**Become a Front End Developer**

**with Altimetrik | 2nd Edition**

**Documentation**

[Michael Armesto]

Hey there ~ This is Michael Armesto personal documentation. A little bit about me, I am a twenty-six-year-old IT student. I started just a few months ago to dive into this techy world and I am loving it.

# Index

**Agiles methodologies** ---------------------------------------------------------------------------------- 4

**GIT**

Version Control System —----------------------------------------------------------------------------- 7

Git commands—---------------------------------------------------------------------------------------- 8

Branching Strategy —---------------------------------------------------------------------------------- 11

Rebase / Squash—------------------------------------------------------------------------------------ 13

**Javascript**

Introduction —-------------------------------------------------------------------------------------------- 15

Getting started —------------------------------------------------------------------------------------------ 16

Syntax & Basic construct —----------------------------------------------------------------------------- 17

Scope —------------------------------------------------------------------------------------------------------ 21

Hoisting —---------------------------------------------------------------------------------------------------- 24

Strict mode —----------------------------------------------------------------------------------------------- 26

**HTML**

What is HTML —------------------------------------------------------------------------------------------- 27

Tags types —----------------------------------------------------------------------------------------------- 28

Inputs —----------------------------------------------------------------------------------------------------- 30

XHTML vs HTML —-------------------------------------------------------------------------------------- 32

Data-attributes —----------------------------------------------------------------------------------------- 35

Accessibility —--------------------------------------------------------------------------------------------- 36

Semantic HTML —---------------------------------------------------------------------------------------- 40

Meta tags & SEO —-------------------------------------------------------------------------------------- 45

SEO —------------------------------------------------------------------------------------------------------- 47

Web crawlers —------------------------------------------------------------------------------------------- 49

Geotagging —---------------------------------------------------------------------------------------------- 52

DOM —------------------------------------------------------------------------------------------------------- 53

How to manipulate DOM with JS —------------------------------------------------------------------- 56

**CSS**

CSS & Selectors —---------------------------------------------------------------------------------------- 60

Specificity —------------------------------------------------------------------------------------------------- 62

Linking CSS —---------------------------------------------------------------------------------------------- 64

Critical CSS, loaders, spinners, skeleton screens —--------------------------------------------- 65

CSS reset, normalize.css —---------------------------------------------------------------------------- 68

Box model, Box-sