</li>
</ul>
```

# CSS RULES - COLORS VALUES

In CSS **colors** can be specified using a text name, RGB values, HEX values, HSL values, RGBA values and HSLA values:

- ▶ Text : tomato
- ▶ RGB : rgb(255, 99, 71)
- ▶ HEX : #ff6347
- ▶ HSL : hsl(9, 100%, 64%)
- ▶ RGBA : rgba(255, 99, 71, .5) //trasparent
- ▶ HSLA : hsla(9, 100%, 64%, .5) //trasparent

# CSS RULES - TEXT COLOR

The color property **sets the color of the foreground content of the selected elements** (which is usually the text, but can also include a couple of other things, such as an underline or overline placed on text using the text-decoration property).

```
p {  
    color: #f00;  
}
```

# CSS RULES - TEXT DECORATION

The `text-decoration` CSS property specifies the **appearance** of decorative lines used on text.

```
/* Keyword values */
text-decoration: none;          /* No text decoration */
text-decoration: underline red;  /* Red underlining */
text-decoration: underline wavy red; /* Red wavy underlining */

/* Global values */
text-decoration: inherit;
text-decoration: initial;
text-decoration: unset;
```

It is a shorthand for setting one or more individual `text-decoration` values in a single declaration, which include `text-decoration-line`, `text-decoration-color`, and `text-decoration-style`.

# CSS RULES - TEXT TRANSFORM

The `text-transform` CSS property specifies **how to capitalize an element's text**. It can be used to make text appear in all-uppercase or all-lowercase, or with each word capitalized.

```
/* Keyword values */  
text-transform: capitalize;  
text-transform: uppercase;  
text-transform: lowercase;  
text-transform: none;
```

```
/* Global values */  
text-transform: inherit;  
text-transform: initial;  
text-transform: unset;
```

# CSS RULES - TEXT SHADOW

The text-shadow CSS property **adds shadows to text**. It accepts a comma-separated list of shadows to be applied to the text and text-decorations of the element. Each shadow is described by some combination of X and Y offsets from the element, blur radius, and color.

```
/* offset-x | offset-y | blur-radius | color */
text-shadow: 1px 1px 2px black;
/* offset-x | offset-y | color */
text-shadow: 5px 5px #558ABB;
/* offset-x | offset-y
 * Use defaults for color and blur-radius */
text-shadow: 5px 10px;
/* Global values */
text-shadow: inherit;
text-shadow: initial;
text-shadow: unset;
```

# CSS RULES - TEXT ALIGN

The `text-align` CSS property describes **how inline content like text is aligned in its parent block element**. `text-align` does not control the alignment of block elements, only their inline content.

```
/* Keyword values */  
text-align: left;  
text-align: right;  
text-align: center;  
text-align: justify;  
  
/* Global values */  
text-align: inherit;  
text-align: initial;  
text-align: unset;
```

# CSS RULES - LINE HEIGHT

The line-height property sets **the amount of space used for lines, such as in text**. On block-level elements, it specifies the minimum height of line boxes within the element. On non-replaced inline elements, it specifies the height that is used to calculate line box height.

```
/* Keyword value */  
line-height: normal;  
  
/* <length> values */  
line-height: 30px;  
  
/* <percentage> values */  
line-height: 34%;  
  
  
/* Global values */  
line-height: inherit;  
line-height: initial;  
line-height: unset;
```

# CSS RULES - LETTER SPACING

The letter-spacing CSS property specifies the spacing behavior between text characters (same for [word-spacing](#)).

```
/* <length> values */  
letter-spacing: 0.3em;  
letter-spacing: 3px;  
letter-spacing: .3px;  
  
/* Keyword values */  
letter-spacing: normal;  
  
/* Global values */  
letter-spacing: inherit;  
letter-spacing: initial;  
letter-spacing: unset;
```

# CSS RULES - FONT SIZE

The font-size property specifies the size of the font . Setting the font size may change the size of other items, since it is used to compute the value of the em and ex units.

```
/* <relative-size> values */  
font-size: smaller;  
font-size: larger;  
/* <length> values */  
font-size: 12px;  
font-size: 0.8em;
```

```
/* Global values */  
font-size: inherit;  
font-size: initial;  
font-size: unset;
```

# CSS RULES - FONT WEIGHT

The font-weight CSS property specifies the weight (or boldness) of the font. The font weights available to you will depend on the font-family you are using. Some fonts are only available in normal and bold.

```
/* Keyword values */
font-weight: normal;
font-weight: bold;

/* Keyword values relative to the parent */
font-weight: lighter;
font-weight: bolder;

/* Numeric keyword values */
font-weight: 100;
font-weight: 200;

/* Global values */
font-weight: inherit;
font-weight: initial;
```

# CSS RULES - FONT FAMILY

The font-family CSS property specifies a prioritized list of one or more font family names and/or generic family names for the selected element.

```
/* A font family name and a generic family name */
font-family: Gill Sans Extrabold, sans-serif;
font-family: "Goudy Bookletter 1911", sans-serif;

/* A generic family name only */
font-family: serif;
font-family: sans-serif;
font-family: monospace;
font-family: cursive;
font-family: fantasy;
font-family: system-ui;

/* Global values */
font-family: inherit;
font-family: initial;
font-family: unset;
```



**EXERCISE  
TIME!**

**STYLING  
LINKS!**

## EXERCISE TIME - STYLING LINKS

Create a set of rules to get the following result:

- ▶ Remove the standard underline in ALL links
- ▶ Change the default colors of link (normal, visited, hover and click)

# CSS RULES - BOX MODEL

When laying out a document, the browser's rendering engine represents each element as a rectangular box according to the standard CSS box model. CSS determines the size, position, and properties (color, background, border size, etc.) of these boxes.

Every box is composed of four parts (or areas), defined by their respective edges: **the content edge, padding edge, border edge, and margin edge**.



# CSS RULES - BORDERS

The **border** CSS property is a shorthand property for setting all individual border property values at once: **border-width**, **border-style**, and **border-color**. As with all shorthand properties, any individual value that is not specified is set to its corresponding initial value. Importantly, **border** cannot be used to specify a custom value for **border-image**, but instead sets it to its initial value, i.e., none.

```
border: <width> [<style> [<color>]]  
  
border: 1px;  
border: 2px dotted;  
border: medium dashed green;  
border: 1px dashed black;
```

# CSS RULES - MARGIN

The margin CSS property sets the margin area on all four sides of an element. It is a shorthand that sets all individual margins at once: **margin-top, margin-right, margin-bottom, and margin-left**.

```
/* Apply to all four sides */
margin: 1px;
/* vertical | horizontal */
margin: 5% auto;
/* top | horizontal | bottom */
margin: 1em auto 2em;
/* top | right | bottom | left */
margin: 2px 1em 0 auto;
/* Global values */
margin: inherit;
margin: initial;
margin: unset;
```

# CSS RULES - PADDING

The padding CSS property sets the padding area on all four sides of an element. It is a shorthand that sets all individual paddings at once: **padding-top, padding-right, padding-bottom, and padding-left**.

```
/* Apply to all four sides */
padding: 1em;
/* vertical | horizontal */
padding: 5% 10%;
/* top | horizontal | bottom */
padding: 1em 2em 2em;
/* top | right | bottom | left */
padding: 5px 1em 0 1em;
/* Global values */
padding: inherit;
padding: initial;
padding: unset;
```

# CSS RULES - DISPLAY

The display CSS property specifies the type of rendering box used for an element. In HTML, default display property values are taken from behaviors described in the HTML specifications or from the browser/user default stylesheet. The default value in XML is inline, including SVG elements.

In addition to the many different display box types, **the value none lets you turn off the display of an element**; when you use none, all descendant elements also have their display turned off. **The document is rendered as though the element doesn't exist in the document tree.**

```
display: block;  
display: inline;  
display: none;  
display: inline-block;  
display: inline-table;  
display: inline-flex;  
display: inline-grid;
```

# CSS RULES - OPACITY

The opacity property sets the opacity level for an element. The opacity-level describes the transparency-level, where 1 is not transparent at all, 0.5 is 50% see-through, and 0 is completely transparent. When using the opacity property to add transparency to the background of an element, all of its child elements become transparent as well. This can make the text inside a fully transparent element hard to read.  
The opacity rule does not collapse the elements but maintains the events on the elements.

```
opacity: 0.5;
```

# CSS RULES - VISIBILITY

The visibility property in CSS has two different functions. It hides rows and columns of a table, and it also hides an element without changing the layout.

The visibility rule does not maintain the events on the elements, anyway it is recommended to do not use this rule anymore.

This rule is used only on Flexboxes with collapse value.

```
visibility: hidden;
```

```
visibility: visible;
```

```
visibility: collapsed;
```

# CSS RULES - POSITION

The position CSS property specifies how an element is positioned in a document.

- ▶ A positioned element is an element whose computed position value is either relative, absolute, fixed, or sticky. (In other words, it's anything except static.)
- ▶ A relatively positioned element is an element whose computed position value is relative. The top and bottom properties specify the vertical offset from its normal position; the left and right properties specify the horizontal offset.
- ▶ An absolutely positioned element is an element whose computed position value is absolute or fixed. The top, right, bottom, and left properties specify offsets from the edges of the element's containing block. (The containing block is the ancestor to which the element is relatively positioned.) If the element has margins, they are added to the offset.
- ▶ A stickily positioned element is an element whose computed position value is sticky. It's treated as relatively positioned until its containing block crosses a specified threshold, at which point it is treated as fixed.

# CSS RULES - POSITION

The top, right, bottom, and left properties determine the final location of positioned elements.

Most of the time, absolutely positioned elements that have height and width set to auto are sized so as to fit their contents. However, non-replaced absolutely positioned elements can be made to fill the available vertical space by specifying both top and bottom and leaving height unspecified (that is, auto). They can likewise be made to fill the available horizontal space by specifying both left and right and leaving width as auto.

Except for the case just described of absolutely positioned elements filling the available space:

- ▶ If both top and bottom are specified (technically, not auto), **top wins**.
- ▶ If both left and right are specified, **left wins when direction is ltr** (English, horizontal Japanese, etc.) and **right wins when direction is rtl** (Persian, Arabic, Hebrew, etc.).

# CSS RULES - POSITION

```
position: static;  
  
position: relative;  
top: 65px; left: 65px;  
  
position: absolute;  
top: 40px; left: 40px;  
  
position: fixed;  
bottom: 20px;  
  
position: sticky;  
top: 20px;
```

# CSS RULES - FLOAT

The float CSS property specifies that an element should be placed along the left or right side of its container, allowing text and inline elements to wrap around it. **The element is removed from the normal flow of the web page, though still remaining a part of the flow (in contrast to absolute positioning).**

```
/* Keyword values */  
float: left;  
float: right;  
float: none;  
float: inline-start;  
float: inline-end;  
  
/* Global values */  
float: inherit;  
float: initial;  
float: unset;
```

# CSS RULES - CLEAR

The clear CSS property specifies whether an element **can be next to floating elements that precede it or must be moved down (cleared) below them**. The clear property applies to both floating and non-floating elements.

```
/* Keyword values */
clear: none;
clear: left;
clear: right;
clear: both;
clear: inline-start;
clear: inline-end;

/* Global values */
clear: inherit;
clear: initial;
clear: unset;
```

# CSS RULES - CLEARFIX

If an element contains only floated elements, its height collapses to nothing. If you want it to always be able to resize, so that it contains floating elements inside it, you need to self-clear its children. This is called clearfix, and one way to do it is clear a replaced ::after pseudo-element on it.

```
#container::after {  
    content: "";  
    display: block;  
    clear: both;  
}
```



**EXCERCISE  
TIME!**

**RED  
CORNERS!**

# CSS - BOX WITH RED CORNERS



Try to make a container with some content (width and height are dynamic) that has four corners as the image on top.

Use the topics covered until now and think outside the box!

# CSS RULES - BACKGROUND

The CSS background shorthand property lets you adjust all of the available background style options at once, including color image, origin and size, repeat method, and other features.

```
/* Using a <background-color> */  
background: green;  
  
/* Using a <bg-image> and <repeat-style> */  
background: url("test.jpg") repeat-y;  
  
/* Using a <box> and <background-color> */  
background: border-box red;  
  
/* A single image, centered and scaled */  
background: no-repeat center/80% url("../img/image.png");
```

# CSS RULES - BACKGROUND

The `background-size` CSS property specifies the size of an element's background image. The image can be left to its natural size, stretched to a new size, or constrained to fit the available space while preserving its intrinsic proportions.

```
/* Keyword values */

background-size: cover;
background-size: contain;

/* One-value syntax */
/* the width of the image (height becomes 'auto') */

background-size: 50%;
background-size: 3.2em;
background-size: 12px;
background-size: auto;
```

# CSS RULES - BACKGROUND

The background-position CSS property sets the initial position for each defined background image.

```
/* Keyword values */
background-position: top;
background-position: bottom;
background-position: left;
background-position: right;
background-position: center;

/* <percentage> values */
background-position: 25% 75%;

/* <length> values */
background-position: 0 0;
background-position: 1cm 2cm;
```

# CSS RULES - CURSOR

The cursor CSS property specifies which mouse cursor to display when the mouse pointer is over an element.

```
/* Keyword value only */

cursor: pointer;
cursor: auto;

/* Using URL and coordinates */

cursor: url(cursor1.png) 4 12, auto;
cursor: url(cursor2.png) 2 2, pointer;

/* Global values */

cursor: inherit;
cursor: initial;
cursor: unset;
```

# CSS RULES - OVERFLOW

The overflow CSS property sets what to do when an element's content is too big to fit in its block formatting context. It is a shorthand for overflow-x and overflow-y.

```
/* Keyword values */

overflow: visible;
overflow: hidden;
overflow: clip;
overflow: scroll;
overflow: auto;
overflow: hidden visible;

/* Global values */

overflow: inherit;
overflow: initial;
overflow: unset;
```

# CSS RULES - Z-INDEX

The z-index CSS property sets the z-order of a positioned element and its descendants. Overlapping elements with a larger z-index cover those with a smaller one.

```
/* Keyword value */
z-index: auto;
/* <integer> values */
z-index: 0;
z-index: 3;
z-index: 289;
z-index: -1; /* Negative values to lower the priority */
```

# CSS - IMPORTING FILES

CSS can be added to HTML by linking to a separate stylesheet file, importing files from existing stylesheets, embedding CSS in a style tag, or adding inline styles directly to HTML elements.

## Link a stylesheet file into HTML file

```
<link rel="stylesheet" [type="text/css"] href="mystyles.css" [media="screen"] />
```

The `rel` attribute is set to `stylesheet` to tell the browser that the linked file is a Cascading Style Sheet (CSS).

The `type` attribute is not required in HTML5.

The `href` attribute is where you specify the path to your CSS file.

The `media` attribute in a link tag specifies when the CSS rules are to be applied. `screen` indicates for use on a computer screen.

- ▶ `projection` for projected presentations.
- ▶ `handheld` for handheld devices (typically with small screens).
- ▶ `print` to style printed web pages.
- ▶ `all` (default value) This is what most people choose.

You can leave off the `media` attribute completely if you want your styles to be applied for all media types.

# CSS - IMPORTING FILES

CSS can be added to HTML by linking to a separate stylesheet file, importing files from existing stylesheets, embedding CSS in a style tag, or adding inline styles directly to HTML elements.

Load a stylesheet file with the @import rule

```
@import "newstyles.css";
```

Useful when the web project has a lot of HTML files, because if you want add a new CSS file in your project, you must add a new HTML link line to every single HTML file. In this way you must modify only the one CSS file where you will import the other files.

Every imported CSS file requires an additional HTTP request which can slow down page rendering.



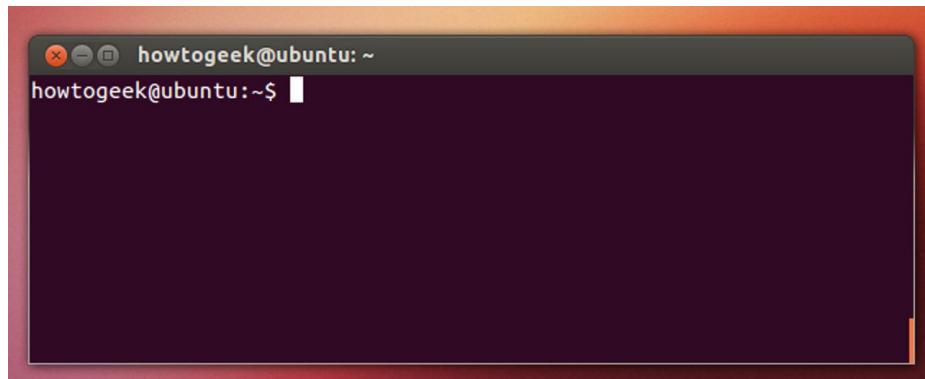
**EXCERCISE  
TIME!**

**LINUX  
TERMINAL!**

# EXERCISE TIME - LINUX TERMINAL

Create a similar Linux Terminal:

- ▶ Full screen
- ▶ Windowed
- ▶ Classic black and green colors
- ▶ Personalise the colors!



# CSS RESPONSIVE - LONG TIME AGO

It is recommended that you set your resolution to 1024x768!



[https://web.archive.org/web/20090122225904if\\_/http://mondoroms.vai.li:80/](https://web.archive.org/web/20090122225904if_/http://mondoroms.vai.li:80/)

## CSS RESPONSIVE - THE OLD PROBLEM

Some time ago the computers **did not have very high resolutions** than today (consider that today you get to 8K whereas before 1024 were much stuff).

Furthermore, there were **no smart devices that could navigate the web pages** we access through computers.

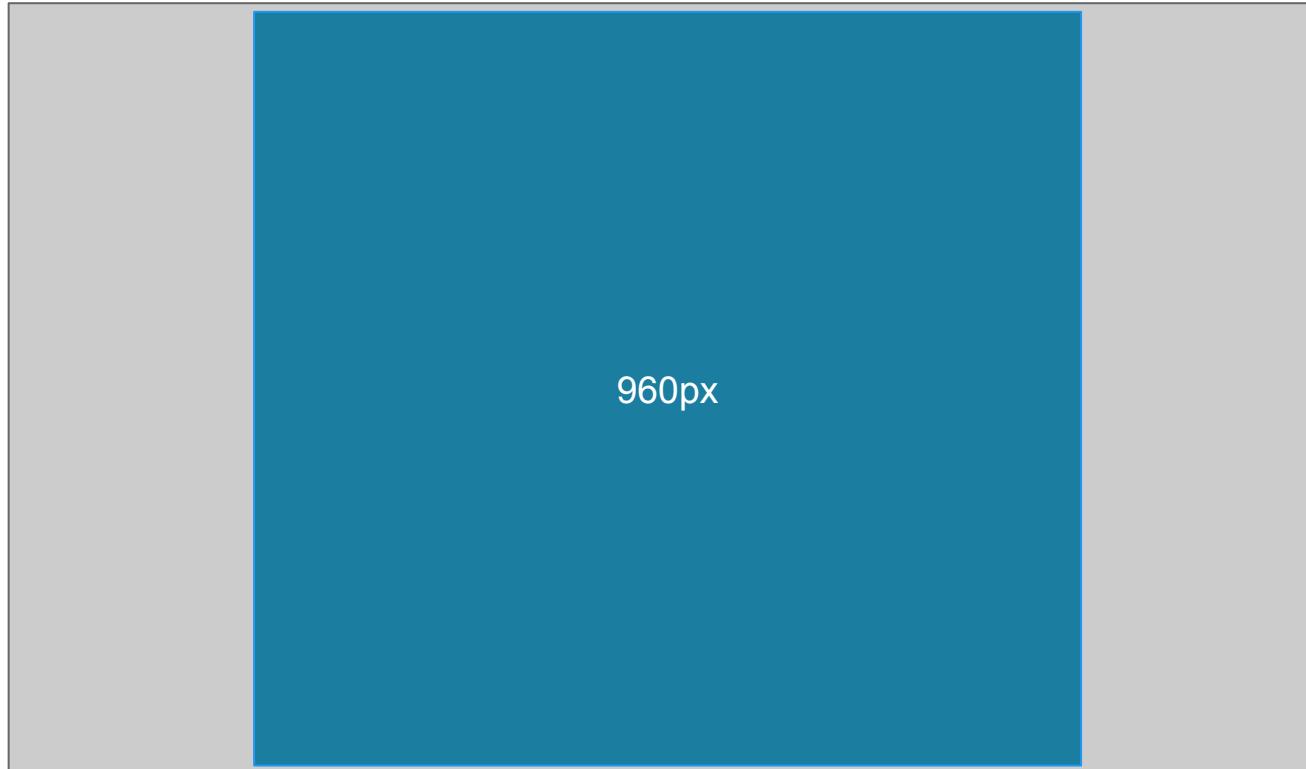
So the websites were made according to **standard measures**.

Then we had to make an analysis of the average resolution used (which was 800x600px), obtaining a maximum development width of **768px** (we also had to consider the same web browser as occluding a side space).

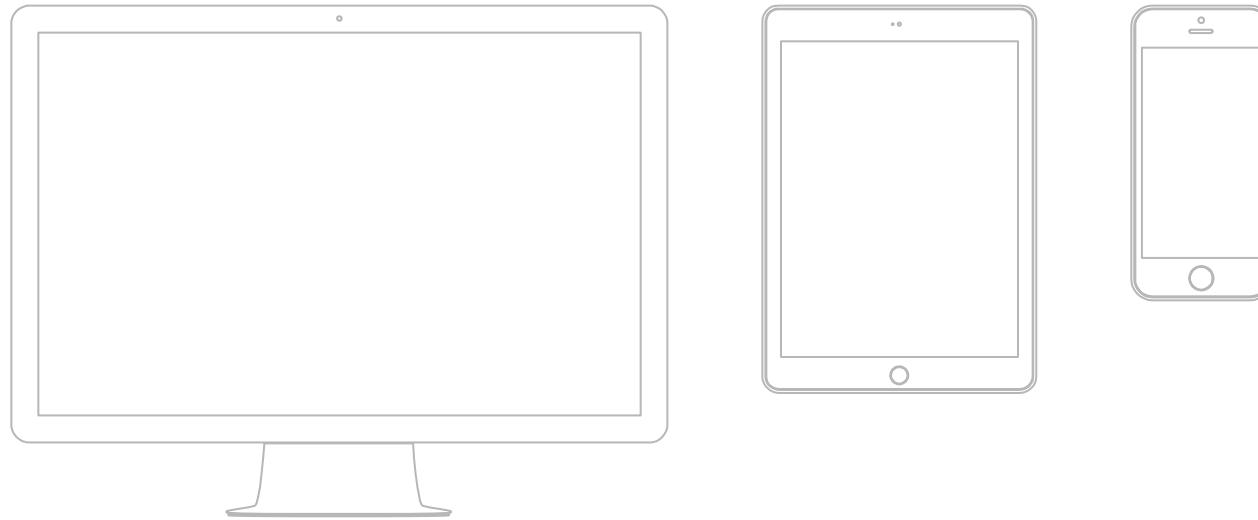
Then with the 1024x768px resolution, we reached a maximum width of **960px** .

# CSS RESPONSIVE - THE OLD PROBLEM

Screen resolution (any)



# CSS RESPONSIVE - THE NEW PROBLEM



With the advent of new multimedia devices, capable of surfing on the same websites, new problems emerged with screen resolution.

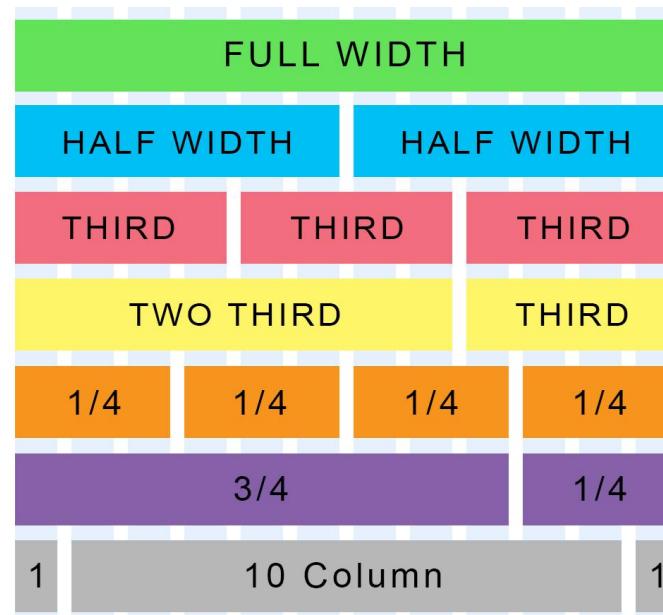
# CSS RESPONSIVE - THE NEW PROBLEM

- ▶ It is no longer possible to create websites with a maximum static width.
- ▶ You have to resize parts of the site depending on your device
- ▶ You have to be careful about the units you will be using
- ▶ It needs a new unit of measure to handle the widths of space!

# CSS RESPONSIVE - THE COLs SYSTEM

The basic idea is to organize the various components of the web page through a new unit of measure, this is called **column**.

A web element (from body to any other) has a total width of **12 columns**.



# CSS RESPONSIVE - THE COLs SYSTEM

Quick example:

```
* {  
    box-sizing: border-box;  
}  
  
.row::after {  
    content: "";  
    clear: both;  
    display: block;  
}  
  
[class*="col-"] {  
    float: left;  
    padding: 15px;  
}  
  
.col-1 {width: 8.33%;}  
.col-2 {width: 16.66%;}  
.col-3 {width: 25%;}  
.col-4 {width: 33.33%;}  
.col-5 {width: 41.66%;}  
.col-6 {width: 50%;}  
.col-7 {width: 58.33%;}  
.col-8 {width: 66.66%;}  
.col-9 {width: 75%;}  
.col-10 {width: 83.33%;}  
.col-11 {width: 91.66%;}  
.col-12 {width: 100%;}
```

# CSS RESPONSIVE - MEDIA QUERY

The @media CSS at-rule can be used to apply styles based on the result of one or more media queries, which test a device's type, specific characteristics, and environment.

In CSS, the @media rule may be placed at the top level of your code or nested inside any other conditional group at-rule.

```
/* Media query */  
@media screen and (min-width: 900px) {  
    div {  
        padding: 1rem 3rem;  
    }  
}
```

# CSS RESPONSIVE - MEDIA TYPES

Media types describe the general category of a device. Unless you use the not or only logical operators, the media type is optional and the all type will be implied.

- ▶ **All**: for all devices
- ▶ **Print**: for material viewed in print preview mode
- ▶ **Screen**: for computer screens

```
/* Media query */  
@media [TYPE] and (min-width: 900px) {  
    div {  
        padding: 1rem 3rem;  
    }  
}
```

# CSS RESPONSIVE - C.S. QUICK FIX

So add the support for mobile devices

```
/* For desktop: */
.col-1 {width: 8.33%;}
.col-2 {width: 16.66%;}
.col-3 {width: 25%;}
.col-4 {width: 33.33%;}
.col-5 {width: 41.66%;}
.col-6 {width: 50%;}
.col-7 {width: 58.33%;}
.col-8 {width: 66.66%;}
.col-9 {width: 75%;}
.col-10 {width: 83.33%;}
.col-11 {width: 91.66%;}
.col-12 {width: 100%;}

@media only screen and (max-width: 768px) {
    /* For mobile devices: */
    [class*="col-"] {
        width: 100%;
    }
}
```

# CSS RESPONSIVE - MOBILE FIRST

Mobile First means designing for mobile before designing for desktop or any other device

```
/* For mobile devices: */
[class*="col-"] {
    width: 100%;
}

@media only screen and (min-width: 768px) {
    /* For desktop: */
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;}
}
```

# CSS RESPONSIVE - BREAKPOINT

Mobile First means designing for mobile before designing for desktop or any other device

```
/* For mobile phones: */  
[class*="col-"] {  
    width: 100%;  
}  
  
@media only screen and (min-width: 600px) {  
    /* For tablets: */  
    .col-m-1 {width: 8.33%;}  
    .col-m-2 {width: 16.66%;}  
    .col-m-3 {width: 25%;}  
    .col-m-4 {width: 33.33%;}  
    .col-m-5 {width: 41.66%;}  
    .col-m-6 {width: 50%;}  
    .col-m-7 {width: 58.33%;}  
    .col-m-8 {width: 66.66%;}  
    .col-m-9 {width: 75%;}  
    .col-m-10 {width: 83.33%;}  
    .col-m-11 {width: 91.66%;}  
    .col-m-12 {width: 100%;}  
}
```

```
@media only screen and (min-width: 768px) {  
    /* For desktop: */  
    .col-1 {width: 8.33%;}  
    .col-2 {width: 16.66%;}  
    .col-3 {width: 25%;}  
    .col-4 {width: 33.33%;}  
    .col-5 {width: 41.66%;}  
    .col-6 {width: 50%;}  
    .col-7 {width: 58.33%;}  
    .col-8 {width: 66.66%;}  
    .col-9 {width: 75%;}  
    .col-10 {width: 83.33%;}  
    .col-11 {width: 91.66%;}  
    .col-12 {width: 100%;}  
}
```

# CSS - FLEXBOX SYSTEM

Until now, the only reliable cross browser-compatible tools available for creating CSS layouts were things like **floats** and **positioning**. These are fine and they work, but in some ways they are also rather limiting and frustrating.

The following simple layout requirements are either difficult or impossible to achieve with such tools, in any kind of convenient, flexible way:

- ▶ Vertically centering a block of content inside its parent.
- ▶ Making all the children of a container take up an equal amount of the available width/height, regardless of how much width/height is available.
- ▶ Making all columns in a multiple column layout adopt the same height even if they contain a different amount of content.

Flexbox layout is most appropriate to the components of an application, and small-scale layouts, while the Grid layout is intended for larger scale layouts.

# CSS - FLEXBOX CONTAINERS

To start with, we need to select which elements are to be laid out as flexible boxes. To do this, we set a special value of display on the parent element of the elements you want to affect.

HTML

```
<section>
  <article>
    <h2>First article</h2>
    <p>Content</p>
  </article>
  ....
</section>
```

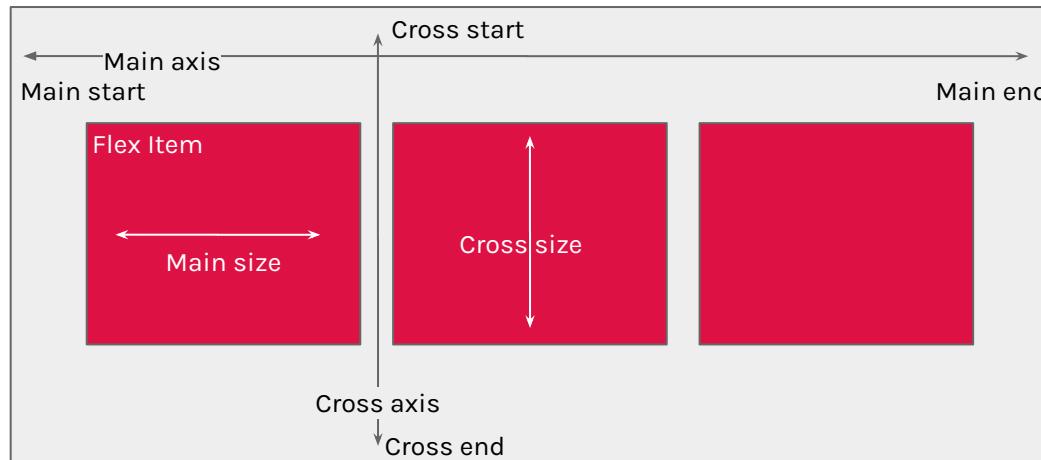
CSS

```
section {
  display: flex;
}
```

This causes the `<section>` element to become a **flex container**, and its children to become **flex items**.

# CSS - FLEXBOX MODEL

Flex Container



- The **main axis** is the axis running in the direction the flex items are being laid out in (e.g. as rows across the page, or columns down the page.) The start and end of this axis are called the **main start and main end**.
- The **cross axis** is the axis running perpendicular to the direction the flex items are being laid out in. The start and end of this axis are called the **cross start and cross end**.
- The **parent element** that has `display: flex` set on it (the `<section>` in our example) is called the flex container.
- The items being laid out as flexible boxes inside the flex container are called **flex items** (the `<article>` elements in our example).

# CSS - FLEXBOX COLS AND ROWS

Flexbox provides a property called **flex-direction** that specifies what direction the main axis runs in (what direction the flexbox children are laid out in).

The default value is set to **row**, which causes them to be laid out in a row in the direction your browser's default language works in (left to right, in the case of an English browser).

```
flex-direction: column; /* row */
```

You'll see that this puts the items back in a column layout, much like they were before we added any CSS.

You can also lay out flex items in a reverse direction using the **row-reverse** and **column-reverse** values.

# CSS - FLEXBOX WRAP

One issue that arises when you have a fixed amount of width or height in your layout is that eventually your flexbox children will overflow their container, breaking the layout.

One way in which you can fix this is to add the following declaration to **flex container** rule:

```
flex-wrap: wrap;
```

And add the following declaration to **flex items** rule:

```
flex: 200px;
```

Also there is a shorthand to do this in the flex container:

```
flex-flow: row wrap;
```

# CSS - FLEX ITEMS SIZE

This is a unitless proportion value that dictates how much of the available space along the main axis each flex item will take up compared to other flex items.

```
article {  
  flex: 1;  
}
```

In this case, we are giving each `<article>` element the same value (a value of 1), which means they will all take up an equal amount of the spare space left after things like padding and margin have been set. It is relative to other flex items, meaning that giving each flex item a value of 400000 would have exactly the same effect.

# CSS - FLEX ITEMS SIZE

Now when you refresh, you'll see that the third `<article>` takes up twice as much of the available width as the other two.

```
article:nth-child(3) {  
  flex: 2;  
}
```

There are now four proportion units available in total (since  $1 + 1 + 2 = 4$ ). The first two flex items have one unit each, so they take 1/4 of the available space each. The third one has two units, so it takes up 2/4 of the available space (or one-half).

# CSS - FLEX ITEMS SIZE

You can also specify a minimum size value inside the flex value.

```
article {  
  flex: 1 200px;  
}  
  
article:nth-child(3) {  
  flex: 2 200px;  
}
```

This basically states "Each flex item will first be given 200px of the available space. After that, the rest of the available space will be shared out according to the proportion units."

# CSS - FLEX ITEMS SIZE

This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion. It dictates what amount of the available space inside the flex container the item should take up.

```
article {  
  flex-grow: 4; /* default 0 */  
}
```

If all items have **flex-grow** set to 1, the remaining space in the container will be distributed equally to all children. If one of the children has a value of 2, the remaining space would take up twice as much space as the others.

Negative numbers are invalid.

# CSS - FLEX ALIGNMENTS

You can also use flexbox features to align flex items along the main or cross axis.

```
div {  
  display: flex;  
  align-items: center;  
  justify-content: space-around;  
}
```

The **align-items** rule controls where the flex items sit on the cross axis.

- ▶ By default, the value is **stretch**, which stretches all flex items to fill the parent in the direction of the cross axis. If the parent does not have a fixed width in the cross axis direction, then all flex items will become as long as the longest flex item.
- ▶ The **center** value in above code causes the items to maintain their intrinsic dimensions, but be centered along the cross axis.
- ▶ You can also have values like **flex-start** and **flex-end**, which will align all items at the start and end of the cross axis respectively.
- ▶ You can override the align-items behavior for individual flex items by applying the **align-self** property to them.

# CSS - FLEX ALIGNMENTS

You can also use flexbox features to align flex items along the main or cross axis.

```
div {  
  display: flex;  
  align-items: center;  
  justify-content: space-around;  
}
```

The **justify-content** controls where the flex items sit on the main axis.

- ▶ The default value is **flex-start**, which makes all the items sit at the start of the main axis.
- ▶ You can use **flex-end** to make them sit at the end.
- ▶ The **center** is also a value for **justify-content**, and will make the flex items sit in the center of the main axis.
- ▶ The value used above, **space-around**, is useful because it distributes all the items evenly along the main axis, with a bit of space left at either end.
- ▶ There is another value, **space-between**, which is very similar to **space-around** except that it does not leave any space at either end.

# CSS - FLEX ORDERING

Flexbox also has a feature for changing the layout order of flex items, without affecting the source order. This is another thing that is impossible to do with traditional layout methods.

```
button:first-child {  
  order: 1;  
}
```

- ▶ The default value is `0`.
- ▶ Flex items with higher order values set on them will appear later in the display order than items with lower order values.
- ▶ Flex items with the same order value will appear in their source order.

You can set negative order values to make items appear earlier than items with `0` set.

```
button:last-child {  
  order: -1;  
}
```

# CSS - FLEX (LONGHAND VERSION)

The **flex** is a shorthand property that can specify up to three different values:

- ▶ The unitless proportion value we discussed above. This can be specified individually using the **flex-grow** longhand property.
- ▶ A second unitless proportion value is the **flex-shrink**, that comes into play when the flex items are overflowing their container. This specifies how much of the overflowing amount is taken away from each flex item's size, to stop them overflowing their container.
- ▶ The minimum size value we discussed above. This can be specified individually using the **flex-basis** longhand value.



**EXCERCISE  
TIME!**

**FLEXBOX  
NAVIGATION**

# EXERCISE TIME - FLEXBOX NAV.

Try to make the following navigation bar with flexbox system:

Desktop



Tablet



Mobile



# CSS - GRID SYSTEM

CSS Grid Layout (aka “Grid System”), is a two-dimensional grid-based layout system that aims to do nothing less than completely change the way we design column-based user interfaces. CSS has always been used to lay out our web pages, but it is never done a very good job of it.

Firstly, we used tables, then floats, positioning and inline-block, but all of these methods were essentially hacks and left out a lot of important functionality (vertical centering, for instance).

Secondly, Flexbox helped out, but it is intended for simpler one-dimensional layouts, not complex two-dimensional ones.

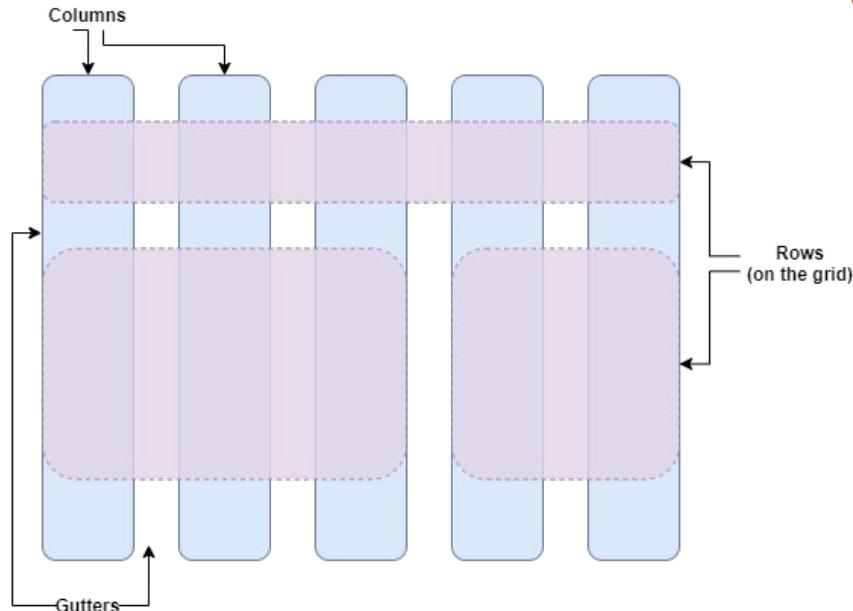
Grid is the very first CSS module created specifically to solve the layout problems we've all been hacking our way around for as long as we have been making websites.

# CSS - GRID MODEL

A grid is a collection of horizontal and vertical lines creating a pattern against which we can line up our design elements.

They help us to create designs where elements don't jump around or change width as we move from page to page, providing greater consistency on our websites.

A grid will typically have **columns**, **rows**, and then **gaps** between each row and column , commonly referred to as **gutters**.



# CSS - GRID LAYOUT

To define a grid we use the `grid` value of the `display` property. As with Flexbox, this switches on Grid Layout, and all of the direct children of the container become grid items.

Add this to the CSS inside your file:

```
.container {  
    display: grid;  
}
```

Unlike flexbox, the items will not immediately look any different.

Declaring `display: grid` gives you a one column grid, so your items will continue to display one below the other as they do in normal flow.

To see something that looks more grid-like, we will need to add some columns to the grid.

```
.container {  
    display: grid;  
    grid-template-columns: 200px 200px 200px;  
}
```

This defines three columns of 200px.

# CSS - GRID LAYOUT (FLEXIBLE)

In addition to creating grids using lengths and percentages, we can use the `fr` unit to flexibly size grid rows and columns. This unit represents one fraction of the available space in the grid container.

```
.container {  
    display: grid;  
    grid-template-columns: 1fr 1fr 1fr; //or repeat(3, 1fr)  
}
```

You should now see that you have flexible tracks. The `fr` unit distributes space in proportion, therefore you can give different positive values to your tracks,

```
.container {  
    display: grid;  
    grid-template-columns: 2fr 1fr 1fr;  
}
```

The first track now gets 2fr of the available space and the other two tracks get 1fr, making the first track larger.

You can mix `fr` units and fixed length tracks, in such a case the space needed for the fixed tracks is taken away before the space is distributed to the other tracks.

# CSS - GRID GAPS

To create gaps between tracks we use the properties `column-gap` for gaps between columns, `row-gap` for gaps between rows, and `gap` to set both at once.

```
.container {  
    display: grid;  
    grid-template-columns: 2fr 1fr 1fr;  
    grid-gap: 20px /*row gap and column gap*/;  
    gap: 20px /*row gap and column gap*/;  
}
```

This is a shorthand for row-gap and column-gap rules.

The grid- prefix is deprecated (but who knows, may never actually be removed from browsers). Essentially grid-gap renamed to gap.

# CSS - GRID EXPLICIT

You can choose to explicitly name the lines. Note [the bracket syntax](#) for the line names:

```
.container {  
  display: grid;  
  grid-template-columns: [first] 40px [line2] 50px [line3] auto [col4-start] 50px [five] 40px [end];  
  grid-template-rows: [row1-start] 25% [row1-end] 100px [third-line] auto [last-line];  
}
```

Note that [a line can have more than one name](#). For example, here the second line will have two names: row1-end and row2-start:

```
.container {  
  display: grid;  
  grid-template-rows: [row1-start] 25% [row1-end row2-start] 25% [row2-end];  
}
```

# CSS - GRID MINMAX

A fairly basic fact about the web is that you never really know how tall something is going to be; additional content or larger font sizes can cause problems with designs that attempt to be pixel perfect in every dimension.

The `minmax()` function lets us set a minimum and maximum size for a track, for example `minmax(100px, auto)`. The minimum size is 100 pixels, but the maximum is auto, which will expand to fit the content.

```
.container {  
    display: grid;  
    grid-template-columns: repeat(3, 1fr);  
    grid-auto-rows: minmax(100px, auto);  
    gap: 20px;  
}
```

# CSS - GRID AUTO-FILL

Sometimes it is helpful to be able to ask grid to create as many columns as will fit into the container. We do this by setting the value of `grid-template-columns` using `repeat()` notation, but instead of passing in a number, pass in the keyword `auto-fill`.

For the second parameter of the function we use `minmax()`, with a minimum value equal to the minimum track size that we would like to have, and a maximum of `1fr`.

```
.container {  
    display: grid;  
    grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));  
    grid-auto-rows: minmax(100px, auto);  
    gap: 20px;  
}
```

# CSS - GRID LINE PLACEMENT

A grid always has lines, which lines start at 1 and relate to the Writing Mode of the document. Therefore in Italian or English, column line 1 is on the left hand side of the grid and row line 1 at the top. In Arabic column line 1 would be on the right hand side, as Arabic is written right to left.

We can place things according to these lines by specifying the start and end line.

```
.item {  
    grid-column-start: <number> | <name> | span <number> | span <name> | auto;  
    grid-column-end: <number> | <name> | span <number> | span <name> | auto;  
    grid-row-start: <number> | <name> | span <number> | span <name> | auto;  
    grid-row-end: <number> | <name> | span <number> | span <name> | auto;  
}
```

- ▶ **<line>** - can be a number to refer to a numbered grid line, or a name to refer to a named grid line
- ▶ **span <number>** - the item will span across the provided number of grid tracks
- ▶ **span <name>** - the item will span across until it hits the next line with the provided name
- ▶ **auto** - indicates auto-placement, an automatic span, or a default span of one

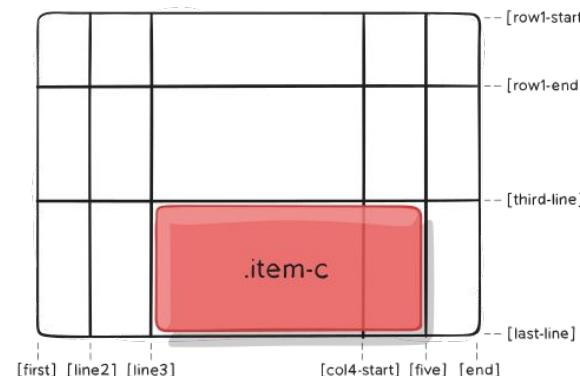
# CSS - GRID LINE PLACEMENT (SHORT)

There is a shorthand for the for grid-column-start + grid-column-end, and grid-row-start + grid-row-end rules, respectively.

```
.item {  
  grid-column: <start-line> / <end-line> | <start-line> / span <value>;  
  grid-row: <start-line> / <end-line> | <start-line> / span <value>;  
}
```

<start-line> / <end-line> - each one accepts all the same values as the longhand version, including span

```
.item-c {  
  grid-column: 3 / span 2;  
  grid-row: third-line / 4;  
}
```



# CSS - GRID TEMPLATE PLACEMENT

An alternative way to place items on your grid is to use the `grid-template-areas` property and giving the various elements of your design a name.

Defines a grid template by referencing the names of the grid areas which are specified with the `grid-area` property.

Repeating the name of a grid area causes the content to span those cells.

A period signifies an empty cell.

The syntax itself provides a visualization of the structure of the grid.

```
.container {  
    grid-template-areas:  
        " | . | none | ..."   
        "...";  
}
```

- ▶ `<grid-area-name>` - the name of a grid area specified with `grid-area`
- ▶ `.` - a period signifies an empty grid cell
- ▶ `none` - no grid areas are defined

# CSS - GRID TEMPLATE PLACEMENT

```
.container {  
    display: grid;  
    grid-template-columns: 50px 50px 50px 50px;  
    grid-template-rows: auto;  
    grid-template-areas:  
        "header header header header"  
        "main main . sidebar"  
        "footer footer footer footer";  
}  
  
.item-a {  
    grid-area: header;  
}  
  
.item-b {  
    grid-area: main;  
}  
  
.item-c {  
    grid-area: sidebar;  
}  
  
.item-d {  
    grid-area: footer;  
}
```

This will create a grid that is four columns wide by three rows tall.

- ▶ The entire top row will be composed of the header area.
- ▶ The middle row will be composed of two main areas, one empty cell, and one sidebar area.
- ▶ The last row is all footer.

Each row in your declaration needs to have the same number of cells.

You can use any number of adjacent periods to declare a single empty cell. As long as the periods have no spaces between them they represent a single cell.

Notice that you're not naming lines with this syntax, just areas.

# CSS - GRID TEMPLATE AREAS

Gives an item a name so that it can be referenced by a template created with the grid-template-areas property.

```
.item {  
  grid-area: <name> | <row-start> / <column-start> / <row-end> / <column-end>;  
}
```

- ▶ `<name>` - a name of your choosing
- ▶ `<row-start> / <column-start> / <row-end> / <column-end>` - can be numbers or named lines

```
.item-d {  
  grid-area: header;  
}
```

# CSS - GRID JUSTIFYING (FROM PARENT)

Aligns grid items along the inline (row) axis. This value applies to all grid items inside the container.

```
.container {  
  justify-items: start | end | center | stretch;  
}
```

- ▶ **start** - aligns items to be flush with the start edge of their cell
- ▶ **end** - aligns items to be flush with the end edge of their cell
- ▶ **center** - aligns items in the center of their cell
- ▶ **stretch** - fills the whole width of the cell (this is the default)

# CSS - GRID JUSTIFYING (FROM ITEM)

Aligns a grid item inside a cell along the inline (row) axis. This value applies to a grid item inside a single cell.

```
.item {  
  justify-self: start | end | center | stretch;  
}
```

- ▶ **start** - aligns the grid item to be flush with the start edge of the cell
- ▶ **end** - aligns the grid item to be flush with the end edge of the cell
- ▶ **center** - aligns the grid item in the center of the cell
- ▶ **stretch** - fills the whole width of the cell (this is the default)

# CSS - GRID ALIGNMENT (FROM PARENT)

Aligns grid items along the block (column) axis (as opposed to justify-items which aligns along the inline (row) axis). This value applies to all grid items inside the container.

```
.container {  
  align-items: start | end | center | stretch;  
}
```

- ▶ **start** - aligns items to be flush with the start edge of their cell
- ▶ **end** - aligns items to be flush with the end edge of their cell
- ▶ **center** - aligns items in the center of their cell
- ▶ **stretch** - fills the whole width of the cell (this is the default)

# CSS - GRID ALIGNMENT (FROM ITEM)

Aligns a grid item inside a cell along the block (column) axis (as opposed to justify-self which aligns along the inline (row) axis). This value applies to the content inside a single grid item.

```
.item {  
  align-self: start | end | center | stretch;  
}
```

- ▶ **start** - aligns the grid item to be flush with the start edge of the cell
- ▶ **end** - aligns the grid item to be flush with the end edge of the cell
- ▶ **center** - aligns the grid item in the center of the cell
- ▶ **stretch** - fills the whole width of the cell (this is the default)



**EXCERCISE  
TIME!**

**GRID LAYOUT**

# EXERCISE TIME - GRID LAYOUT

Try to make the following layout with grid system:

This is my lovely blog

## Other things

Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuada ultrices. Phasellus turpis est, posuere sit amet dapibus ut, facilisis sed est.

## My article

Duis felis orci, pulvinar id metus ut, rutrum luctus orci. Cras porttitor imperdiet nunc, at ultricies tellus laoreet sit amet. Sed auctor cursus massa at porta. Integer ligula ipsum, tristique sit amet orci vel, viverra egestas ligula. Curabitur vehicula tellus neque, ac ornare ex malesuada et. In vitae convallis lacus. Aliquam erat volutpat. Suspendisse ac imperdiet turpis. Aenean finibus sollicitudin eros pharetra congue. Duis ornare egestas augue ut luctus. Proin blandit quam nec lacus varius commodo et a urna. Ut id ornare felis, eget fermentum sapien.

Nam vulputate diam nec tempor bibendum. Donec luctus augue eget malesuada ultrices. Phasellus turpis est, posuere sit amet dapibus ut, facilisis sed est. Nam id risus quis ante semper consectetur eget aliquam lorem. Vivamus tristique elit dolor, sed pretium metus suscipit vel. Mauris ultricies lectus sed lobortis finibus. Vivamus eu urna eget velit cursus viverra quis vestibulum sem. Aliquam tincidunt eget purus in interdum. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Contact me@mysite.com

# CSS - SYSTEM CHOICE

- ▶ **“Vanilla”** / **Columns** **system**  
This is the “older” and the native approach to develop a layout. This approach is not really difficult but there are some troubles making some complex layout. All browsers support this approach.
- ▶ **Flexbox** **system**  
Flexbox layout is most appropriate to the components of an application, and small-scale layouts. Not even all browsers support this approach.
- ▶ **Grid** **system**  
This is the most structural approach to develop a layout. Also not all browser support this approach and the rules are changing (or not?).

Try to design and develop using the vanilla approach or some framework (such as Bootstrap!)

# CSS RECAP!

- ▶ CSS is a cascade style defining language.
- ▶ CSS relies on selectors and rules
- ▶ Selectors are used to identify a set of DOM objects.
- ▶ Rules are used to edit the behavior of these elements.
- ▶ The responsive factor is important!
- ▶ There are a lot of system to develop a layout.
- ▶ Are you a computer scientist? Yes you must hate CSS but you need it ❤.
- ▶ Now you have the basis to study the Bootstrap framework (html and css):  
<https://getbootstrap.com/docs/4.6/>

You can find other CSS rules and attributes at:  
<https://developer.mozilla.org/en-US/docs/Web/CSS>

# 4.

## JAVASCRIPT

```
alert("Hello world!");
```

# JS LANGUAGE

*Any app that can be written in JavaScript, will eventually be written in JavaScript.*

(Jeff Atwood)

*The strength of JavaScript is that you can do anything. The weakness is that you will.*

(Reg Braithwaite)

# JS WHAT IS IT?

JavaScript is a programming language that allows you to implement complex things on web pages. Every time a web page does more than just sit there and display static information for you to look at, displaying timely content updates, or interactive maps, or animated 2D/3D graphics, or scrolling video jukeboxes, etc., you can bet that JavaScript is probably involved.

It is the third layer of the layer cake of standard web technologies.

It is a **Dynamic Typed Language** (data type become known at runtime) and a **Multi-Paradigm Language** (Funcional, Imperative, Asynchronous, etc.).

In addition, JavaScript is a **Single-Threaded non-blocking Language**.

# JS WHAT IS IT?!?

JavaScript is the implementation of the ECMA Script (or ES), that is the standard technical specification of scripting language and maintained by the ECMA International.

Basically, the implementations of languages (such as Javascript or Actionscript) were designed only for the client and web development.

There was a lot of version, the most used is the sixth versions (or ES6).

# JS WHAT CAN IT DO?

- ▶ Store useful values inside variables
- ▶ Operations on pieces of text
- ▶ Running code in response to certain events occurring on a web page.
- ▶ Manage the HTML elements
- ▶ It can use the APIs!
- ▶ Execute some sync or async calls to the internet
- ▶ Draw and render 2D or 3D graphics
- ▶ Get your location
- ▶ Play videos and audios
- ▶ And much more.....

# JS HOW IT WORKS?

The JavaScript is executed by the browser's JavaScript engine, after the HTML and CSS have been assembled and put together into a web page. This ensures that the structure and style of the page are already in place by the time the JavaScript starts to run.

This is a good thing, as a very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface.

If the JavaScript loaded and tried to run before the HTML and CSS was there to affect, then errors would occur.

# JS SECURITY

Each browser tab is its own separate bucket for running code in, this means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab or on another website.

This is a good security measure if this were not the case, then pirates could start writing code to steal information from other websites, and other such bad things.

Also the web **browsers run it in sandbox mode** so you can't access directly to the host system of the user (image if you can that which you can install or delete any files from visitor's computer)

# JS WHAT YOU CAN DO WITH JS?



The image shows two examples of web-based applications:

- Left Application:** A screenshot of a Telegram-like messaging interface. It shows a list of recent conversations with users like Pavel Durov, Kristi Dirks, and Mark Watney. Each message includes a timestamp and a preview of the message content.
- Right Application:** A screenshot of a file manager titled 'Cloud Drive'. It lists various files and folders, including documents, images, and presentations. The interface includes standard file operations like 'New Folder', 'File upload', and 'Delete'.

# JS INTERPRETATION VS COMPIRATION

JavaScript is an interpreted language, the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer.

For example C/C++ are compiled into assembly language that is then run by the computer.

# JS COMPARISON WITH JAVA

## Javascript

- ▶ Object-oriented. No distinction between types of objects.  
Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.
- ▶ Variable data types are not declared (dynamic typing).
- ▶ Interpreted

## Java

- ▶ Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy.  
Classes and instances cannot have properties or methods added dynamically.
- ▶ Variable data types must be declared (static typing).
- ▶ Semi-compiled (or semi-interpreted)

# JS HOW ADD IT TO YOUR PAGE?

JavaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses `<link>` elements to apply external stylesheets and `<style>` elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML, the `<script>` element.

Internal:

1. `<script>`
2. `// JavaScript goes here`
3. `</script>`

External:

1. `<script src="script.js"></script>`

# JS WHERE ADD IT IN YOUR PAGE?

Javascript can be placed in the <head> section.

However it's better to place it in the bottom of the <body> section, because placing scripts there improves the display speed, because script compilation slows down the display.

# JS HELLO WORLD!

```
alert('Hello world!');
```

```
console.log('Hello world!');
```

# JS COMMENTS

```
// a one line comment
```

```
/* this is a longer,  
   multi-line comment  
*/
```

```
/* You can't, however, /* nest comments */ SyntaxError */
```

# JS VARIABLES

You use variables as symbolic names for values in your application. The names of variables, called **identifiers**, conform to certain rules.

A JavaScript identifier **must start with a letter, underscore (\_), or dollar sign (\$)**; subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

# JS VARIABLES - DECLARE

You can declare a variable in three ways:

- ▶ With the keyword **var**. For example, `var x = 42`. This syntax can be used to declare both local and global variables.
- ▶ By simply assigning it a value. For example, `x = 42`. This always declares a global variable, if it is declared outside of any function. It generates a strict JavaScript warning. You shouldn't use this variant.
- ▶ With the keyword **let**. For example, `let y = 13`. This syntax can be used to declare a block-scope local variable.
- ▶ With the keyword **const**. For example, `const y = 13`. This syntax can be used to declare a block-scope local constant.

# JS VARIABLES - UNDEFINED

A variable declared using the var or let statement with no assigned value specified has the value of **undefined**.

```
var a;  
console.log('The value of a is ' + a); // The value of a is undefined  
  
console.log('The value of b is ' + b); // The value of b is undefined  
var b;  
  
console.log('The value of c is ' + c); // Uncaught ReferenceError: c is not defined  
  
let x;  
console.log('The value of x is ' + x); // The value of x is undefined  
  
console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not defined  
let y;
```

# JS PRIMITIVE DATA TYPES

//string

```
var greeting = 'Hello Kitty';
```

```
var restaurant = "Pamela's Place";
```

//number

```
var myAge = 28;
```

```
var pi = 3.14;
```

//boolean

```
var catsAreBest = true;
```

```
var dogsRule = false;
```

//undefined

```
var notDefinedYet;
```

//null

```
var goodPickupLines = null;
```

//object and functions!

# JS STRINGS

A string holds an ordered list of characters:

```
var alphabet = "abcdefghijklmnopqrstuvwxyz";
```

The length property reports the size of the string:

```
console.log(alphabet.length); // 26
```

Each character has an index. The first character is always at index 0. The last character is always at index length-1:

```
console.log(alphabet[0]); // 'a'
```

```
console.log(alphabet[1]); // 'b'
```

```
console.log(alphabet[2]); // 'c'
```

```
console.log(alphabet[alphabet.length]); // undefined
```

```
console.log(alphabet[alphabet.length-1]); // 'z'
```

```
console.log(alphabet[alphabet.length-2]); // 'y'
```

# JS EXPRESSIONS

Variables can also store the result of any "expression"

```
var x = 2 + 2;
```

```
var y = x * 3;
```

```
var name = 'Claire';
```

```
var greeting = 'Hello ' + name;
```

```
var title = 'Baroness';
```

```
var formalGreeting = greeting + ', ' + title;
```

# JS DATA TYPE CONVERSION

JavaScript is a dynamically typed language. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution.

```
var x;  
x = 2;  
x = 'Hi';  
console.log(typeof(x));
```

# JS IF STATEMENT

The if statement executes a statement if a specified condition is truthy. If the condition is false, another statement can be executed.

```
if (condition) {  
    statements1  
} else {  
    statements2  
}
```

# JS SWITCH STATEMENT

The switch statement evaluates an expression, matching the expression's value to a case clause, and executes statements associated with that case.

```
switch (expression) {  
    case value1:  
        //Statements executed when the  
        //result of expression matches value1  
        [break;]  
    case value2:  
        //Statements executed when the  
        //result of expression matches value2  
        [break;]  
    ...  
    [default:  
        //Statements executed when none of  
        //the values match the value of the expression  
        [break;]]  
}
```

# JS CONDITIONALS

The construct if ( Expression ) Statement will coerce the result of evaluating the Expression to a boolean using the abstract method ToBoolean for which the ES5 spec defines the following algorithm:

Argument Type	Result
Undefined	false
Null	false
Boolean	The result equals the input argument (no conversion).
Number	The result is false if the argument is +0, -0, or NaN; otherwise the result is true.
String	The result is false if the argument is the empty String (its length is zero); otherwise the result is true.
Object	true

# JS COMPARISON OPERATORS

Comparison operators are used in logical statements to determine equality or difference between variables or values. Given that  $x = 5$ , the table below explains the comparison operators:

Operator	Description
<code>==</code>	equal to
<code>===</code>	equal value and equal type
<code>!=</code>	not equal
<code>!==</code>	not equal value or not equal type
<code>&gt;</code>	greater than
<code>&lt;</code>	less than
<code>&gt;=</code>	greater than or equal to
<code>&lt;=</code>	less than or equal to

# JS COMPARISON TABLE

	true	false	"false"	"true"	-1	0	1	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
false	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
1	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
0	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
-1	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
"true"	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
"false"	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
"1"	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
"0"	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
"-1"	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
""	white	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
null	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
undefined	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
Infinity	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
-Infinity	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
[]	green	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
{}	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
[[]]	green	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
[0]	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
[1]	green	green	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white
NaN	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white	white



# JS TERNARY OPERATOR

The conditional (ternary) operator is the only JavaScript operator that takes three operands. This operator is frequently used as a shortcut for the if statement.

*condition ? expr1 : expr2*

```
var elvisLives = Math.PI > 4 ? 'Yep' : 'Nope';  
  
'The fee is ' + (isMember ? '$2.00' : '$10.00');
```

# JS WHILE STATEMENT

The while statement creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

```
var n = 0;  
var x = 0;  
  
while (n < 3) {  
    n++;  
    x += n;  
}  
}
```

# JS FOR STATEMENT

The for statement creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement (usually a block statement) to be executed in the loop.

```
for ([initialization]; [condition]; [final-expression])  
  statement  
  
for (var i = 0; i < 9; i++) {  
  console.log(i);  
  // more statements  
}
```

To prematurely exit a loop, use the **break** statement

# JS FOR STATEMENT

The for statement creates a loop that iterates all objects which are inside an iterable object.

```
for(var index in arr) {  
    console.log(arr[index]);  
    // more statements  
}  
  
for(var elem of arr) {  
    console.log(elem);  
    // more statements  
}
```

# JS ARRAYS

The JavaScript Array object is a global object that is used in the construction of arrays; which are high-level, list-like objects.

```
var fruits = ["Apple", "Banana"];
console.log(fruits.length);
// 2

var first = fruits[0];
// Apple

var last = fruits[fruits.length - 1];
// Banana

fruits.forEach(function (item, index, array) {
  console.log(item, index);
});
// Apple 0
// Banana 1
```

# JS ARRAYS

```
var newLength = fruits.push("Orange");
// ["Apple", "Banana", "Orange"]

var last = fruits.pop(); // remove Orange (from the end)
// ["Apple", "Banana"];

var first = fruits.shift(); // remove Apple from the front
// ["Banana"];

var newLength = fruits.unshift("Strawberry") // add to the front
// ["Strawberry", "Banana"];

fruits.push("Mango");
// ["Strawberry", "Banana", "Mango"]

var pos = fruits.indexOf("Banana");
// 1

var removedItem = fruits.splice(pos, 1); // this is how to remove an item
// ["Strawberry", "Mango"]

var shallowCopy = fruits.slice(); // this is how to make a copy
// ["Strawberry", "Mango"]
```

# JS FUNCTIONS

A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:

- ▶ The name of the function.
- ▶ A list of parameters to the function, enclosed in parentheses and separated by commas.
- ▶ The JavaScript statements that define the function, enclosed in curly brackets, {}.

```
function square(number) {  
    return number * number;  
}
```

# JS FUNCTIONS

While the function declaration above is syntactically a statement, functions can also be created by a function expression. Such a function can be anonymous; it does not have to have a name. For example, the function square could have been defined as:

```
var square = function(number) { return number * number; };
var x = square(4); // x gets the value 16
```

However, a name can be provided with a function expression and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1); };
console.log(factorial(3));
```

# JS FUNCTIONS

Function expressions are convenient when passing a function as an argument to another function. The following example shows a map function that should receive a function as first argument and an array as second argument.

```
function map(f, a) {  
    var result = []; // Create a new Array  
    var i; // Declare variable  
    for (i = 0; i != a.length; i++)  
        result[i] = f(a[i]);  
    return result;  
}  
  
var f = function(x) {  
    return x * x * x;  
}  
  
var numbers = [0,1, 2, 5,10];  
var cube = map(f,numbers);  
console.log(cube);
```

# JS ARROW FUNCTIONS

An **arrow function** expression is a syntactically compact alternative to a regular function expression, although without its own bindings to the `this`, `arguments`, `super`, or `new.target` keywords. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

```
(param1, param2, ..., paramN) => { statements }  
(param1, param2, ..., paramN) => expression  
// equivalent to: => { return expression; }  
  
// Parentheses are optional when there's only one parameter name:  
(singleParam) => { statements }  
singleParam => { statements }  
  
// The parameter list for a function with no parameters should be written  
with a pair of parentheses.  
() => { statements }
```

Two factors influenced the introduction of arrow functions: the need for shorter functions and the behavior of the **this** keyword.

# JS ARROW FUNCTIONS

Before arrow functions, every new function defined its own **this** value based on how the function was called:

- ▶ A new object in the case of a constructor.
- ▶ undefined in strict mode function calls.
- ▶ The base object if the function was called as an "object method".
- ▶ etc.

Alternatively, a bound function could be created so that a preassigned this value would be passed to the bound target function.

An arrow function does not have its own **this**. The **this** value of the enclosing lexical scope is used; arrow functions follow the normal variable lookup rules. So while searching for **this** which is not present in current scope, an arrow function ends up finding the **this** from its enclosing scope.

# JS HOISTING

**Hoisting** was thought up as a general way of thinking about how execution contexts (specifically the creation and execution phases) work in JavaScript. A strict definition of hoisting suggests that **variable and function declarations are physically moved to the top of your code**, but this is not in fact what happens. Instead, the variable and function declarations are put into memory but **stay exactly where you typed them in your code**.

```
function studentName(name) {  
    console.log("My student's name is " + name);  
}  
  
studentName("Tiger"); /* The result of the code above is: "My student's name is Tiger" */
```

The above code snippet is how you would expect to write the code for it to work. Now, let's see what happens when we call the function before we write it:

```
studentName("Tiger");  
  
function studentName(name) {  
    console.log("My student's name is " + name);  
}  
  
/* The result of the code above is: "My student's name is Tiger" */
```

# JS HOISTING

JavaScript only hoists declarations, not initializations. If a variable is declared and initialized after using it, the value will be undefined.

```
console.log(num); // Returns undefined, as only declaration was hoisted, no initialization has happened at this stage
var num; // declaration
num = 6; // initialization
```

The example below only has initialization. No hoisting happens so trying to read the variable results in ReferenceError exception.

```
console.log(num); // throws ReferenceError exception
num = 6; // initialization
```

# JS SCOPE

JS Variables have "function scope". They are visible in the function where they're defined:

```
function addNumbers(num1, num2) {  
  var localResult = num1 + num2;  
  
  console.log("The local result is: " + localResult);  
}  
  
addNumbers(5, 7);  
  
console.log(localResult);
```

# JS CLOSURE

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to).

```
var pet = function(name) {    // The outer function defines a variable called "name"
  var getName = function() {
    return name;                // The inner function has access to the "name" variable of the outer function
  }
  return getName;              // Return the inner function, thereby exposing it to outer scopes
}
myPet = pet('Vivie');

myPet();                      // Returns "Vivie"
```

# JS OBJECTS

An object is a collection of related data and/or functionality. As with many things in JavaScript, creating an object often begins with defining and initializing a variable.

```
var obj= { [key]:[value][,....]};

var person = {
    name: [Federico, Fausto],
    age: 28,
    gender: 'male',
    interests: ['nintendo', 'cloud'],
    bio: function() {
        alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' +
        this.interests[0] + ' and ' + this.interests[1] + '.');
    },
    greeting: function() {
        alert('Hi! I\'m ' + this.name[0] + '.');
    }
};
```

# JS OBJECTS

Javascript relies on the prototypal inheritance. In Javascript everything is an Object and each Object holds a “link” to its prototype and this creates a prototype chain where objects can inherit behaviors from other objects.

This differs from classical inheritance where you define a class or blueprint for each object and instantiate it.

This can sounds weird but it is one of the low-level concepts that makes JavaScript a very flexible multi-paradigm language.

```
const Animal = {race : "Human"};  
  
const Person = Object.create(Animal);  
  
console.log(Person.race); //human
```

# JS OBJECTS - DOT NOTATION

Above, you accessed the object's properties and methods using dot notation. The object name (person) acts as the namespace – it must be entered first to access anything encapsulated inside the object. Next you write a dot, then the item you want to access – this can be the name of a simple property, an item of an array property, or a call to one of the object's methods

```
person.age
```

```
person.interests[1]
```

```
person.bio()
```

# JS OBJECTS - BRACKET NOTATION

This looks very similar to how you access the items in an array, and it is basically the same thing – instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called associative arrays – they map strings to values in the same way that arrays map numbers to values.

```
person['age']  
person['name']['first']
```

# JS OBJECTS - THIS

The `this` keyword refers to the current object the code is being written inside so in this case `this` is equivalent to `person`. When we start creating constructors, etc., this is very useful it will always ensure that the correct values are used when a member's context changes (e.g. two different `person` object instances may have different names, but will want to use their own name when saying their greeting).

```
var person1 = {
  name: 'Chris',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}

var person2 = {
  name: 'Brian',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}
```

# JS OBJECTS - CLASSES

JavaScript classes are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

```
class Student {  
    name;  
    surname;  
    #gender; //private  
    constructor(name, surname) {  
        this.name = name;  
        this.surname = surname;  
    }  
}
```

An important difference between function declarations and class declarations is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError.

# JS DOM

The Document Object Model (DOM) is a programming interface for HTML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page.

For example, the standard DOM specifies that the getElementsByName method in the code below must return a list of all the <P> elements in the document:

```
var paragraphs = document.getElementsByTagName("P");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

# ACCESS THE DOM

Different browsers have different implementations of the DOM, and these implementations exhibit varying degrees of conformance to the actual DOM standard, but every web browser uses some document object model to make web pages accessible via JavaScript.

When you create a script—whether it's inline in a <script> element or included in the web page by means of a script loading instruction—you can immediately begin using the API for the document or window elements to manipulate the document itself or to get at the children of that document, which are the various elements in the web page.

This following JavaScript will display an alert when the document is loaded (and when the whole DOM is available for use):

```
<body onload="window.alert('Page loaded!');">
```

# ACCESS THE DOM

Another example. This function creates a new **H1** element, adds text to that element, and then adds the H1 to the tree for this document:

```
// run this function when the document is loaded
window.onload = function() {
    // create a couple of elements in an otherwise empty HTML page
    var heading = document.createElement("h1");
    var heading_text = document.createTextNode("Big Head!");
    heading.appendChild(heading_text);
    document.body.appendChild(heading);
}
```

# DOM INTERFACES

Document and window objects are the objects whose interfaces you generally use most often in DOM programming.

In simple terms, the window object represents something like the browser, and the document object is the root of the document itself.

Element inherits from the generic Node interface, and together these two interfaces provide many of the methods and properties you use on individual elements.

These elements may also have specific interfaces for dealing with the kind of data those elements hold.

# DOM INTERFACES

- `document.getElementById(id)`
- `document.getElementsByTagName(name)`
- `document.getElementsByClassName(className)`
- `document.createElement(name)`
- `document.querySelector(selector)`
- `parentNode.appendChild(node)`
- `element.innerHTML`
- `element.style.left`
- `element.setAttribute()`
- `element.getAttribute()`
- `element.addEventListener()`
- `window.content`
- `window.onload`
- `console.log()`
- `window.scrollTo()`

# JS EVENTS

JavaScript is a Single-Threaded language which means that JavaScript can only run one instruction at a time, even if your CPU has multiple cores and threads.

Then, the most popular question is “How does JavaScript handle jobs at the same time?”. JavaScript has a concurrency model based on an event loop.

When a JavaScript code is executed, two regions of memory are allocated on the machine, the call stack (a high performance memory) and the heap.

The Stack is used to execute the functions and saves a frame copy of the functions and a copy of its local variables.

The Heap is used in a more complex situation. It is an unstructured memory pool where will be stored things like objects or primitive values inside of closures. It is Garbage Collected, then you do not manually manage the memory allocation or free memory up.

# JS EVENTS

A JavaScript runtime uses a message **queue**, which is a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.

The runtime starts handling the messages on the queue, starting with the oldest one. The message is removed from the queue and its corresponding function is called with the message as an input parameter.

As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

**Event Loop** refers to a feature implemented by engines that allow JavaScript to offload tasks to separate threads. Browser and Node APIs execute long-running tasks separately from the the main JavaScript thread, then enqueue a callback function to run on the main thread when the task is complete.

# JS EVENTS

Think of the Event Loop as message queue between the single JavaScript thread and the OS.

```
while (queue.waitForMessage()) {  
    queue.processNextMessage();  
}
```

`queue.waitForMessage()` waits synchronously for a message to arrive (if one is not already available and waiting to be handled).

Each message is processed completely before any other message is processed.

If a message takes too long to complete, the application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog.

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks. Handling I/O is typically performed via events and callbacks, so when the application is waiting for an IndexedDB query to return or an XHR request to return, it can still process other things like user input.

Legacy exceptions exist like alert or synchronous XHR, **but it is considered a good practice to avoid them.**

# DOM EVENTS

There are 3 ways to register event handlers for a DOM element.

Using the addEventListener function that sets up a function that will be called whenever the specified event is delivered to the target.  
Internet Explorer 6-8 didn't support this method

```
// Assuming myButton is a button element
myButton.addEventListener('click', greet);

function greet(event){
    // print and have a look at the event object
    // always print arguments in case of overlooking any other arguments
    console.log('greet:', arguments);
    alert('hello world');

}
```

# DOM EVENTS

Using the attribute. The JavaScript code in the attribute is passed the Event object via the event parameter. This way should be avoided. This makes the markup bigger and less readable.

Concerns of content/structure and behavior are not well-separated, making a bug harder to find

```
<button onclick="alert('Hello world!')">
```

# DOM EVENTS

Using the element event function. The problem with this method is that only one handler can be set per element and per event.

```
// Assuming myButton is a button element  
myButton.onclick = function(event){alert('Hello world');};
```

# JS ASYNCHRONOUS

To allow us to understand what asynchronous JavaScript is, we ought to start off by making sure we understand what synchronous JavaScript is.

A lot of the functionality we have looked at in previous learning area modules is synchronous you run some code, and the result is returned as soon as the browser can do so.

While each operation is being processed, nothing else can happen/rendering is paused. This is because as we said in the previous article, JavaScript is single threaded. Only one thing can happen at a time, on a single main thread, and everything else is blocked until an operation completes.

For these reasons, many Web API features now use asynchronous code to run, especially those that access or fetch some kind of resource from an external device, such as fetching a file from the network, accessing a database and returning data from it, accessing a video stream from a webcam, or broadcasting the display to a VR headset.

There are two main types of asynchronous code style you will come across in JavaScript code, old-style callbacks, the promise-style code and the newer async/await style.

# JS ASYNCHRONOUS CALLBACKS

Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background. When the background code finishes running, it calls the callback function to let you know the work is done, or to let you know that something of interest has happened.

An example of an async callback is the second parameter of the `addEventListener()` method.

```
btn.addEventListener('click', () => {
  alert('You clicked me!');
  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

The first parameter is the type of event to be listened for, and the second parameter is a callback function that is invoked when the event is fired.

When we pass a callback function as an argument to another function, we are only passing the function's reference as an argument, i.e. the callback function is not executed immediately.

It is “called back” (hence the name) asynchronously somewhere inside the containing function’s body. The containing function is responsible for executing the callback function when the time comes.

# JS ASYNCHRONOUS PROMISES

Promises are the new style of async code that you'll see used in modern Web APIs. This concept can take practice to get used to; it feels a little like Schrödinger's cat in action.

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

- ▶ **pending**: initial state, neither fulfilled nor rejected.
- ▶ **fulfilled**: meaning that the operation was completed successfully.
- ▶ **rejected**: meaning that the operation failed.

A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's **then** method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.

The methods `promise.then()`, `promise.catch()`, and `promise.finally()` are used to associate further action with a promise that becomes settled.

# JS ASYNCHRONOUS PROMISES

The `.then()` method takes up to two arguments; the first argument is a callback function for the resolved case of the promise, and the second argument is a callback function for the rejected case. Each `.then()` returns a newly generated promise object, which can optionally be used for chaining.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});
myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

Processing continues to the next link of the chain even when a `.then()` lacks a callback function that returns a Promise object. Therefore, a chain can safely omit every rejection callback function until the final `.catch()`.

# JS ASYNCHRONOUS PROMISES

Handling a rejected promise in each `.then()` has consequences further down the promise chain.  
Sometimes there is no choice, because an error must be handled immediately.

In such cases we must throw an error of some type to maintain error state down the chain. On the other hand, in the absence of an immediate need, it is simpler to leave out error handling until a final `.catch()` statement. A `.catch()` is really just a `.then()` without a slot for a callback function for the case when the promise is resolved.

```
myPromise  
  .then(handleResolvedA)  
  .then(handleResolvedB)  
  .then(handleResolvedC)  
  .catch(handleRejectedAny);
```

The termination condition of a promise determines the "settled" state of the next promise in the chain. A "resolved" state indicates a successful completion of the promise, while a "rejected" state indicates a lack of success. The return value of each resolved promise in the chain is passed along to the next `.then()`, while the reason for rejection is passed along to the next rejection-handler function in the chain.

The promises of a chain are nested like Russian dolls, but get popped like the top of a stack. The first promise in the chain is most deeply nested and is the first to pop.

# JS ASYNC/AWAIT

More recent additions to the JavaScript language are **async functions** and the **await** keyword. These features basically act as syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards.

They make async code look more like old-school synchronous code, so they are well worth learning. This article gives you what you need to know.

First of all we have the **async** keyword, which you put in front of a function declaration to turn it into an **async function**. An **async function** is a function that knows how to expect the possibility of the **await** keyword being used to invoke asynchronous code.

```
async function hello() { return "Hello" };
hello();
//or
let hello = async function() { return "Hello" };
hello();
```

Invoking the function now returns a promise. This is one of the traits of async functions, their return values are guaranteed to be converted to promises.

# JS ASYNC/AWAIT

The advantage of an async function only becomes apparent when you combine it with the **await** keyword. await only works inside async functions within regular JavaScript code, however it can be used on its own with JavaScript modules.

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

You can use **await** when calling any function that returns a Promise, including web API functions and inside of another async function.

```
async function hello() {  
    return greeting = await Promise.resolve("Hello");  
};  
  
hello().then(alert);
```

And if you want to add error handling, you can use a synchronous try/catch structure with **async/await**.

# HTTP REQUESTS

When you are surfing the web and navigating between web pages, what you are really doing is telling your browser to request information from any of these servers.

It kinda looks as follows: your browser sends a request, waits awkwardly for the server to respond to the request, and (once the server responds) processes the request.

All of this communication is made possible because of something known as the **HTTP protocol**.

Normally, this approach is synchronous!!

# JS AJAX

**Ajax** (Asynchronous JavaScript and XML) is the traditional way to make an asynchronous HTTP request. Data can be sent using the HTTP POST method and received using the HTTP GET method.

```
const http = new XMLHttpRequest();
const url = 'https://jsonplaceholder.typicode.com/todos/';
http.open("GET", url);

//define a functions that will be executed when a request is completed
http.onreadystatechange = function(e){
    if (this.readyState == 4 && this.status == 200) {
        console.log(http.responseText)
    }
}

//send the request
http.send();
```

# JS AJAX

To make an HTTP call in Ajax, you need to initialize a new `XMLHttpRequest()` method, specify the URL endpoint and HTTP method (in this case GET).

Finally, we use the `open()` method to tie the HTTP method and URL endpoint together and call the `send()` method to fire off the request.

We log the HTTP response to the console by using the `XMLHttpRequest.onreadystatechange` property which contains the event handler to be called when the `readystatechange` event is fired.

The `onreadystatechange` property has two values, `readyState` and `status` which allow us to check the state of our request.

If `readyState` is equal to 4, it means the request is done. The `readyState` property has 5 responses.

# JS FETCH

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses.

It also provides a global **fetch()** method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using **XMLHttpRequest**. Fetch provides a better alternative that can be easily used by other technologies such as Service Workers.

Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP.

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

The simplest use of `fetch()` takes one argument, the path to the resource you want to fetch, and returns a promise containing the response (a Response object).

# JS FETCH

```
// Example POST method implementation:  
const data = { username: 'example' };  
  
fetch('https://example.com/profile', {  
  method: 'POST', // or 'PUT'  
  headers: {  
    'Content-Type': 'application/json',  
  },  
  body: JSON.stringify(data),  
})  
.then(response => response.json())  
.then(data => {  
  console.log('Success:', data);  
})  
.catch((error) => {  
  console.error('Error:', error);  
});
```

# JS FETCH

The `fetch` specification differs from other async http calls in the following significant ways:

- ▶ The Promise returned from `fetch()` won't reject on HTTP error status even if the response is an HTTP 404 or 500. Instead, as soon as the server responds with headers, the Promise will resolve normally (with the `ok` property of the response set to false if the response is not in the range 200-299), and it will only reject on network failure or if anything prevented the request from completing.
- ▶ `fetch()` won't send cross-origin cookies unless you set the `credentials` init option. The spec changed the default credentials policy to same-origin.

# JS RECAP!

- ▶ JS is a dynamic multi-paradigm programming language.
- ▶ JS is executed by the browser.
- ▶ Also, you can use it on server side!
- ▶ It Allows the manipulation of the DOM.
- ▶ Ok, ok, it is crazy...
- ▶ ...But also powerful!
- ▶ It relies on the async programming.
- ▶ You can develop any thing with JavaScript!

# 5. JQUERY

`$() = JQuery();`

# WHAT IS JQUERY

jQuery is a JavaScript Library that focuses on simplifying DOM manipulation, AJAX calls, and Event handling. It is used by JavaScript developers frequently.

jQuery uses a format, `$(selector).action()` to assign an element(s) to an event. To explain it in detail, `$(selector)` will call jQuery to select selector element(s), and assign it to an event API called `.action()`.

One important thing to know is that jQuery is just a JavaScript library. All the power of jQuery is accessed via JavaScript, so having a strong grasp of JavaScript is essential for understanding, structuring, and debugging your code.

While working with jQuery regularly can, over time, improve your proficiency with JavaScript, it can be hard to get started writing jQuery without a working knowledge of JavaScript's built-in constructs and syntax.

# HOW JQUERY WORKS

To ensure that their code runs after the browser finishes loading the document, many JavaScript programmers wrap their code in an `onload` function:

```
window.onload = function() {  
    alert( "welcome" );  
};
```

Unfortunately, the code doesn't run until all images are finished downloading. To run code as soon as the document is ready to be manipulated, jQuery has a statement known as the `ready` event:

```
$( document ).ready(function() {  
    // Your code here.  
});
```

The jQuery library exposes its methods and properties via two properties of the `window` object called [jQuery](#) and [\\$](#).

[\\$](#) is simply an alias for [jQuery](#) and it's often employed because it's shorter and faster to write.

# JQUERY SELECTORS

The most basic concept of jQuery is to "select some elements and do something with them." jQuery supports most CSS3 selectors, as well as some non-standard selectors

```
$( "#myId" ); // Note IDs must be unique per page.  
$( ".myClass" );  
$( "input[name='first_name']" );  
$( "#contents ul.people li" );  
$( "div.myClass, ul.people" );  
  
$( "a.external:first" );  
  
$( "tr:odd" );  
  
// Select all input-like elements in a form (more on this below).  
$( "#myForm :input" );  
$( "div:visible" );  
  
// All except the first three divs,  
$( "div:gt(2)" );
```

# JQUERY SELECTORS

Query doesn't cache elements for you. If you've made a selection that you might need to make again, you should save the selection in a variable rather than making the selection repeatedly.

```
var divs = $("div");
```

Once the selection is stored in a variable, you can call jQuery methods on the variable just like you would have called them on the original selection.

A selection only fetches the elements that are on the page at the time the selection is made. If elements are added to the page later, you'll have to repeat the selection or otherwise add them to the selection stored in the variable. Stored selections don't magically update when the DOM changes.

# REFINING & FILTERING SELECTORS

Sometimes the selection contains more than what you're after.

jQuery offers several methods for refining and filtering selections.

```
// Refining selections.  
$( "div.foo" ).has( "p" ); // div.foo elements that contain <p> tags  
$( "h1" ).not( ".bar" ); // h1 elements that don't have a class of bar  
$( "ul li" ).filter( ".current" ); // unordered list items with class of current  
$( "ul li" ).first(); // just the first unordered list item  
$( "ul li" ).eq( 5 ); // the sixth
```

# MANIPULATING ELEMENTS

There are many ways to change an existing element. Among the most common tasks is changing the inner HTML or attribute of an element. jQuery offers simple, cross-browser methods for these sorts of manipulations. You can also get information about elements using many of the same methods in their getter incarnations.

**.html()** - Get or set the HTML contents.

**.text()** - Get or set the text contents; HTML will be stripped.

**.attr()** - Get or set the value of the provided attribute.

**.width()** - Get or set the width in pixels of the first element in the selection as an integer.

**.height()** - Get or set the height in pixels of the first element in the selection as an integer.

**.position()** - Get an object with position information for the first element in the selection, relative to its first positioned ancestor. *This is a getter only.*

**.val()** - Get or set the value of form elements.

*// Changing the HTML of an element.*

```
$( "#myDiv p:first" ).html( "New <strong>first</strong> paragraph!" );
```

# MANIPULATING ELEMENTS

While there are a variety of ways to move elements around the DOM, there are generally two approaches: place the selected element(s) relative to another element or place an element relative to the selected element(s).

jQuery provides `.insertAfter()` and `.after()`. The `.insertAfter()` method places the selected element(s) after the element provided as an argument. The `.after()` method places the element provided as an argument after the selected element. Several other methods follow this pattern: `.insertBefore()` and `.before()`, `.appendTo()` and `.append()`, and `.prependTo()` and `.prepend()`.

```
// Moving elements using different approaches.  
// Make the first list item the last list item:  
var li = $( "#myList li:first" ).appendTo( "#myList" );  
// Another approach to the same problem:  
$( "#myList" ).append( $( "#myList li:first" ) );  
// Note that there's no way to access the list item  
// that we moved, as this returns the list itself.
```

# MANIPULATING ELEMENTS

There are two ways to remove elements from the page: `.remove()` and `.detach()`. Use `.remove()` when you want to permanently remove the selection from the page. While `.remove()` does return the removed element(s), those elements will not have their associated data and events attached to them if you return them to the page.

Use `.detach()` if you need the data and events to persist. Like `.remove()`, it returns the selection, but it also maintains the data and events associated with the selection, so you can restore the selection to the page at a later time.

The `.detach()` method is extremely valuable if you are doing heavy manipulation on an element. In that case, it's beneficial to `.detach()` the element from the page, work on it in your code, then restore it to the page when you're done. This limits expensive "DOM touches" while maintaining the element's data and events.

# CREATING ELEMENTS

jQuery offers a trivial and elegant way to create new elements using the same `$()` method used to make selections:

```
// Creating new elements from an HTML string.  
$( "<p>This is a new paragraph</p>" );  
$( "<li class=\"new\">new list item</li>" );  
// Creating a new element with an attribute object.  
$( "<a/>" , {  
    html: "This is a <strong>new</strong> link",  
    "class": "new",  
    href: "foo.html"  
});
```

# CREATING ELEMENTS

The syntax for adding new elements to the page is easy, so it is tempting to forget that there is a huge performance cost for adding to the DOM repeatedly.

If you're adding many elements to the same container, you will want to concatenate all the HTML into a single string, and then append that string to the container instead of appending the elements one at a time.

Use an array to gather all the pieces together, then join them into a single string for appending:

```
var myItems = [];
var myList = $( "#myList" );
for ( var i = 0; i < 100; i++ ) {
    myItems.push( "<li>item " + i + "</li>" );
}
myList.append( myItems.join( "" ) );
```

# MANIPULATING ATTRIBUTES

jQuery's attribute manipulation capabilities are extensive. Basic changes are simple, but the `.attr()` method also allows for more complex manipulations. It can either set an explicit value, or set a value using the return value of a function.

When the function syntax is used, the function receives two arguments: the zero-based index of the element whose attribute is being changed, and the current value of the attribute being changed.

```
// Manipulating a single attribute.  
$( "#myDiv a:first" ).attr( "href", "newDestination.html" );  
  
// Manipulating multiple attributes.  
$( "#myDiv a:first" ).attr({  
    href: "newDestination.html",  
    rel: "nofollow"  
});
```

# MANIPULATING DATA ATTRIBUTES

There's often data about an element you want to store with the element. jQuery offers a straightforward way to store data related to an element, and it manages the memory issues for you.

In addition to passing `.data()` a single key-value pair to store data, you can also pass an object containing one or more pairs.

```
// Storing and retrieving data related to an element.  
$( "#myDiv" ).data( "keyName", { foo: "bar" } );  
$( "#myDiv" ).data( "keyName" ); // Returns { foo: "bar" }
```

# DOM TRAVERSING

Traversing can be broken down into three basic parts: parents, children, and siblings. jQuery has an abundance of easy-to-use methods for all these parts. Notice that each of these methods can optionally be passed string selectors, and some can also take another jQuery object in order to filter your selection down.

## Parents:

```
// Selecting an element's direct parent:  
$( "span.subchild" ).parent();  
  
// Selecting all the parents of an element that match a given selector:  
$( "span.subchild" ).parents( "div.parent" );  
$( "span.subchild" ).parents();  
  
// Selecting all the parents of an element up to, but *not including* the selector:  
$( "span.subchild" ).parentsUntil( "div.grandparent" );  
  
// Selecting the closest parent, note that only one parent will be selected  
// and that the initial element itself is included in the search:  
$( "span.subchild" ).closest( "div" );  
  
// returns [ div.child ] as the selector is also included in the search:  
$( "div.child" ).closest( "div" );
```

# DOM TRAVERSING

The methods for finding child elements from a selection include `.children()` and `.find()`. The difference between these methods lies in how far into the child structure the selection is made. `.children()` only operates on direct child nodes, while `.find()` can traverse recursively into children, children of those children, and so on.

```
// Selecting an element's direct children:  
$( "div.grandparent" ).children( "div" );  
// Finding all elements within a selection that match the selector:  
$( "div.grandparent" ).find( "div" );
```

# ASYNC. JS & XML **AJAX**

The XMLHttpRequest object is part of a technology called Ajax (Asynchronous JavaScript and XML). Using Ajax, data could then be passed between the browser and the server, using the XMLHttpRequest API, without having to reload the web page.

Ajax requests are triggered by JavaScript code; your code sends a request to a URL, and when it receives a response, a callback function can be triggered to handle the response.

Because the request is asynchronous, the rest of your code continues to execute while the request is being processed, so it's imperative that a callback be used to handle the response.

Unfortunately, different browsers implement the Ajax API differently. Typically this meant that developers would have to account for all the different browsers to ensure that Ajax would work universally.

# ASYNC. JS & XML AJAX

The asynchronicity of Ajax catches many new jQuery users off guard. Because Ajax calls are asynchronous by default, the response is not immediately available. Responses can only be handled using a callback.

So, for example, the following code will not work:

```
var response;
$.get( "request.php", function( r ) {
    response = r;
});
console.log( response ); // undefined
```

Instead, we need to pass a callback function to our request; this callback will run when the request succeeds, at which point we can access the data that it returned, if any.

```
$.get( "request.php", function( response ) {
    console.log( response ); // server response
});
```

# JQUERY AJAX

jQuery's core `$.ajax()` method is a powerful and straightforward way of creating Ajax requests. It takes a configuration object that contains all the instructions jQuery requires to complete the request.

The `$.ajax()` method is particularly valuable because it offers the ability to specify both success and failure callbacks. Also, its ability to take a configuration object that can be defined separately makes it easier to write reusable code.

# JQUERY AJAX

```
// Using the core $.ajax() method
$.ajax({
  // The URL for the request
  url: "post.php",
  // The data to send (will be converted to a query string)
  data: {
    id: 123
  },
  // Whether this is a POST or GET request
  type: "GET",
  // The type of data we expect back
  dataType : "json",
})
// Code to run if the request succeeds (is done);
// The response is passed to the function
.done(function( json ) {
  $( "<h1>" ).text( json.title ).appendTo( "body" );
  $( "<div class=\"content\">" ).html( json.html ).appendTo( "body" );
})
// Code to run if the request fails; the raw request and
// status codes are passed to the function
.fail(function( xhr, status, errorThrown ) {
  alert( "Sorry, there was a problem!" );
  console.log( "Error: " + errorThrown );
  console.log( "Status: " + status );
})
// Code to run regardless of success or failure;
.always(function( xhr, status ) {
  alert( "The request is complete!" );
});
```

# JQUERY AJAX METHODS

The Ajax convenience functions provided by jQuery can be useful, terse ways to accomplish Ajax requests. These methods are just "wrappers" around the core `$.ajax()` method, and simply pre-set some of the options on the `$.ajax()` method.

```
// Using jQuery's Ajax convenience methods
// Get plain text or HTML (or POST)
$.get( "/users.php", {
    userId: 1234
}, function( resp ) {
    console.log( resp ); // server response
});
// Add a script to the page, then run a function defined in it
$.getScript( "/static/js/myScript.js", function() {
    functionFromMyScript();
});
// Get JSON-formatted data from the server
$.getJSON( "/details.php", function( resp ) {
    // Log each key in the response data
    $.each( resp, function( key, value ) {
        console.log( key + " : " + value );
    });
});
```

# JQUERY RECAP!

- ▶ JQuery is a JS library.
- ▶ It helps to “discover” DOMs
- ▶ Also, it implements its DOM object, that is JQuery Element
- ▶ JQuery Element has a lot of useful attributes and methods
- ▶ Also, the event management is more easy than the normal JS way
- ▶ It is an old library that is used today by a lot of plugins and libraries

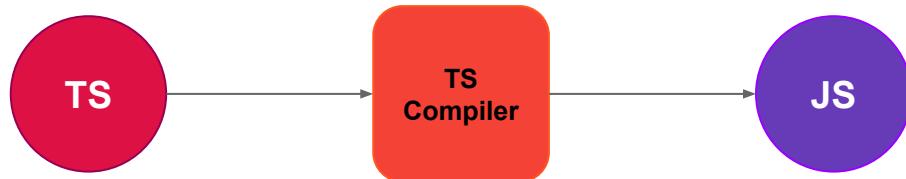
# 6.

## TYPESCRIPT

```
const name: type;
```

# INTRODUCTION TO TYPESCRIPT

TypeScript **is a superset** of JavaScript. It was designed by Anders Hejlsberg (designer of C#) at Microsoft. TypeScript **builds on top of JavaScript**. First, you write the TypeScript code. Then, you compile the TypeScript code into plain JavaScript code using a TypeScript compiler. Once you have the plain JavaScript code, you can deploy it to any environments that JavaScript runs.



TypeScript uses the JavaScript syntaxes and adds additional syntaxes for supporting Types. If you have a JavaScript program that does not have any syntax errors, it is also a TypeScript program. It means that all JavaScript programs are TypeScript programs.



# INTRODUCTION TO TYPESCRIPT

- ▶ **TypeScript is just JavaScript**  
TypeScript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript.
- ▶ **TypeScript supports other JS libraries**  
Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.
- ▶ **JavaScript is TypeScript**  
This means that any valid .js file can be renamed to .ts and compiled with other TypeScript files.
- ▶ **TypeScript is portable**  
TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on.
- ▶ **Based on ECMAScript 4,5,6 and some Microsoft magik!**



# WHY USE TYPESCRIPT?

- ▶ **TypeScript is more reliable**  
In contrast to JavaScript, TypeScript code is more reliable and easier to refactor. This enables developers to evade errors and do rewrites much easier.
- ▶ **TypeScript is more explicit**  
Making types explicit focuses our attention on how exactly our system is built, and how different parts of it interact with each other. In large-scale systems, it is important to be able to abstract away the rest of the system while keeping the context in mind.
- ▶ **Compiled**  
This means that any error or something else will be notified to the developer, also during the compile phase.

Yes, i said  
**COMPILED**



# WHAT DO YOU NEED?

- ▶ **Node.js**  
We will talk about this in the future.
- ▶ **TypeScript Compiler**  
A Node.js module that compiles TypeScript into JavaScript.
- ▶ **IDE or Visual Studio Code**  
Is a code editor that supports TypeScript. VS Code is highly recommended.
- ▶ **Live Server**  
Allows you to launch a development local Server with the hot reload feature.

# TYPESCRIPT GETTING STARTED

## ► Node.js Installation

To install Node.js we will use a **Node Version Manager** (or NVM) which allows us to select, install and switch the Node.js version that we want to use.

### ► On Linux, macOS (<https://github.com/nvm-sh/nvm#installing-and-updating>)

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash
$ export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s "${XDG_CONFIG_HOME}/nvm")"
$ [ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
```

### ► On Windows (<https://github.com/coreybutler/nvm-windows/releases>)

Download the installer and install on your system.

After the installation, we can use the nvm command to install and use a Node.js version:

- `nvm install <version>`
- `nvm use <version>`

# TYPESCRIPT GETTING STARTED

## ▶ **TypeScript compiler Installation**

To install the TypeScript compiler, you launch the Terminal on macOS or Linux and Command Prompt on Windows and type the following command:

- ▶ `npm install -g typescript`
- ▶ `npm install -g ts-node`

After the installation, you can type the following command to check the current version of the TypeScript compiler:

- ▶ `tsc --v`

# TYPESCRIPT HELLO WORLD

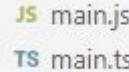
To start we need to define a folder as root of our project, subsequently create a new file named **main.ts**. Yes, the extension of TypeScript file is **.ts** .

In this file type the following code:

```
let message: string = 'Hello, World!';  
console.log(message);
```

Then open a new Terminal inside of the project folder and execute the following command to compile the **main.ts** file.

**tsc main.ts**



If everything is ok, we will see a new file called **main.js**

Finally if you want run your code inside of your terminal, try this command;

**node main.js**

or **ts-node main.ts**

# TYPESCRIPT HELLO WORLD IN WEB

First, create a new file called `index.html` and include the `main.js` as follows:

```
<!DOCTYPE html>
<html lang="it">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>TypeScript: Hello, World!</title>
</head>
<body>
  <script src="main.js"></script>
</body>
</html>
```

Second, change the `main.ts` code to the following:

```
let message: string = 'Hello, World!';
// create a new heading 1 element
let heading = document.createElement('h1');
heading.textContent = message;
// add the heading the document
document.body.appendChild(heading);
```

Then, compile it.

# TYPESCRIPT TYPE ANNOTATION

TypeScript uses type annotations to explicitly specify types for identifiers such variables, functions, objects, etc.

TypeScript uses the syntax `:type` after an identifier as the type annotation, where type can be any valid type.

Once an identifier is annotated with a type, it can be used as that type only. If the identifier is used as a different type, the [TypeScript compiler will issue an error.](#)

```
let variableName: type;  
let variableName: type = value;  
const constantName: type = value;
```

In this syntax, the type annotation comes after the variable or constant name and `:` is preceded by a colon (`:`).

```
let counter: number;  
counter = 1;  
counter = 'Hello'; // compile error Type '"Hello"' is not assignable to type 'number'.
```

# TYPESCRIPT TYPES ANNOTATION

## ▶ Primitives

```
let name: string = 'John';  
let age: number = 25;  
let active: boolean = true;
```

## ▶ Arrays

```
let arrayName: type[];  
  
let names: string[] = ['Mario', 'Luigi', 'Peach', 'Toad'];
```

## ▶ Objects

```
let character: {  
    name: string;  
    age: number  
};  
  
character = {  
    name: 'Mario',  
    age: 30  
};
```

# TYPESCRIPT FUNCTIONS ANNOTATION

The following shows a function annotation with parameter type annotation and return type annotation:

```
let greeting : (name: string) => string;
greeting = function (name: string) {
    return `Hello ${name}`;
};
```

In this example, you can assign any function that accepts a string and returns a string to the greeting variable.

The following causes an error because the function that is assigned to the greeting variable doesn't match with its function type.

```
greeting = function () {
    console.log('Hello');
};

//Error : Type '() => void' is not assignable to type '(name: string) => string'. Type
'void' is not assignable to type 'string'.
```

# TYPESCRIPT TYPE INFERENCE

When you declare a variable, you can use a type annotation to explicitly specify a type for it.

```
let counter: number;
```

However, if you initialize the `counter` variable to a number, TypeScript will infer the type of the `counter` variable to be `number`.

```
let counter = 0;  
//It is equivalent to the following statement:  
let counter: number = 0;
```

Likewise, when you assign a function parameter a value and a return value, [TypeScript infers the type of the parameter and the return](#) to the type of the default value.

```
function increment(counter: number) {  
    return counter++;  
}  
  
//It is same as:  
  
function increment(counter: number) => number {  
    return counter++;  
}
```

# TYPESCRIPT TYPE INFERENCE

Consider the following array assignment:

```
let items = [1, 2, 3, null];
```

To infer the type of **items** variable, TypeScript needs to consider the type of each element in the array.

It uses the best common type algorithm to analyze each candidate type and select the type that is compatible with all other candidates.

In this case, TypeScript selects the number array type (**number[]**) as the best common type.

# TYPESCRIPT TYPE NUMBER

All numbers in TypeScript are either floating-point values or big integers. The floating-point numbers have the type **number** while the big integers get the type **bigint**.

```
let val: number;  
let val = 9.50;  
let bin = 0b100; //binary  
let octal = 0o10; //octal  
let hex = 0xFF; //hex  
  
// bigint  
let big: bigint = 9007199254740991n;
```

# TYPESCRIPT TYPE OBJECT

The TypeScript object type represents all values that are not in primitive types. The following are primitive types in TypeScript:

- ▶ `number`
- ▶ `bignum`
- ▶ `string`
- ▶ `boolean`
- ▶ `null`
- ▶ `undefined`
- ▶ `symbol`

The following shows how to declare a variable that holds an object:

```
let player: object;  
player = {  
    nickname: 'Fedyfausto',  
    email: 'federico.santoro@unict.it',  
    level: 33,  
    title: 'The Left Hand of God'  
};  
  
console.log(player);
```

# TYPESCRIPT TYPE OBJECT

TypeScript has another type called **Object** with the letter **O** in uppercase. It's important to understand the differences between them. The **object** type represents all non-primitive values while the **Object** type describes the functionality of all objects.

For example, the **Object** type has the **toString()** and **valueOf()** methods that can be accessible by any object.

TypeScript has another type called empty type denoted by **{ }**, which is quite similar to the object type. The empty type **{ }** describes an object that has no property on its own. If you try to access a property on such object, TypeScript will issue a compile-time error.

# TYPESCRIPT TYPE TUPLE

A tuple works like an array with some additional considerations:

- ▶ The number of elements in the tuple is fixed.
- ▶ The types of elements are known, and need not be the same.

For example, you can use a tuple to represent a value as a pair of a **string** and a **number**:

```
let skill: [string, number];
skill = ['Swordsmanship', 90];
```

The order of values in a tuple is important. If you change the order of values of the skill tuple to [90, 'Swordsmanship'], you'll get an error. For this reason, it is a good practice to use tuples with data that is related to each other in a specific order.

```
let color: [number, number, number] = [255, 0, 0];
```

A tuple can have optional elements specified using the question mark (?) postfix:

```
let bgColor, headerColor: [number, number, number, number?];
bgColor = [0, 255, 255, 0.5];
headerColor = [0, 255, 255];
```

# TYPESCRIPT TYPE ENUM

An enum is a group of named constant values. Enum stands for enumerated type. To define an enum, you follow these steps:

- ▶ First, use the `enum` keyword followed by the name of the enum.
- ▶ Then, define constant values for the enum.

```
enum name {constant1, constant2, ...};
```

It is a good practice to use the constant values defined by enums in the code.

TypeScript defines the numeric value of an enum's member based on the order of that member that appears in the enum definition. It is possible to explicitly specify numbers for the members of an enum.

```
enum Month {  
    Jan = 1,  
    Feb,  
    Mar,  
    ...  
};
```

# TYPESCRIPT TYPE ANY

Sometimes, you may need to store a value in a variable. But you do not know its type at the time of writing the program. And the unknown value may come from a third party API or user input.

In this case, you want to opt-out of the type checking and allow the value to pass through the compile-time check.

To do so, you use the **any** type. The any type allows you to assign a value of any type to a variable:

```
const json = `{"player": "Fedyfausto", "level":33}`;
const currentPlayer = JSON.parse(json);
```

The any type provides you with a way to work with existing JavaScript codebase. It allows you to gradually opt-in and opt-out of type checking during compilation. Therefore, you can use the any type for migrating a JavaScript project over to TypeScript.

If you declare a variable without specifying a type, TypeScript assumes that you use the any type.

If you declare a variable with the object type, you can also assign it any value. However, you cannot call a method on it even though the method actually exists.

```
let result: any;
result = 10.123;
console.log(result.toFixed()); // OK
let result: object;
result = 10.123;
console.log(result.toFixed()); // error
```

# TYPESCRIPT TYPE VOID

The **void** type denotes the absence of having any type at all. It is a little like the opposite of the **any** type. Typically, you use the **void** type as the return type of functions that do not return a value.

```
function log(message): void {  
    console.log(message);  
}
```

It is a good practice to add the **void** type as the return type of a function or a method that does not return any value. By doing this, you can gain the following benefits:

- ▶ Improve clarity of the code: you do not have to read the whole function body to see if it returns anything.
- ▶ Ensure type-safe: you will never assign the function with the void return type to a variable.

Notice that if you use the void type for a variable, you can only assign undefined to that variable.

```
let useless: void = undefined;  
useless = 1; // error
```

# TYPESCRIPT TYPE NEVER

The **never** type is a type that contains no values. Because of this, you cannot assign any value to a variable with a never type.

Typically, you use the never type to represent the return type of a function that always throws an error or an indefinite loop.

```
function throwError(message: string): never {  
    throw new Error(message);  
}
```

Variables can also acquire the never type when you narrow its type by a type guard that can never be true.

# TYPESCRIPT TYPE UNION

Sometimes, you will run into a function that expects a parameter that is either a number or a string. For example:

```
function add(a: any, b: any) {  
    if (typeof a === 'number' && typeof b === 'number') {  
        return a + b;  
    }  
    if (typeof a === 'string' && typeof b === 'string') {  
        return a.concat(b);  
    }  
    throw new Error('Parameters must be numbers or strings');  
}
```

In this example:

- ▶ the `add()` function will calculate the sum of its parameters if they are numbers.
- ▶ In case the parameters are strings, the `add()` function will concatenate them into a single string.
- ▶ If the parameters are neither numbers nor strings, the `add()` function throws an error.

# TYPESCRIPT TYPE UNION

The problem with the parameters of the add() function is that its parameters have the any type. It means that you can call the function with arguments that are neither numbers nor strings, the TypeScript will be fine with it. To resolve this, you can use the TypeScript union type. The union type allows you to combine multiple types into one type.

```
let result: number | string;  
result = 10; // OK  
result = 'Hi'; // also OK  
result = false; // a boolean value, not OK
```

A union type describes a value that can be one of several types, not just two. For example **number | string | boolean** is the type of a value that can be a number, a string, or a boolean.

```
function add(a: number | string, b: number | string) {  
    if (typeof a === 'number' && typeof b === 'number') {  
        return a + b;  
    }  
    if (typeof a === 'string' && typeof b === 'string') {  
        return a.concat(b);  
    }  
    throw new Error('Parameters must be numbers or strings');  
}
```

# TYPESCRIPT TYPE ALIASES

Type aliases allow you to create a new name for an existing type.

```
type alias = existingType;
```

The existing type can be any valid TypeScript type. The following example use the type alias chars for the string type:

```
type chars = string;
let message: chars; // same as string type
```

It is useful to create type aliases for union types. For example:

```
type alphanumeric = string | number;
let input: alphanumeric;
input = 100; // valid
input = 'Hi'; // valid
input = false; // Compiler error
```

# TYPESCRIPT TYPE LITERAL STRING

The string literal types allow you to define a type that accepts only one specified string literal.

```
let click: 'click';
```

The `click` is a string literal type that accepts only the string literal '`click`'. If you assign the literal string `click` to the `click`, it will be valid. However, [when you assign another string literal to the click, the TypeScript compiler will issue an error](#).

The string literal type is useful to limit a possible string value in a variable. The string literal types can combine nicely with the [union types](#) and aliases types to define a finite set of string literal values for a variable:

```
type MouseEvent: 'click' | 'dblclick' | 'mouseup' | 'mousedown';
let mouseEvent: MouseEvent;
mouseEvent = 'click'; // valid
mouseEvent = 'dblclick'; // valid
mouseEvent = 'mouseup'; // valid
mouseEvent = 'mousedown'; // valid
mouseEvent = 'mouseover'; // compiler error

let anotherEvent: MouseEvent;
```

# TYPESCRIPT OPTIONAL PARAMS

In JavaScript, you can call a function without passing any arguments even though the function specifies parameters. Therefore, JavaScript supports the optional parameters by default.

In TypeScript, the compiler checks every function call and issues an error in the following cases:

- ▶ The number of arguments is different from the number of parameters specified in the function.
- ▶ Or the types of arguments are not compatible with the types of function parameters.

Because the compiler thoroughly checks the passing arguments, you need to annotate optional parameters to instruct the compiler not to issue an error when you omit the arguments. To make a function parameter optional, you use the **?** after the parameter name.

```
function multiply(a: number, b: number, c?: number): number {  
  
    if (typeof c !== 'undefined') {  
        return a * b * c;  
    }  
    return a * b;  
}
```

# TYPESCRIPT REST PARAMS

A rest parameter allows you a function to accept zero or more arguments of the specified type. In TypeScript, rest parameters follow these rules:

- ▶ A function has only one rest parameter.
- ▶ The rest parameter appears last in the parameter list.
- ▶ The type of the rest parameter is an array type.

To declare a rest parameter, you prefix the parameter name with three dots and use the array type as the type annotation:

```
function fn(...rest: type[]) {  
    //...  
}
```

# TYPESCRIPT FUNCTION OVERLOADING

In TypeScript, function overloading allow you to establish the relationship between the parameter types and result types of a function.

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
    return a + b;
}
```

In this example, we added two overloads to the `add()` function. The first overload tells the compiler that when the arguments are numbers, the `add()` function should return a number. The second overload does the same but for a string.

Each function overload defines a combination of types supported by the `add()` function. It describes the mapping between the parameters and the result they return.

# TYPESCRIPT FUNCTION OVERLOADING

When you overload a function, the number of required parameters must be the same. If an overload has more parameters than the other, you have to make the additional parameters optional.

```
function sum(a: number, b: number): number;
function sum(a: number, b: number, c: number): number;
function sum(a: number, b: number, c?: number): number {
    if (c) return a + b + c;
    return a + b;
}
```

The `sum()` function accepts either two or three numbers. The third parameter is optional. If you do not make it optional, you will get an error.

When a function is a property of a class, it is called a method. TypeScript also supports method overloading.

# TYPESCRIPT CLASSES

JavaScript does not have a concept of class like other programming languages. ES6 allowed you to define a class which is simply syntactic sugar for creating constructor function and prototypal inheritance:

```
class Person {  
    ssn;  
    firstName;  
    lastName;  
  
    constructor(ssn, firstName, lastName) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName() {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

# TYPESCRIPT CLASSES

TypeScript class adds type annotations to the properties and methods of the class. The following shows the Person class in TypeScript:

```
class Person {  
    ssn: string;  
    firstName: string;  
    lastName: string;  
  
    constructor(ssn: string, firstName: string, lastName: string) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

# TYPESCRIPT CLASSES - ACCESS MOD

Access modifiers change the visibility of the properties and methods of a class. TypeScript provides three access modifiers: private, protected and public. Note that TypeScript controls the access logically during compilation time, not at runtime.

The **private** modifier limits the visibility to the same-class only. When you add the **private** modifier to a property or method, you can access that property or method within the same class. Any attempt to access private properties or methods outside the class will result in an error at compile time.

The **public** modifier allows class properties and methods to be accessible from all locations. If you don't specify any access modifier for properties and methods, they will take the **public** modifier by default.

The **protected** modifier allows properties and methods of a class to be accessible within same class and within subclasses.

When a class (child class) inherits from another class (parent class), it is a subclass of the parent class.

# TYPESCRIPT CLASSES - READ ONLY

TypeScript provides the **readonly** modifier that allows you to mark the properties of a class immutable. The assignment to a readonly property can only occur in one of two places, in the property declaration and in the constructor of the same class.

```
class Person {  
    readonly birthDate: Date;  
  
    constructor(birthDate: Date) {  
        this.birthDate = birthDate;  
    }  
}
```

In this example, the **birthdate** property is a readonly property that is initialized in the constructor of the **Person** class.

# TYPESCRIPT CLASSES - GET/SET

The getters and setters allow you to control the access to the properties of a class.

- ▶ A getter method returns the value of the property's value. A getter is also called an accessor.
- ▶ A setter method updates the property's value. A setter is also known as a mutator.

```
class Person {  
    private _age: number;  
    private _firstName: string;  
    private _lastName: string;  
  
    public get age() {  
        return this._age;  
    }  
  
    public set age(theAge: number) {  
        if (theAge <= 0 || theAge >= 200) {  
            throw new Error('The age is invalid');  
        }  
        this._age = theAge;  
    }  
}
```

# TYPESCRIPT CLASSES - INHERITANCE

A class can reuse the properties and methods of another class. This is called inheritance in TypeScript.

The class which inherits properties and methods is called the **child class**. And the class whose properties and methods are inherited is known as the **parent class**.

```
class Person {  
    constructor(private firstName: string, private lastName: string) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    getFullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

To inherit a class, you use the **extends** keyword. For example the following **Employee** class inherits the Person class:

```
class Employee extends Person {  
    //..  
}
```

# TYPESCRIPT CLASSES - INHERITANCE

Because the `Person` class has a constructor that initializes the `firstName` and `lastName` properties, you need to initialize these properties in the constructor of the `Employee` class by calling its parent class' constructor.

To call the constructor of the parent class in the constructor of the child class, you use the `super()` syntax.

```
class Employee extends Person {  
    constructor(  
        firstName: string,  
        lastName: string,  
        private jobTitle: string) {  
  
        // call the constructor of the Person class:  
        super(firstName, lastName);  
    }  
}
```

Because the `Employee` class inherits properties and methods of the `Person` class, you can call the `getFullName()` method on the `employee` object.

# TYPESCRIPT CLASSES - INHERITANCE

When you call the `employee.getFullName()` method on the `employee` object, the `getFullName()` method of the `Person` class is executed. If you want the `Employee` class has its own version of the `getFullName()` method, you can define it in the `Employee` class.

```
class Employee extends Person {  
    constructor(  
        firstName: string,  
        lastName: string,  
        private jobTitle: string) {  
  
        super(firstName, lastName);  
    }  
  
    getFullName(): string {  
        return super.getFullName() + `I'm a ${this.jobTitle}.`;  
    }  
}
```

# TYPESCRIPT CLASSES - STATIC

Unlike an instance property, a static property is shared among all instances of a class. To declare a static property, you use the **static** keyword. To access a static property, you use the **className.propertyName** syntax. Similar to the static property, a **static method** is also shared across instances of the class.

```
class Employee {  
    private static headcount: number = 0;  
  
    constructor(  
        private firstName: string,  
        private lastName: string,  
        private jobTitle: string) {  
  
        Employee.headcount++;  
    }  
  
    public static getHeadcount() {  
        return Employee.headcount;  
    }  
}
```

# TYPESCRIPT CLASSES - ABSTRACT

An abstract class is typically used to define common behaviors for derived classes to extend. Unlike a regular class, **an abstract class cannot be instantiated directly**. To declare an abstract class, you use the **abstract** keyword.

```
abstract class Employee {  
    constructor(private firstName: string, private lastName: string) {  
    }  
    abstract getSalary(): number  
    get fullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

Typically, an abstract class contains one or more abstract methods. An abstract method does not contain implementation. It only defines the signature of the method without including the method body.

# TYPESCRIPT CLASSES - ABSTRACT

```
class FullTimeEmployee extends Employee {  
    constructor(firstName: string, lastName: string, private salary: number) {  
        super(firstName, lastName);  
    }  
    getSalary(): number {  
        return this.salary;  
    }  
}
```

In this `FullTimeEmployee` class, the salary is set in the constructor. Because the `getSalary()` is an abstract method of the `Employee` class, the `FullTimeEmployee` class needs to implement this method. In this example, it just returns the salary without any calculation.

# TYPESCRIPT INTERFACE

TypeScript interfaces define the contracts within your code. They also provide explicit names for type checking.

```
interface Person {  
    firstName: string;  
    lastName: string;  
}
```

By convention, the interface names are in the camel case. They use a single capitalized letter to separate words in there names.

After defining the Person interface, you can use it as a type. And you can annotate the function parameter with the interface name:

```
function getFullName(person: Person) {  
    return `${person.firstName} ${person.lastName}`;  
}  
  
let fedy = {  
    firstName: 'Federico',  
    lastName: 'Santoro'  
};  
console.log(getFullName(fedy));
```

# TYPESCRIPT INTERFACE

In addition to describing an object with properties, **interfaces also allow you to describe function types.** To describe a function type, you assign the interface to the function signature that contains the parameter list with types and returned types.

```
interface StringFormat {  
    (str: string, isUpper: boolean): string  
}  
  
let format: StringFormat;  
  
format = function (str: string, isUpper: boolean) {  
    return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();  
};  
  
console.log(format('hi', true));
```

The **StringFormat** interface ensures that all the callers of the function that implements it pass in the required arguments: a **string** and a **boolean**.

# TYPESCRIPT INTERFACE

If you have worked with Java or C#, you can find that the main use of the [interface is to define a contract between unrelated classes.](#)

For example, the following Json interface can be implemented by any unrelated classes:

```
interface Json {  
    toJSON(): string  
}
```

The following declares a class that implements the Json interface:

```
class Person implements Json {  
    constructor(private firstName: string,  
               private lastName: string) {  
    }  
    toJSON(): string {  
        return JSON.stringify(this);  
    }  
}
```

# TYPESCRIPT INTERFACE - MULTIPLE

To extend an interface, you use the extends keyword with the following syntax:

```
interface A {  
    a(): void  
}  
  
interface B extends A {  
    b(): void  
}
```

The interface B extends the interface A, which then have both methods a() and b() . An interface can extend multiple interfaces, creating a combination of all the interfaces.

```
interface C {  
    c(): void  
}  
  
interface D extends B, C {  
    d(): void  
}
```

In this example, the interface D extends the interfaces B and C. So D has all the methods of B and C interfaces, which are a(), b(), and c() methods.

# TYPESCRIPT INTERFACE - CLASSES

TypeScript **allows an interface to extend a class**. In this case, the interface inherits the properties and methods of the class. Also, the interface can inherit the private and protected members of the class, not just the public members.

It means that when an interface extends a class with private or protected members, the interface can only be implemented by that class or subclasses of that class from which the interface extends.

By doing this, you restrict the usage of the interface to only class or subclasses of the class from which the interface extends.

```
class Control {  
    private state: boolean;  
}  
  
interface StatefulControl extends Control {  
    enable(): void  
}  
  
class Button extends Control implements StatefulControl {  
    enable() {}  
}  
  
class TextBox extends Control implements StatefulControl {  
    enable() {}  
}
```



# TYPESCRIPT TYPE CASTING

JavaScript does not have a concept of type casting because variables have dynamic types. However, every variable in TypeScript has a type. Type castings allow you to convert a variable from one type to another.

In TypeScript, you can use the **as** keyword or **<>** operator for type castings.

```
let input = document.querySelector('input["type="text"]');
let enteredText = (input as HTMLInputElement).value;
```

Besides the **as** keyword, you can use the **<>** operator to carry a type casting.

```
let input = <HTMLInputElement>document.querySelector('input[type="text"]);

console.log(input.value);
```

Also, the casting keyword or operator can be used for type assertion or type narrowing.

# TS RECAP!

- ▶ TS is a typed multi-paradigm programming language.
- ▶ TS is compiled in JS and then executed by the browser.
- ▶ Also, you can use it on server side!
- ▶ TS is a superset of JS
- ▶ JS is also TS code
- ▶ There is a lot of things to introduce (such as modules!)

# 7.

## FRONTEND ENV.

HTML, CSS, JS,  
FRAMEWORKS

# THE WEB SITE FRONTEND

As we said a web application and web site can be split up in two parts, the **Frontend** and the Backend.

We just saw that the Frontend is all the visual stuff you see on the web application, for instances the images, the buttons, the other graphical elements and the interaction with the final user.

# THE FRONTEND TECHNOLOGIES



- ▶ **HTML**

Markup language used to define the structure of the HTML Document

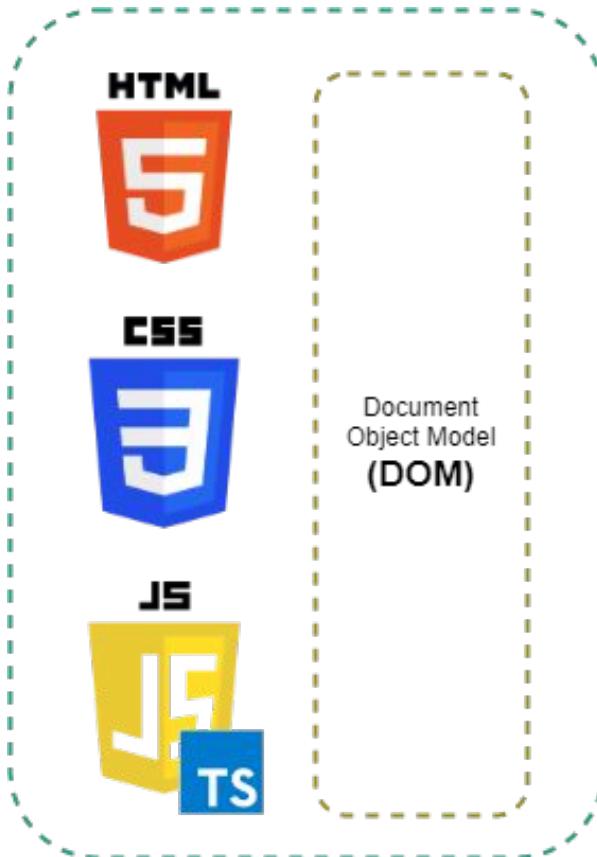
- ▶ **CSS**

Cascade language used to define rules, style and behavior of HTML elements.

- ▶ **JS and TS**

Programming Language use to implement the user interaction and application logic.

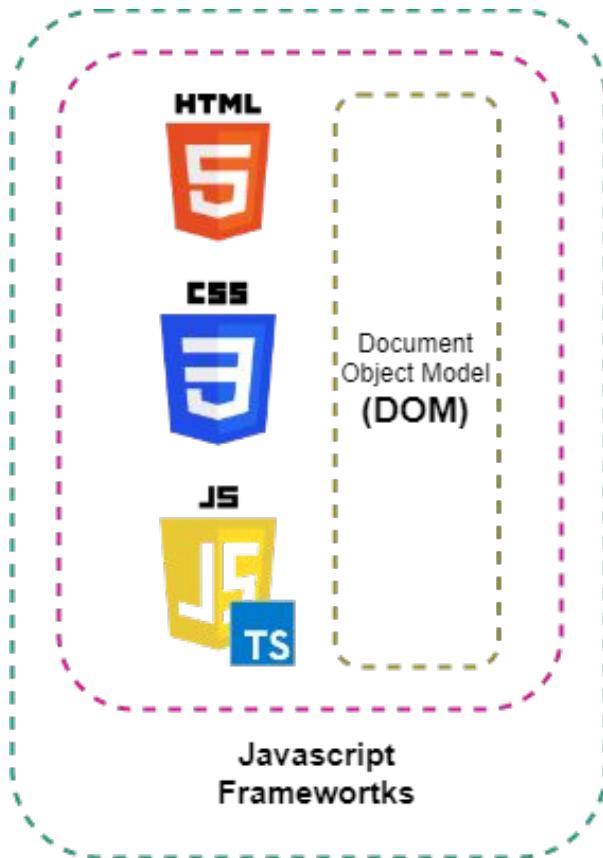
# THE FRONTEND FRAMEWORKS



The behaviors, style, content and other things about HTML elements can be managed with a special features of JS, the **Document Object Model**.

Anyway, using the DOM could be repetitive and hard to manage. To simplify this, there are a lot of **Javascript Frameworks!**

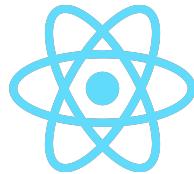
# THE FRONTEND FRAMEWORKS



Javascript frameworks gives a  
nicer way to create a Web  
Application and they take care of  
updating the entire page and  
elements.

Then, using a JS Framework  
allow to do not manage the DOM  
directly anymore.

# THE FRONTEND FRAMEWORKS

**React.js**

- ▶ Uses JS and also TS
- ▶ It is more a library
- ▶ Minimal by design

**Angular**

- ▶ Uses TS
- ▶ Complete Platform
- ▶ Not really flexible

**Vue.js**

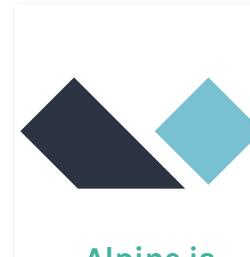
- ▶ Uses JS and also TS
- ▶ Progressive Framework
- ▶ Plugin System

**Svelte.js**

- ▶ Compile to JS
- ▶ No Virtual DOM
- ▶ Minimal and Flexible

**Solid.js**

- ▶ Compile to JS
- ▶ No Virtual DOM
- ▶ High Performance

**Alpine.js**

- ▶ Uses JS with HTML
- ▶ Extends the HTML
- ▶ Lightweight

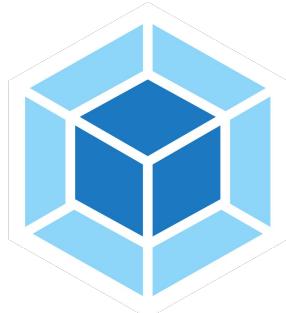
# THE FRONTEND BUNDLERS

Javascript, Typescript and all Frontend Frameworks are missing a lot of features that the other programming languages have.

One of these is being able to split up the code into different files and to organize our code.

To solve this we must use something called **bundler**.  
A bundler organizes a set of source files (as JS, images, css, etc.) into a single file.

# THE FRONTEND BUNDLERS



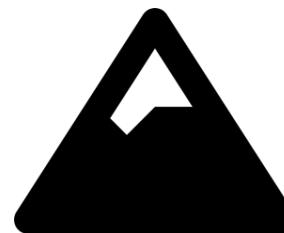
## Webpack

- ▶ Long-term caching mechanism
- ▶ Code splitting and lazy loading
- ▶ Handles the dependencies automatically



## Rollup

- ▶ Helps application optimizations
- ▶ Supports tree-shaking and scope-hoisting
- ▶ Offers more elasticity



## Snowpack

- ▶ Build dependencies once
- ▶ High Performances
- ▶ Supports lazy loading

# THE FRONTEND PREPROCESSORS

CSS as a language has the same problems as javascript.

It is difficult organize the code into different files and CSS is also missing a lot of useful features.

Then we can use such as bundler for CSS, There are called **Preprocessors**.

# THE FRONTEND PREPROCESSORS



## SASS

- ▶ Organize CSS into different files
- ▶ Enhances CSS
- ▶ Conditional logic



## Less

- ▶ Js Library
- ▶ Does not have conditional logic
- ▶ Offers built-in functions



## Stylus

- ▶ Combination of SASS and Less
- ▶ Conditional logic and postfix conditional
- ▶ Supports all mixins

# THE FRONTEND **CSS FRAMEWORKS**

The CSS Preprocessors and Javascript are also used to develop a set of styles, functionalities and widgets ready to be used in your Web Application.

These are called **CSS Frameworks**.

These are basically a bunch of CSS and JS code that solve common problems and save a lot of time.

# THE FRONTEND CSS FRAMEWORKS



## Bootstrap

- ▶ Most popular frontend framework
- ▶ Fully-featured
- ▶ Mature and Customizable



## Foundation

- ▶ Mobile First
- ▶ Generic Style
- ▶ Email design and Animations



## Bulma

- ▶ Aesthetic design
- ▶ Easy to customize
- ▶ No Javascript

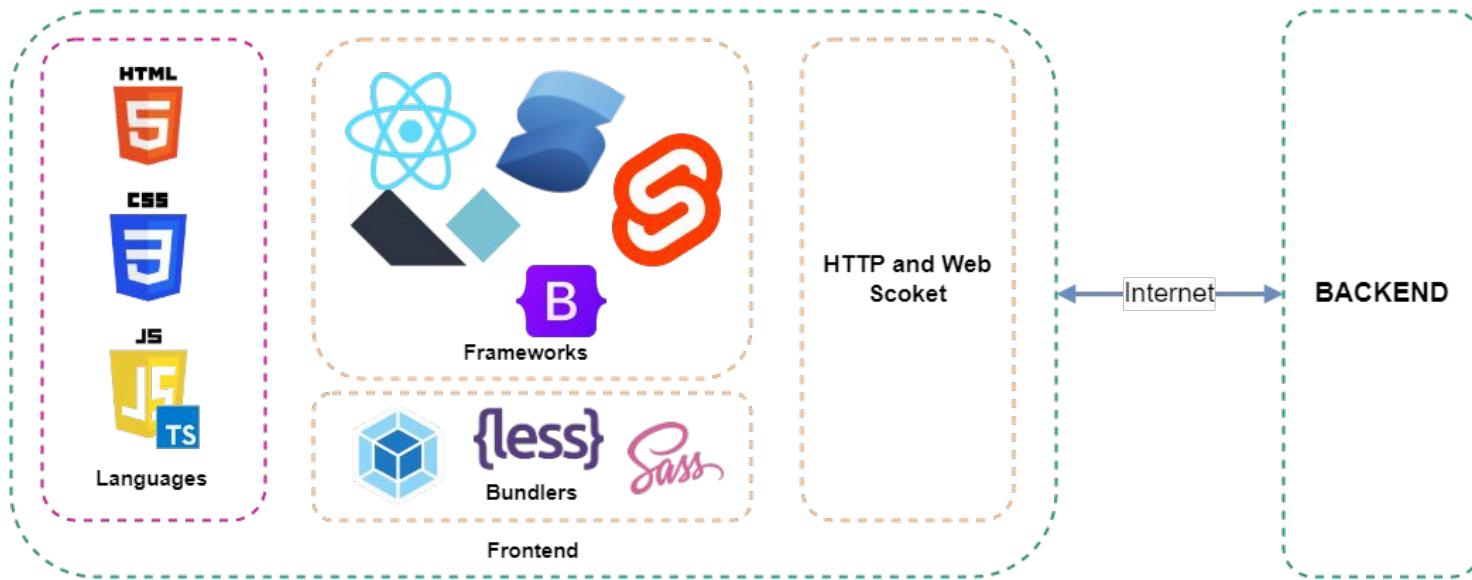


## Tailwind

- ▶ Atomic CSS
- ▶ Reusable components
- ▶ No design

# THE FRONTEND ENVIRONMENT

There are a lot of technologies to develop your Frontend, nevertheless all of these uses the languages that we saw in this course.



# FE RECAP!

- ▶ The frontend is the client and UI part of a web application
- ▶ The knowledge of HTML, CSS and JS is a must
- ▶ There are a lot of tools and frameworks that helps to develop web applications
- ▶ Try to study and learn one of these!

You can find the complete Frontendroadmap:

<https://www.freecodecamp.org/news/2019-web-developer-roadmap/#front-end-web-development-roadmap>

# 8.

## BACKEND INTRODUCTION

REQUEST, RESPONSE!

# THE CLIENT/SERVER INFRASTRUCTURE

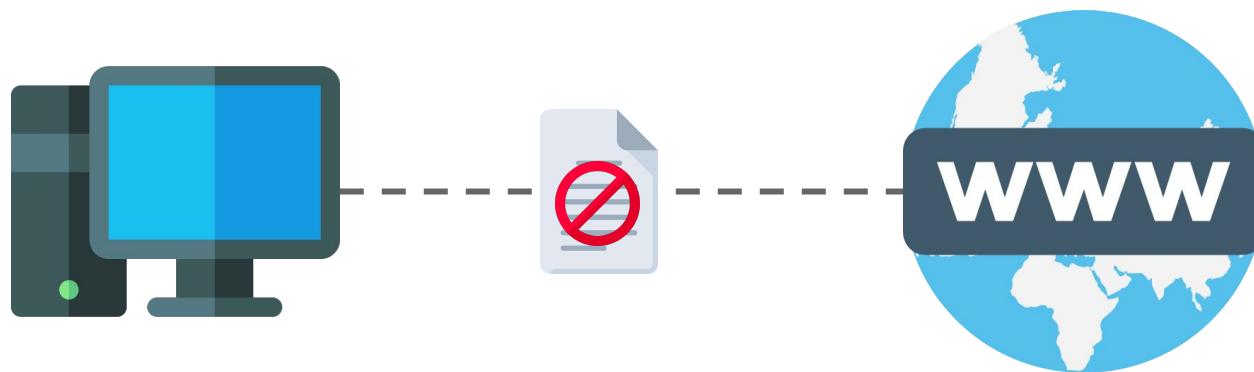
Any computer that is connected to the Internet can send message across the Internet to another computer



That computer is sending the message is called the **Client**, and the computer that is receiving the message is called the **Server**.

# THE CLIENT/SERVER INFRASTRUCTURE

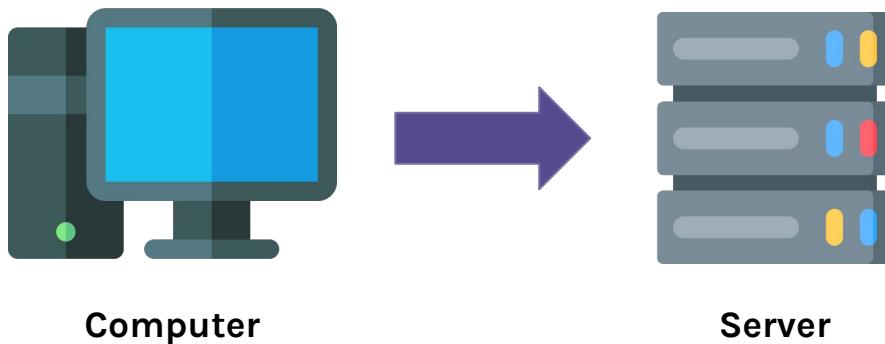
Before this happens, computers they can not receive messages from the Internet by default



We must program them to be able to receive messages.  
To do that we need a **Backend programming language**.

# THE BACKEND PROG. LANGS.

Almost every programming language has a feature that turns a computer into a server and allows it to receive messages.



You also need to think about aspects related to computer networks, such as routing and **socket ports**.

# THE BACKEND PROG. LANGS.

Among the most famous languages there are:



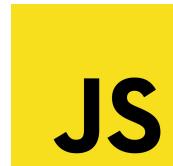
However, using a backend programming language by it self is actually really difficult, and requires a huge amount of code.

Then there are two tools that we use to help with this.

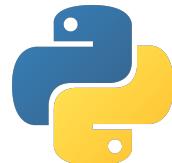
# THE BACKEND FRAMEWORKS

A Backend framework helps us create a server much easier with a lot less code.

Each backend programming language has a few different frameworks to choose from.



express



# THE BACKEND PACKAGES

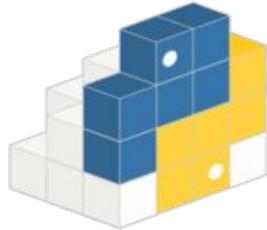
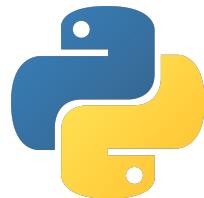
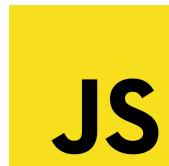
In the backend, in addition to frameworks, you also use parts of code that other users have written and disclosed, these are called **packages**.

They are usually libraries to simplify development, taking advantage of ready-made solutions, such as performing mathematical calculations, connecting to a database, or managing user authentication.

In order to install and use these packages, we will use a tool called **package manager**.

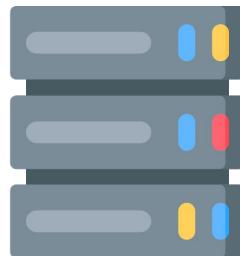
# THE BACKEND PACKAGES

Each language has its own package manager (even more than one)



# THE BACKEND TECHNOLOGIES

These are all the technologies we need to create our backend server.

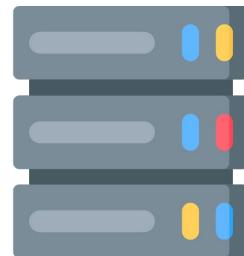


- ✓ Backend Programming Language
  - Backend Framework
  - Package Manager

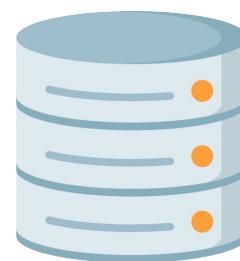
The next problem we have is we need somewhere to save the data for our web application. To do this we use a **database**.

# THE BACKEND DATABASES

A Database helps us store and manage data. It is just a piece of software that usually runs on a different computer (but also in the same of backend software) and we must to do some setup then our backend can communicate with the database.



**Server**  
**(Backend server)**



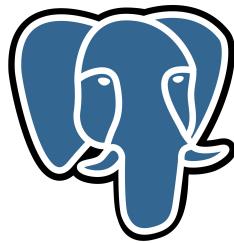
**Database**  
**(Database server)**

# THE BACKEND DATABASES



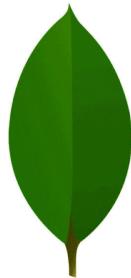
## MySQL (MariaDB)

- ▶ Relational database
- ▶ Single process
- ▶ For high volume of reads



## PostgreSQL

- ▶ Object relational database
- ▶ Multi process
- ▶ For high volume of I/O



## MongoDB

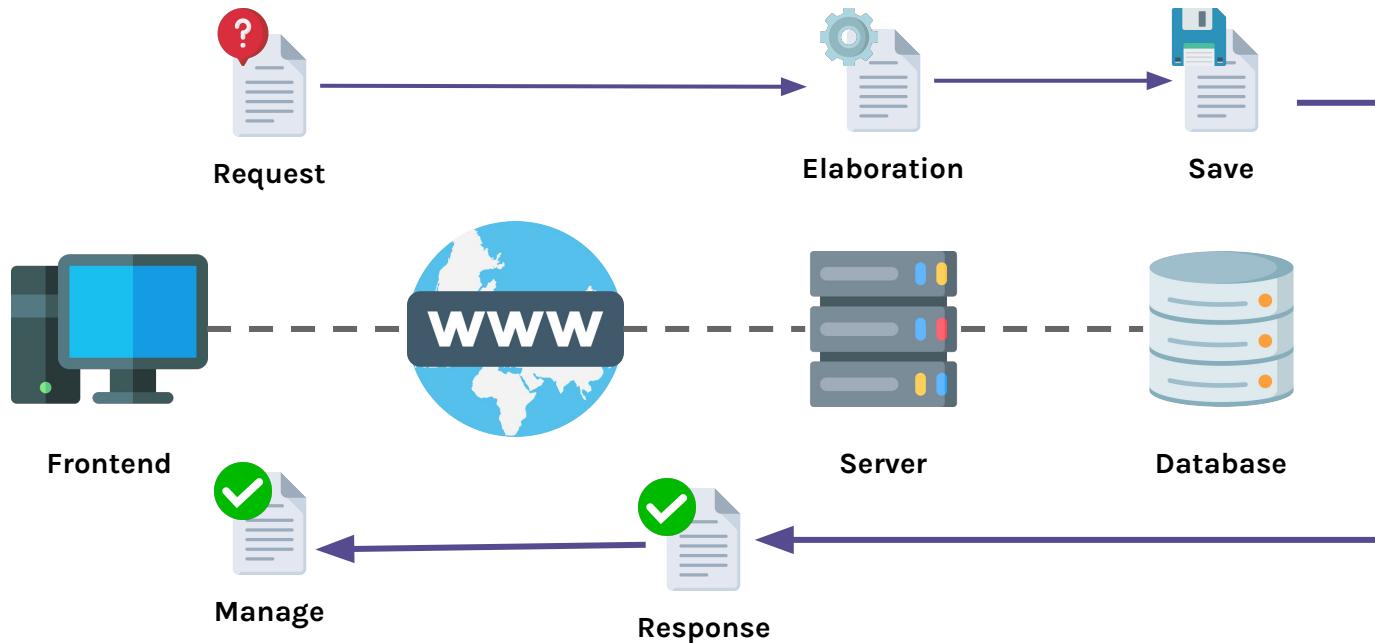
- ▶ JSON Document database
- ▶ Distributable
- ▶ For unstructured data

# THE BACKEND BASIC INFRASTRUCTURE

Then we are just starting out and this is basically all we need for the backend. You can build most of your projects with just a server and a database.



# THE BACKEND REQ-RESP CYCLE



This is generally how web applications work.

# THE BACKEND REQUEST EXAMPLE

```
POST http://youirdomani.com/orders
{
  order: [
    {
      id : 12345
      name : "Nintendo Switch",
      quantity : 1
    }
  ],
  paymentMethod : "paypal"
}
```

As we just seen before, in a HTTP request we can identify the **method** and the **path**.

In this example, we are sending a POST request to resource /orders to create and save a new order defined by JSON payload.

# THE BACKEND APIs

In the backend we use a programming language and a backend framework to define what type of request are allowed and how we should handle these requests.

```
app.post("/orders", (request, response) => {
  const order = new Order(request);
  database.save(order);
  response.send("Order saved");
}

app.get("/orders", (request, response) => {
  const orders = database.getOrders();
  response.json(orders);
}

app.delete("/orders/:id", (request, response) => {
  database.deleteOrder(request);
  response.send("Order deleted");
})
```

This list of all different types of request that the backend allows is called **API**

**(Application Programming Interface).**

API is one of the most important concepts in backend development. These allows other languages to communicate with others!

# THE BACKEND REST API

In this example do we choose **POST /orders** as a just naming convention for our requests. This naming convention is called **REST (REpresentational State Transfer)**. In REST the type of request has special meaning (such as POST for creating, etc.)

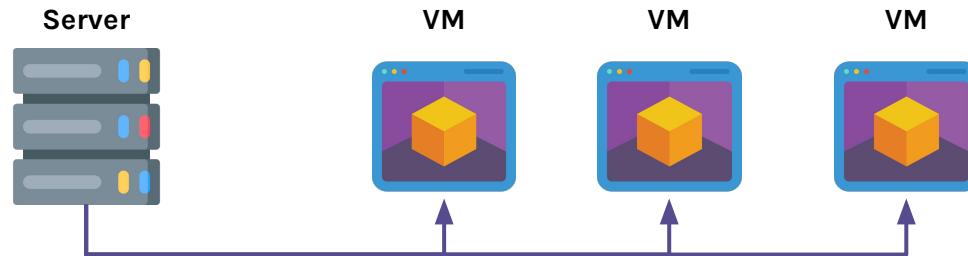
An API that uses the the REST naming conventions is called **REST API**.

This is NOT a protocol but an architecture!

There are several other conventions that we could use, such as **GraphQL** which uses POST for all requests and another one is called **RPC** (Remote Procedure Call) which uses POST and more detailed URLs.

# THE BACKEND INFRASTRUCTURE

Nowadays, instead of companies purchasing their own computers to run their web applications, they rent computers from a cloud computing company.

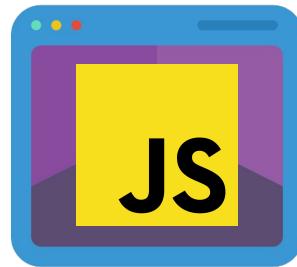


The basic idea of cloud computing is you are renting a bunch of computers. This is also known as **IaaS** (Infrastructure as a Service). Behind the scenes, these companies have a giant, powerful computer and inside its software it is running many smaller computers and we are renting one of these smaller computers. These smaller computers only exist in the software then we call them **Virtual Machines** (or vms). Typically, these are run by a hypervisor (such as kvm or qemu) via some tool (VMWare, Proxmox, Openstack).

# THE BACKEND INFRASTRUCTURE

Then, to run a web application we rent a virtual machine from a cloud company to run our backend, and we also rent another virtual machine to run our database.

Backend



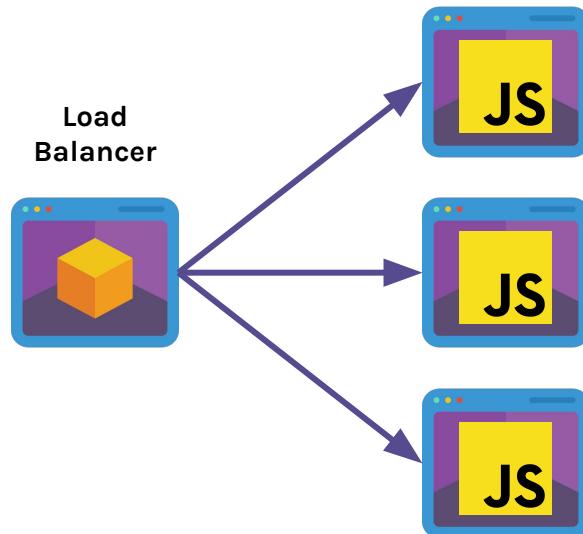
Database



Another problem we must solve is what if our web application gets really popular during some period, and we start getting a lot of requests and Internet traffic that our server can not handle?

# THE BACKEND INFRASTRUCTURE

With cloud computing, we can set up multiple VMs running the same backend software, and then set up a special VM in front of these called **Load Balancer**. A Load Balancer re-distribute requests evenly across our VMs. Once the high traffic period is over, we can just shut down some VMs when we do not need them.

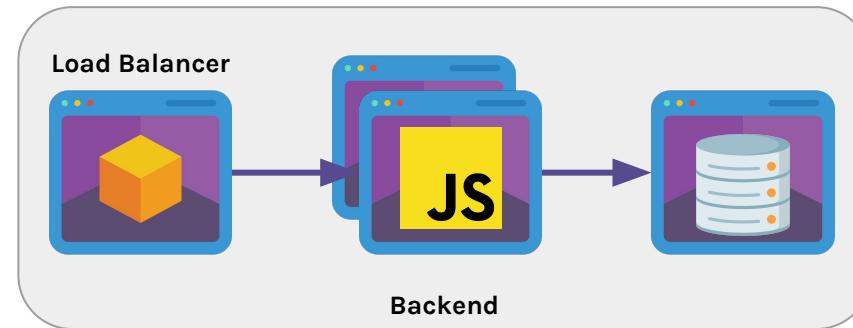


We still have another problem. We now have a lot of VMs that we need to create and setup, and it takes a lot of time and effort. Cloud computing companies offer another service called **PaaS** (Platform as a Service). This lets us upload our backend code and it will setup all VMs and Load Balancers for us.

Another problem is the huge of functions.

# THE BACKEND INFRASTRUCTURE

For instance, our backend contains code that saves an order to the database, charge some user's credit and email confirmation.

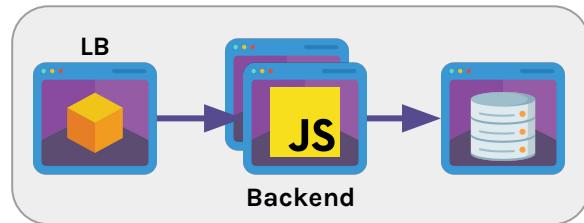


- ▶ Saves Orders
- ▶ Charges Credit
- ▶ Sends Emails

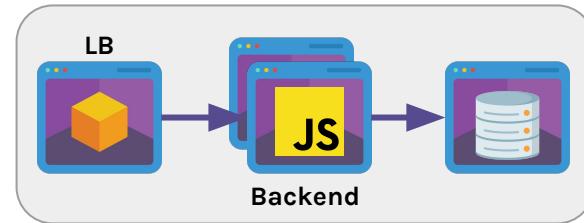
In the real world this backend can be millions of lines of code, so we split up into three code bases. Then each of these code bases will have their own backend, load balancer and database.

# THE BACKEND INFRASTRUCTURE

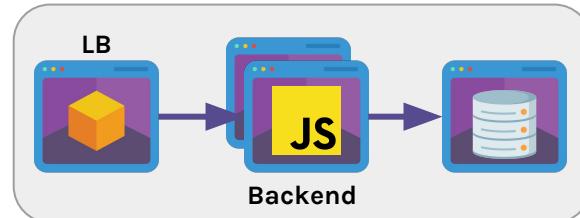
When we need to send an email, our orders backend will send a request to the email backend, which will send the email. Splitting up our backend into separate backends is called **Microservices**. It help us to maintain our base code smaller and focused.



Orders Backend



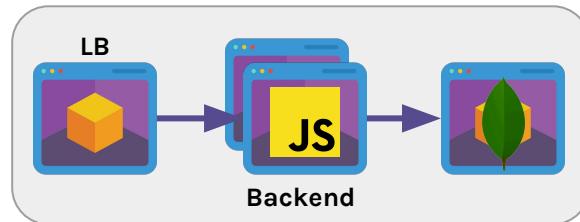
Payments Backend



Email Backend

# THE BACKEND INFRASTRUCTURE

Each Microservice does not have to use the same Load Balancer, programming language and database. For instance, one microservice can be using Javascript and MongoDB, another one can be using Python and PostgreSQL. In addition, a Microservice could be offered by another company.



Orders Backend



Payments Backend

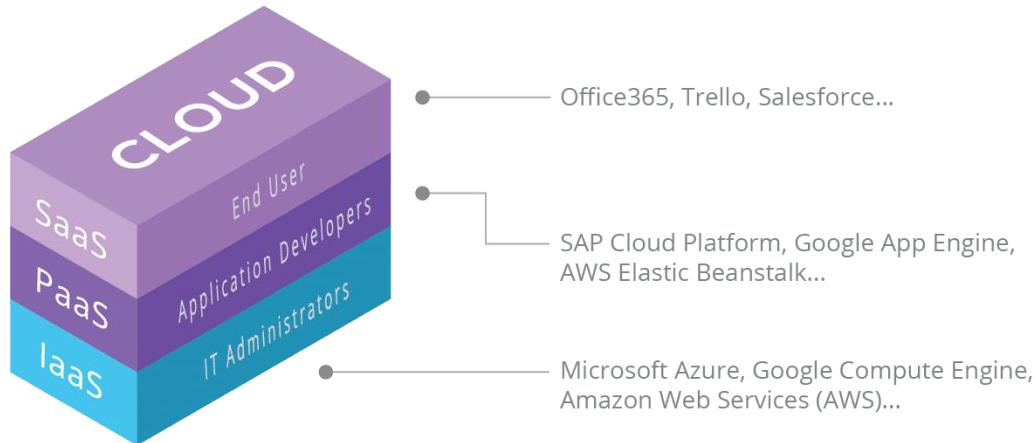


Email Service

When a company provides a Backend and an API that outside applications can use, this is called a **SaaS** (Software as a Service)

# THE BACKEND CLOUD COMPUTING

So these three concepts we just looked at, Infrastructure as a Service, Platform as a Service and Software as a Service are the three foundations of Cloud Computing.



These days most companies use cloud computing to run the backend for their web applications instead of buying and managing physical servers themselves (**Bare Metal**).

# THE BACKEND ADD. TECHNOLOGIES

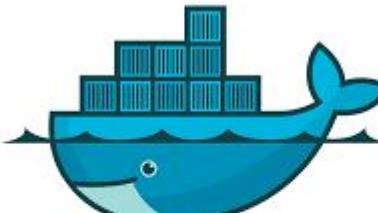
Previously we introduced some databases like MySQL, PostgreSQL and MongoDB. These sometimes are called **Primary Databases**, because they are the main databases that our web application uses.



Generally we start our backend with a server and primary database and then bring in these additional technologies if we need to. If we allow our users to upload images, a primary database is not good for storing such data, so we could use a **BLOB** store like a common CDN service to store and load user uploaded images.

# THE BACKEND ADD. TECHNOLOGIES

If our web application is getting a lot of traffic and we need to take some stress off from our primary database, we would add a cache service, like **Redis**, to improve performance.



Also, if we need to schedule something or a job queue, we would use **RabbitMQ** to schedule some task or messages.

# BACKEND RECAP!

- ▶ Basically, we need to know Client/Server architecture
- ▶ There are a lot of languages for the backend
- ▶ Also there are a lot of databases and other technologies!
- ▶ The cloud computing allows the setup of complex infrastructure
- ▶ If we need something, someone just did it as SaaS
- ▶ Experiment and develop your projects!

You can find the complete Backend roadmap:

<https://www.freecodecamp.org/news/2019-web-developer-roadmap/#back-end-web-development-roadmap>

# 9. **NODE JS** **INTRODUCTION**

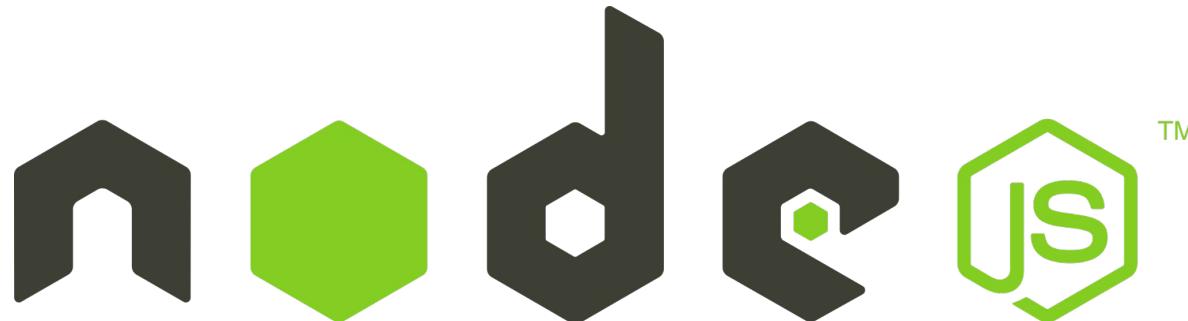
require("slides")

# WHAT IS NODE JS

Node.js is an environment to run Javascript outside of the Browser,

and then outside of the Browser's Sandbox. Was created in 2009 and it is built on top of Chrome V8 JS Engine. With the help of Node it is never been easier to build Full Stack web application, the Frontend and Backend are built using the same language!

It is open-source and cross-platform and is perfect for data-intensive and real-time applications.



# BROWSER VS NODE JS

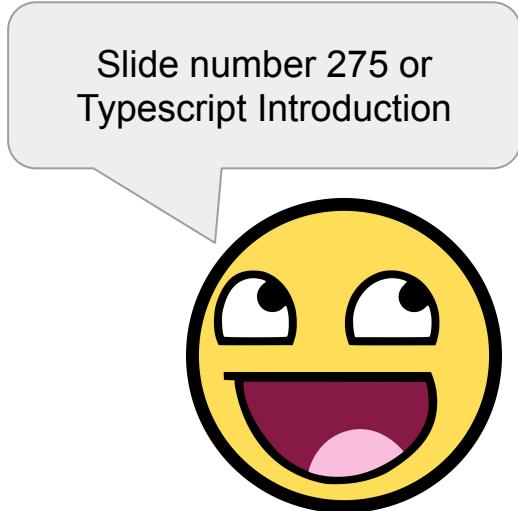
## Browser

- ▶ DOM
- ▶ Window
- ▶ Interactive Apps
- ▶ No Filesystem
- ▶ Fragmentation
- ▶ ES6 Modules
- ▶ Browser Console

## Node.js

- ▶ No DOM
- ▶ Global
- ▶ Server Side Apps
- ▶ Filesystem
- ▶ Versioning
- ▶ CommonJS
- ▶ O.S. Terminal

# INSTALL NODE JS



Slide number 275 or  
Typescript Introduction

# NODE JS HELLO WORLD

Firstly, we need to create a new empty folder as **Project Folder**. Inside of this directory create a new JS file (such as main.js or app.js, or whatever you want). Inside of this file try to create a simple code as hello world.

```
const amount = 17;

if (amount < 18){
    console.log("You did not pass the exam!");
} else {
    console.log("Maybe you can pass the exam...");
}

console.log("Joking :)");
```

To run every JS file on your system, just run **node <filename>** on your terminal.

All vanilla functions that we found in **window** object on web browsers, we found them in **global** object.

# NODE JS GLOBALS

In Node.js there are several global vars. In browsers we know that we have access to the window object and we can get bunch of useful things from this global, for example a querySelector or fetch. There is not window object. In Node.js there is a real concept of global variables.

- ▶ **`__dirname`**  
Path of current directory  
  
`console.log(__dirname)`
- ▶ **`__filename`**  
Current filename  
  
`console.log(__filename)`  
`console.log(process)`
- ▶ **`require`**  
Function to use modules
- ▶ **`module`**  
Info about current module
- ▶ **`process`**  
Info about env. where the program is executed

# NODE JS MODULES

When you run a Node.js application you run one file, but your application can be a set of files. The first file is commonly known as **Entrypoint** or **Main** file. Normally, you giant application code will be split up into modules. These are similar to the vanilla JS and React modules but the syntax is different, because Node.js uses **CommonJs** to load modules. By default, every file in node is a module!

Then to create a new module, you must create a new empty file.

# NODE JS MODULES

```
//names.js file
//local
const SECRET = "token";
// public
const fedy = "Federico";
const fausto = "Fausto";
const mario = "Mario";
module.exports = {fedy, fausto, mario}
//console.log(module);

=====
//myconsole.js
const trace = (message) => {
  console.log(message);
}
module.exports = trace
=====

//app.js file

const names = require("./names.js");
const trace = require("./myconsole.js");
trace(names);
```

For instances, create a new file where we will define some variables, one private (maybe a secret token) and some public variable that we want export. In the globals vars we have the **module** global that print some information about the module. There is a object called **exports** that defines what will be exported from the module. Then we can decide what can be “public” and what do not be.

Finally, we can import the new module with **require** global function, using the dot path syntax.

# NODE JS BUILT-IN MODULES

Node.js had a lot of built-in modules. To load one of these you must use their name instead of a path in require function.

- ▶ OS Module                    Each of these modules has its attributes and methods.
- ▶ PATH Module
- ▶ File System Module         You can find more built-in modules on the documentation!
- ▶ HTTP Module
- ▶ more...

# NODE JS BUILT-IN MODULES

**OS** module can provide a lot of information about operating system and more!

```
const os = require("os"); //os information

//info about user
const user = os.userInfo();
console.log(user);
//uptime
console.log(os.uptime());
```

```
const currentOS = {
  name : os.type(),
  release : os.release(),
  totalMem : os.totalmem(),
  freeMem : os.freemem()
};

console.log(currentOS);
```

# NODE JS BUILT-IN MODULES

**PATH** module allow to interact with os path filesystem.

```
const path = require("path"); //get the basename
//platform specific separator
console.log(path.sep)

//joins sequence of paths using os separator
const filePath = path.join("/path",
  "directoryName", "fine.txt");
console.log(filePath); //resolve to get absolute path

const base = path.basename(filePath);
console.log(base);

const absolute =
  path.resolve(__dirname, "contentdir",
  "file.txt");
console.log(absolute);
```

# NODE JS BUILT-IN MODULES

**FS** module allow to interact with os filesystem. It can work in sync and async mode. For now consider these as different methods.

```
const {readFileSync, writeFileSync} = require("fs");
// same of const fs = require("fs");
// fs.readFileSync()
```

```
const first = readFileSync("./file.txt");
const second = readFileSync("./file2.txt");
console.log(first, second);
```

```
//concatenate two file into new one (or append)
writeFileSync(
  "./file3.txt", //path
  first + second, //content
  { //options
    flag : "a" //append
  }
);
```

# NODE JS BUILT-IN MODULES

**FS** module allow to interact with os file system also in async mode.

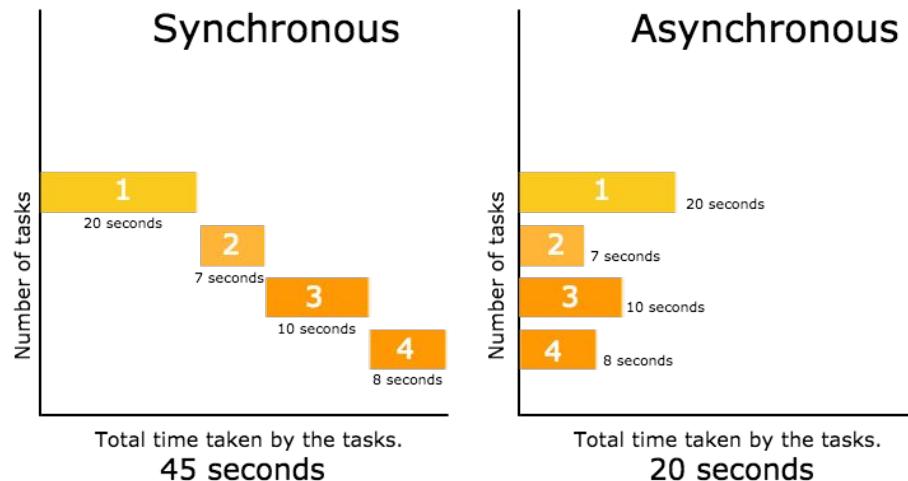
Remember that, like events, we must manage the finish of a read or write.

```
const {readFile, writeFile} = require("fs");
readFile("./file.txt", "utf-8", (err, result) => {
  if (err){
    console.log(err);
    return;
  }
  const first = result;
  readFile("./file2.txt", "utf-8", (err, result) => {
    if (err){
      console.log(err);
      return;
    }
    const second = result;
    writeFile("./file3.txt", first+second);
  });
});
```

# NODE JS SYNC VS ASYNC

Some operation can take long time of elaboration, blocking other tasks.

The basic idea behind Node.js and Async operation is to manage the concurrency via **asynchronous** operations. There are also the Promises and `async/await` methods.



# NODE JS HTTP

Http is one of the built-in Node.js modules that allows to instantiate a http server or client.

```
const http = require("http");

const server = http.createServer((req, res) => {
  res.write("Response");
  res.end();
});

//listen to TCP port 8080
server.listen(8080);
```

This code runs a very basic HTTP server! Try it! However, there is no URL REST logic.

# NODE JS HTTP

You can also improve your path “routing” using the request variable of the callback.

```
const http = require("http");

const server = http.createServer((req, res) => {
  if(req.url === "/"){
    res.end("Home Page")
  }
  if(req.url === "/about"){
    res.end("About")
  }
  res.end("<h1>404</h1>");
});

//listen to TCP port 8080
server.listen(8080);
```

This is a really basic example of REST routing!  
Now before we going forward we need some external modules!

# NODE JS NPM

**NPM** (Node Package Manager) is the Package Manager of Node.js (and not only). This is automatically installed with Node. With NPM you can find everything and you can also upload and share your code.

NPM allows to install packages that are external modules with their dependencies.

Beware, **THERE IS NO QUALITY CONTROL** in NPM, then you could find some broken code or malware!

# NODE JS NPM

- ▶ **npm**  
The global CLI command installed with node
- ▶ **npm <i | install> <package name>**  
Install LOCALLY the package and its dependencies
- ▶ **npm <i | install> -g <package name>**  
Install GLOBALLY the package and its dependencies
- ▶ **npm <u | uninstall> <package name>**
- ▶ **npm init [-y]**  
Initialize the manifest of the project
- ▶ **package.json**  
Manifest file (stores important info about project and packages)

# NODE JS NPM - PACKAGE.JSON

The package.json file is the most important file that defines some important information about the project.

- ▶ The name, description and version of the project
- ▶ The name of the author
- ▶ Some scripts that can me personalised
- ▶ The main of the project
- ▶ List of dependencies and their version
- ▶ List of development dependencies and their version

The package.json also allows to commit and deploy the project without its dependencies installed but you can install these after the first run.

# NODE JS NPM - DEPENDENCIES

When a package is installed via NPM, its information are saved into the package.json file, but where will be saved and how could you use a package?

Until now we just create our modules and loaded them via path file and the built-in modules using their names only.

When a package is installed (for instance locally), in your project a project folder named **node\_modules** is automatically created and the packages are installed into this folder.

Finally, when we need to load a package name (such as bootstrap) we can use only the name of the package (**require("bootstrap")**).

This mechanism allows to differentiate from personal modules and third-party modules. In addition, a package can requires a lot of dependencies and this folder could get really heavy in terms of space disk.

Then, when the project is ready to be committed and deployed, you just upload the entire project without the node\_modules folder and launch the **npm install** to install all dependencies defined into the package.json file (then add node\_modules in your .gitignore!).

# NODE JS EVENTS

When we working with JS on web browsers a big part of our work is the handling events. In the browser most of these events are triggered by some of external interaction, just like button press.

In Node.js we can listen for some specific events and then run some callback on trigger. Also, we can define our events!

Many of built.in modules use the events to work.

To manage and use events we need to use a special class called **EventEmitter** from **events** built-in module.

# NODE JS EVENTS

```
const EventEmitter = require("events");

//custom event emitter
const customEmitter = new EventEmitter();

//listen for specific event
customEmitter.on("response", (name, age) => {
    console.log("data received "+name+" "+age);
});

//you can have multiple callbacks for the same event
customEmitter.on("response", () => {
    console.log("other logic")
});
//the order MATTER

//emit specific event
customEmitter.emit("response", "Mario", 35);
```

# NODE JS HTTP WITH EVENTS

```
const http = require("http");

const server = http.createServer();

server.on("request", (req, res) => {
  res.end("Welcome");
});

server.listen(8080);
```

A lot of modules and built-in modules use EventEmitter and events!

The http modules create a server that is an EventEmitter instance and it has its events, such as request event that is fired when a user make a request to our server.

# NODE JS STREAMS

**Streams** are used to read or write sequentially. When we have to handle and manipulate streaming data, for instance continue source or a big file, streams come in real handy and extends **EventEmitter** class.

There are four different types of streams:

- ▶ **Writeable**  
Used to write data sequentially
- ▶ **Readable**  
Used to read data sequentially
- ▶ **Duplex**  
Used to both read and write sequentially
- ▶ **Transform**  
Data can be modified when writing or reading

# NODE JS STREAMS

These are useful also to manage big files to send as response in http!

```
const {createReadStream} = require("fs");
const stream = createReadStream("./bigfile");

stream.on("data", (result) =>{
    //buffer size is 64kb!
    //this is can be controlled with options
    console.log(result);
});

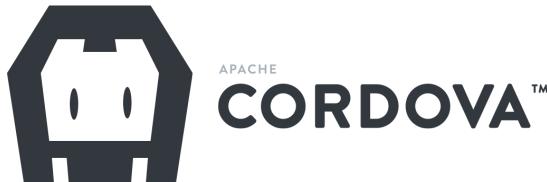
stream.on("error", (err)=>{
    console.error(err);
});
```

```
const http = require("http");
const fs = require("fs");

http.createServer((req, res) => {
    const fileStream =
        fs.createReadStream("./bigfile", "utf-8");
    fileStream.on("open", () => {
        //push from read to write stream
        fileStream.pipe(res);
    });
    fileStream.on("error", (err) => {res.end(err)});
}).listen(8080);
```

# NODE JS TECHNOLOGIES

The publication of Node.js has brought to light different ways of using it, leading to the creation of new technologies.



# NODEJS RECAP!

- ▶ Now you can run JS on your server!
- ▶ It supports non blocking I/O
- ▶ It is Event driven!
- ▶ You can write your modules or use other's modules
- ▶ You can run your code everywhere
- ▶ Not only backend apps but also, you can develop desktop or mobile applications!

You can find a useful cheatsheet at:

<https://gist.github.com/LeCoupa/985b82968d8285987dc3>

# 10.

## EXPRESS.JS

## INTRODUCTION

app.use(cors())

# WHAT IS EXPRESS.JS

Express.js is a minimal and flexible Node.js web application framework, designed to make developing web applications and APIs much faster and easier. Express is built on top of http module but it is not a built-in module but nowadays is a standard for creating web applications.

Also, there are a lot of libraries, modules and entire frameworks for Express.js. It is not comparable to client-side frameworks like React.

# WHY USE EXPRESS.JS

- ▶ Makes building web applications with Node.js much easier
- ▶ Used for server apps as well as API services and Microservices
- ▶ Extremely light, fast and free
- ▶ Full control of requests and responses
- ▶ The most popular framework on Node.js
- ▶ Great to use with client side frameworks

To install this library on your project, just use NPM to install locally:

```
npm install express
```

# EXPRESS.JS SIMPLE SERVER

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

// same of
//const app = require("express")();

app.get("/", (req, res) => {
  res.send("Home Page");
});

app.listen(8080, () => {
  console.log("Server listen on port 8080");
});
```

As for the http module, there you must create an express application instance that has the same method listen to run the http server on a specific port.

But the express application has a lot of methods that could be used.

# EXPRESS.JS APP METHODS

The Express application instance provides few methods within work to make our web application backend. Some of these are related to the HTTP verbs.

- ▶ **get**  
GET requests management
- ▶ **post**  
POST requests management
- ▶ **put**  
PUT requests management
- ▶ **all**  
To manage all types of requests
- ▶ **use**  
This is responsible for **middlewares**  
  
We will introduce the concept of middleware later.

# EXPRESS.JS HTTP COMPARISON

```
const http = require("http");

const server = http.createServer((req, res) => {
  if(req.url === "/"){
    res.end("Home Page")
  }
  if(req.url === "/about"){
    res.end("About")
  }
  res.end("<h1>404</h1>");
});

//listen to TCP port 8080
server.listen(8080);

const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.status(200).send("Home Page");
});

app.get("/about", (req, res) => {
  res.status(200).send("About");
});

// * means everything
app.all("*", (req, res) => {
  res.status(404).send("<h1>404</h1>")
});

app.listen(8080);
```

# EXPRESS.JS BASIC ROUTE HANDLING

- ▶ Handling requests/routes is simple
- ▶ Access to params, query strings, url parts, body, etc
- ▶ Express has a router so we can store routes in separate files and export
- ▶ We can parse incoming data with a parser

```
app.get("/", (req, res) => {
    // Fetch from database
    // Load pages or files
    // Return JSON
    // Access to request and response
});
```

# EXPRESS.JS STATIC FILES

In the previously examples we just return a string, a HTML string or a single file to the client when this one make a GET request (for instance to the root route).

In the reality, a web application (frontend) has a lot of files, such as HTML, CSS, JS, images, etc. files that they are difficult to manage one by one.

In addiction, these file do not change in time but only if the developer will update the frontend code.

In fact, the frontend application can update it self via javascript, then can change the UI and information with the same code without generating it dynamically.

Therefore, we can say that, a typical frontend code and all its assets are “static” files.

# EXPRESS.JS STATIC FILES

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

// we define that a special folder of static files,
// our client
app.use(express.static("./www"));

// * means everything
app.all("*", (req, res) => {
    res.status(404).send("<h1>404</h1>")
});

app.listen(8080);
```

Static files does not mean that your application is not dynamic. A web application is a browser app, then your client Javascript will manage the dynamic aspects of dynamic application.

Static files are typically files that the server does not have to change or generate. To define this we will use **middleware** (we see later).  
Express will manage all request to these static files (routes, etc).

If do you need to generate dynamically your data and then client, you must use a **Server Side Rendering Framework (SSR)**

# EXPRESS.JS API vs SSR

## API

- ▶ JSON
- ▶ Send Data
- ▶ `res.json()`

## SSR

- ▶ Template
- ▶ Send Template
- ▶ `res.render()`

The API architecture is one of the most used developing a backend software. With API all data will sent in JSON type and then is compatible with all environments.

Another method is that to generate dynamically an entire client to each request, such as PHP with Laravel, using the render of Express with a render engine. The last one is one of the most complicated and not really picked up for the backend development nowadays.

# EXPRESS.JS ROUTE PARAMS

Sometimes, the client want to access to a specific resource or maybe delete or update it (get information about a specific product). To do this, we need to access to the **route parameters**.

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

//not dynamic solution
app.get("/api/products/1", (req, res) => {
  res.json(products[0]);
});

//dynamic solution
app.get("/api/products/:productID", (req, res) => {
  //params are all strings
  //same of const req.params.productID
  const {productID} = req.params;
  res.json(products[Number(productID)]);
});
```

app.get("/api/products/:productID/reviews/:reviewID", (req, res) => {  
 ...  
});

To access to these params we have to use the **request** variable its **params** attribute.

There we will find all params that are defined in the url route.

In the url route you can define the name of params with the syntax **:paramName**.

# EXPRESS.JS QUERY PARAMS

In addition, the client would send more information about that resource (remember the search on Google?). To do this, we need to access to the **query parameters (or URL parameters)**.

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

app.get("/api/v1/search", (req, res) => {
  const {search, limit} = req.query;
  let sortedProducts = [...products];
  if(search){
    sortedProducts = sortedProducts
      .filter((product)=> product.name.startsWith(search));
  }
  if(limit){
    sortedProducts = sortedProducts.slice(0, Number(limit));
  }
  if(sortedProducts.length<1){
    res.status(200).json({success:true, data:[]} );
  }
  res.status(200).json({success:true, data:sortedProducts});
});
```

To access to these params we have to use the **request** variable its **query** attribute. There we will find all params that are defined in the url query.

# EXPRESS.JS MIDDLEWARES

Middlewares are functions that execute during the request to the server. Each middleware function has access to request and response objects (and more stuff). In express everything is a middleware, it is the heart of express.

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

// req => middleware => res
const logger = (req, res, next) => {
    // log the request
    console.log(req.method, req.url);
    // then i can go to the next middleware
    next();
};

app.get("/", logger, (req, res) => { res.send("Home")});
app.get("/about", logger, (req, res) => { res.send("About")});

app.listen(8080);
```

A middleware sits between a **request** and **response**. Also a middleware has a reference to the **next** middleware and must be ever called. You can stack unlimited number of middlewares!

# EXPRESS.JS USE

Middlewares are a powerful tool within we can add some behaviors based on route and methods, such as manipulation or check of data in input or user authentication. But append these function in our single routes could be a massy. We need something that apply a middleware to all routes!

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

// req => middleware => res
const logger = (req, res, next) => {
  ...
};

app.use(logger);

app.get("/", (req, res) => { res.send("Home")});
app.get("/about", (req, res) => { res.send("About")});

app.listen(8080);
```

The **use** method allows to append a middlewares based on its invocation position to all routes and other middlewares!  
You can also define a path!

```
app.use("/api", logger);
```

# EXPRESS.JS POST

By default you can not access to the body (payload) of a POST request. You must set and use a middleware of express that parses the request body to something readable, such as JSON. This is the **urlencoded** middleware (the old bodyParser) and the **json** middleware.

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

app.use(express.static("./www"));

//parse form data
app.use(express.urlencoded({extended:false}));
app.use(express.json());

app.post("/login", (req, res) => {
  const {email} = req.body;
  if(email){
    return res.status(200).send("Welcome back "+email);
  }
  return res.status(401).send("Email must be sent");
});
```

The flag extended : false needs to force the parsing with body-parser.  
This works also for PUT and DELETE methods.

# EXPRESS.JS ROUTER

Router allows the developers to group the request routes by some logic. Also, they are used typically in the MVC pattern.

```
//router.js
const express = require("express");
const router = express.Router();

router.get("/peoples", (req, res) => { res.send(peoples)});
router.get("/peoples/:id", (req, res) => { res.send(peoples[req.params.id])});

module.exports = router;

//app.js
const express = require("express");
const app = express();
const mainRouter = require("./router");

app.use(express.static("./www"));

//parse form data
app.use(express.urlencoded({extenstion:false}));
app.use(express.json());

//load router
app.use("/api", mainRouter);
```

The usage of routers allow to organize request routes in groups and categories! In this way, you can expand your backend more easily. Also you can implement the MVC Controllers and Models (them are middlewares!).

# EXPRESS RECAP!

- ▶ It is a lightweight framework
- ▶ It allows to manage requests and responses
- ▶ It allows to manage all types of requests
- ▶ It uses middlewares!
- ▶ It can serve static files
- ▶ It is expandable with a lot of modules!

You can find a useful documentation at:

<https://expressjs.com/>

# 11.

## WebSockets & Socket.IO

Real time Apps!

# WHAT ARE WEBSOCKETS

When a user surfs the web, the browser (client) sends certain requests via REST routes ('GET', 'POST', etc) to the server that hosts the website they are trying to access. The server receives the requests and sends information as responses back to the client, which receives and renders the response information on the page. If you need to update some information, you must do several requests to the server in a interval time. This approach is called **Polling**.

**WebSockets** are different because they work by holding the connection from the client to the server open at all times. This way, the server can send information to the client, even in the absence of an explicit request from the client. Clients can still make HTTP requests to the server like normal; however, the WebSocket connection allows for constant communication in the case of new data being created and therefore needing to be rendered client-side.

# HOW WEBSOCKETS WORKS

A connection starts with a HTTP handshake that request an upgrade for the protocol, this is called WebSocket Handshake request. The handshake starts with an HTTP request/response, allowing servers to handle HTTP connections as well as WebSocket connections on the same port. Once the connection is established, communication switches to a bidirectional binary protocol which does not conform to the HTTP protocol. In addition to **Upgrade** headers, the client sends a **Sec-WebSocket-Key** header containing base64-encoded random bytes, and the server replies with a hash of the key in the Sec-WebSocket-Accept header. This is intended to prevent a caching proxy from re-sending a previous WebSocket conversation, and does not provide any authentication, privacy, or integrity.

Once the connection is established, the client and server can send WebSocket data or text frames back and forth in full-duplex mode. The data is minimally framed, with a small header followed by payload. WebSocket transmissions are described as "**messages**", where a single message can optionally be split across several data frames

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMBDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

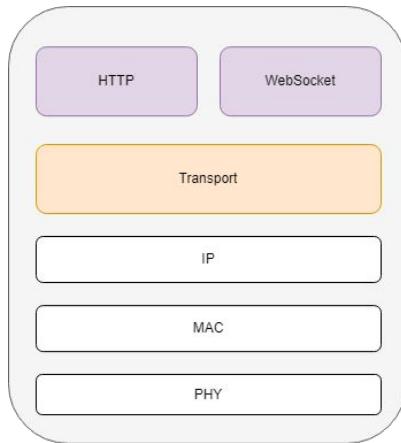
```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
HSmrC0sM1YUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

# USING WEBSOCKET CLIENT

WebSockets work at Application layer and can work without a specific server. In fact, a WebSocket server can run using an existing HTTP server or standalone.

To define a WebSocket connection you have to define an URL using the **ws://** protocol, instead of **http://**.

Like HTTP, WebSockets support the SSL encryption and you can specify that using the protocol **wss://**.



```
var connection = new WebSocket('ws://html5rocks.websocket.org/echo')
// When the connection is open, send some data to the server
connection.onopen = function () {
    connection.send('Ping'); // Send the message 'Ping' to the server
};

// Log errors
connection.onerror = function (error) {
    console.log('WebSocket Error ' + error);
};

// Log messages from the server
connection.onmessage = function (e) {
    console.log('Server: ' + e.data);
};
```

Attaching some event handlers immediately to the connection allows you to know when the connection is opened, received incoming messages, or there is an error.

# USING WEBSOCKET SERVER

**ws** is a popular library for using WebSockets in Node.js applications. To integrate WebSockets in your application, first, you need to install the ‘ws’ library from npm.

```
//stand alone server
const ws = require('ws')

const connection = new ws.Server({ port: 8080 })

connection.on('connection', ws => {
  ws.on('message', message => {
    console.log(`Received message => ${message}`);
    ws.send('Message From Server');
  });
});

//with express server
const express = require('express');
const ws = require('ws');

const app = express();

// Set up a headless websocket server
const wsServer = new ws.Server({ noServer: true });
wsServer.on('connection', socket => {
  socket.on('message', message => console.log(message));
});

// `server` is a vanilla Node.js HTTP server

const server = app.listen(3000);
server.on('upgrade', (request, socket, head) => {
  wsServer.handleUpgrade(request, socket, head, socket => {
    wsServer.emit('connection', socket, request);
  });
});
```

# SOME WEBSOCKET PROBLEMS

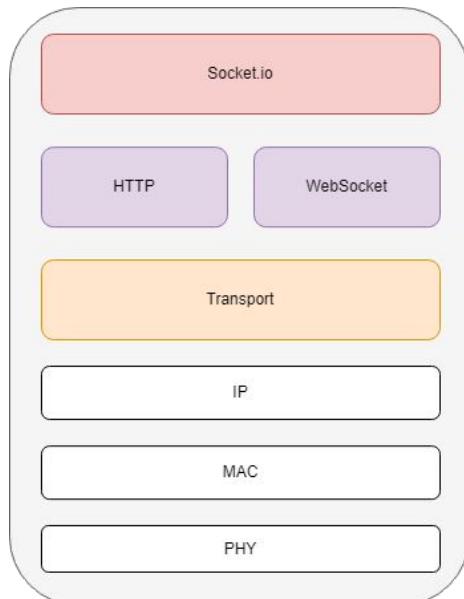
This protocol is really young but its usage on Web Application and Internet of Things applications increased in the last years. Nevertheless, there are some problems with WebSockets:

- ▶ Not meant for server-to-server communication, maybe some low level UDP socket is preferred
- ▶ Not meant for client-to-client (p2p) communication, for that there is WebRTC!
- ▶ If you are using proxies, these must be compatible with WebSockets
- ▶ Server must supports the WebSockets and its handshake
- ▶ The backend language must supports the WebSockets
- ▶ BROWSER MUST support the WebSockets (not all browsers support ws)

Then, we need something that allows the development of a real-time full-duplex communication without these problems!

# WHAT IS SOCKET.IO

Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server. It is built on top of the WebSocket protocol and provides additional guarantees like fallback to HTTP long-polling or automatic reconnection.



The bidirectional channel between the Socket.IO server (Node.js) and the Socket.IO client (browser, Node.js, or another programming language) is established with a WebSocket connection whenever possible, and will use HTTP long-polling as fallback.

The Socket.IO codebase is split into two distinct layers:

- ▶ the low-level plumbing: what we call **Engine.IO**, the engine inside Socket.IO
- ▶ the high-level API: **Socket.IO** itself

# HOW SOCKET.IO WORKS

**Engine.IO** is responsible for establishing the low-level connection between the server and the client. It handles:

- ▶ The various **transports** (WebSockets or HTTP long-polling)
- ▶ The **upgrade** mechanism
- ▶ The **disconnection** detection

By default, the client establishes the connection with the HTTP long-polling transport. While WebSocket is clearly the best way to establish a bidirectional communication, experience has shown that it is not always possible to establish a WebSocket connection, due to corporate proxies, personal firewall, etc.

To summarize, Engine.IO focuses on reliability and to upgrade, the client will:

- ▶ ensure its outgoing buffer is empty
- ▶ put the current transport in read-only mode
- ▶ try to establish a connection with the other transport
- ▶ if successful, close the first transport

The Engine.IO connection is considered as closed when:

- ▶ one HTTP request (either GET or POST) fails (for example, when the server is shutdown)
- ▶ the WebSocket connection is closed (for example, when the user closes the tab in its browser)
- ▶ `socket.disconnect()` is called on the server-side or on the client-side

There is also a heartbeat mechanism which checks that the connection between the server and the client is still up and running:

# SOCKET.IO INSTALLATION (SERVER)

To install the Socket.io server you have to use NPM to install it:

```
npm install socket.io
```

By default, Socket.IO use the WebSocket server provided by the ws package.

There are 2 optional packages that can be installed alongside this package. These packages are binary add-ons which improve certain operations. Prebuilt binaries are available for the most popular platforms so you don't necessarily need to have a C++ compiler installed on your machine. To install them run the NPM command:

```
npm install --save-optional bufferutil utf-8-validate
```

In addition, you can install and use any WebSocket server implementation which exposes the same API as ws module.

# SOCKET.IO USAGE (SERVER)

The Server instance (often called io) has a few attributes that may be of use in your application.

```
//stand alone server
const { Server } = require("socket.io");

const io = new Server({ /* options */ });

io.on("connection", (socket) => {
  // ...
});

io.listen(3000);

//with http server
const { createServer } = require("http");
const { Server } = require("socket.io");

const httpServer = createServer();
const io = new Server(httpServer, { /* options */ });

io.on("connection", (socket) => {
  // ...
});

httpServer.listen(8080);

//with express
const express = require("express");
const { createServer } = require("http");
const { Server } = require("socket.io");

const app = express();
const httpServer = createServer(app);
const io = new Server(httpServer, { /* options */ });

io.on("connection", (socket) => {
  // ...
});

httpServer.listen(8080);
```

# SOCKET.IO SOCKET

A Socket is the fundamental class for interacting with the client. It inherits all the methods of the Node.js **EventEmitter**, like emit, on, once or removeListener.

```
// Each new connection is assigned a
// random 20-characters identifier.

// server-side
io.on("connection", (socket) => {
  console.log(socket.id); // ojIckSD2jqNzOqIrAGzL
});

// client-side
socket.on("connect", () => {
  console.log(socket.id); // ojIckSD2jqNzOqIrAGzL
});

// on creation, the Socket joins the room identified by its own id
// which means you can use it for private messaging:
io.on("connection", socket => {
  socket.on("private message", (anotherSocketId, msg) => {
    socket.to(anotherSocketId).emit("private message", socket.id, msg);
  });
});
```

The idea is that to emit and listen for some events that developer can define. For instance, in the client and in the server could define an event called “onMessage” and use this one to exchange messages, emitting this event with a payload!

# SOCKET.IO BROADCASTING

Broadcasting means sending a message to all connected clients. Broadcasting can be done at multiple levels. We can send the message to all the connected clients, to clients on a namespace and clients in a particular room.

To broadcast an event to all the clients, we can use the **io.sockets.emit** method.

```
io.on('connection', (socket) => {
  clients++;
  io.sockets.emit('broadcast', { description: clients + ' clients connected!' });
  socket.on('disconnect', () => {
    clients--;
    io.sockets.emit('broadcast', { description: clients + ' clients connected!' });
  });
});
```

Now, if we want to send an event to everyone, but the client that caused it (in the previous example, it was caused by new clients on connecting), we can use the **socket.broadcast.emit**.

```
io.on('connection', (socket) => {
  clients++;
  socket.emit('newclientconnect', { description: 'Hey, welcome!' });
  socket.broadcast.emit('newclientconnect', { description: clients + ' clients connected!' })
  socket.on('disconnect', () => {
    clients--;
    socket.broadcast.emit('newclientconnect', { description: clients + ' clients connected!' })
  });
});
```

# SOCKET.IO NAMESPACES

Socket.IO allows you to "namespace" your sockets, which essentially means assigning different endpoints or paths. This is a useful feature to minimize the number of resources (TCP connections) and at the same time separate concerns within your application by introducing separation between communication channels. Multiple namespaces actually share the same WebSockets connection thus saving us socket ports on the server.

The root namespace '/' is the default namespace, which is joined by clients if a namespace is not specified by the client while connecting to the server. All connections to the server using the socket-object client side are made to the default namespace.

```
const nsp = io.of('/my-namespace');
nsp.on('connection', function(socket){
  console.log('someone connected');
  nsp.emit('hi', 'Hello everyone!');
});
```

Now, to connect a client to this namespace, you need to provide the namespace as an argument to the io constructor call to create a connection and a socket object on client side.

# SOCKET.IO ROOMS

Within each namespace, you can also define arbitrary channels that sockets can join and leave. These channels are called **rooms**. Rooms are used to further-separate concerns. Rooms also share the same socket connection like namespaces. One thing to keep in mind while using rooms is that they can only be joined on the server side.

```
const roomno = 1;
io.on('connection', function(socket){
  socket.join("room-"+roomno);
  //Send this event to everyone in the room.
  io.sockets.in("room-"+roomno).emit('connectToRoom',
"You are in room no. "+roomno);
})
```

You can call the join method on the socket to subscribe the socket to a given channel/room. For example, let us create rooms called 'room-<room-number>' and join some clients. As soon as this room is full, create another room and join clients there.

You can also implement this in custom namespaces in the same fashion. To leave a room, you need to call the leave function just as you called the join function on the socket.

# SOCKET.IO CHAT EXAMPLE - SERVER

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.use(express.static("./www"));
app.use(express.urlencoded({extended:false}));
app.use(express.json());

const users = [];

io.on('connection', function(socket){
  console.log('A user connected');
  socket.on('setUsername', function(data){
    console.log(data);
    if(users.indexOf(data) > -1){
      socket.emit('userExists', data + ' username is taken! Try some other username.');
    } else {
      users.push(data);
      socket.emit('userSet', {username: data});
    }
  });
  socket.on('msg', function(data){
    //Send message to everyone
    io.sockets.emit('newmsg', data);
  })
});
http.listen(3000, function(){
  console.log('listening on localhost:3000');
});
```

# SOCKET.IO CHAT EXAMPLE - CLIENT

```
const socket = io();
function setUsername(){
    socket.emit('setUsername', document.getElementById('name').value);
};

let user;
socket.on('userExists', function(data){
    document.getElementById('error-container').innerHTML = data;
});
socket.on('userSet', function(data){
    user = data.username;
    document.body.innerHTML = '<input type="text" id="message">\\
    <button type="button" name="button" onclick="sendMessage()">Send</button>\\
    <div id="message-container"></div>';
});
function sendMessage(){
    const msg = document.getElementById('message').value;
    if(msg){
        socket.emit('msg', {message: msg, user: user});
    }
}
socket.on('newmsg', function(data){
    if(user){
        document.getElementById('message-container').innerHTML += '<div><b>' + data.user + '</b>: ' + data.message + '</div>'
    }
})
})
```

# SOCKET.IO INSTALLATION - CLIENT

There are several implementations of Socket.IO client for any languages. The main JS client is just a library that you can download or load from CDN.

For list of CDN or library download:

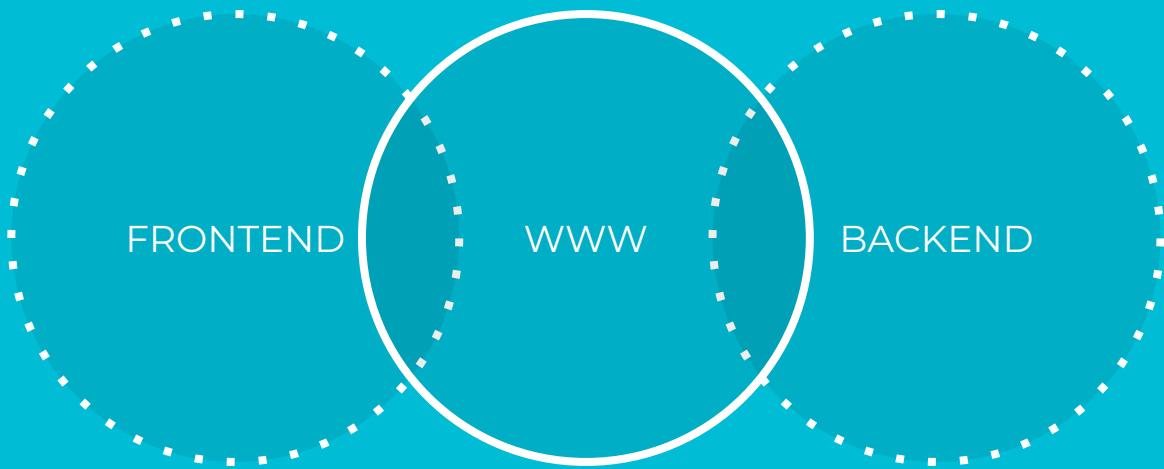
<https://socket.io/docs/v4/client-installation/>

# WS & SOCKET.IO RECAP!

- ▶ WebSockets allow to develop real-time web applications
- ▶ Are built on top of transport protocol and use HTTP for the handshake
- ▶ WebSocket could not implemented or supported by languages
- ▶ Socket.IO is a library that is build on top of HTTP and WebSockets
- ▶ Socket.IO is based on EventEmitter
- ▶ It provides a lot of functionalities, such as broadcasting and rooms!

Now you can try to build your real time chat!

**WEB  
DEVELOPMENT  
COURSE  
ENDS**





# THANKS!

## Any questions?

You can find me at @fedyfausto or federico.santoro@unict.it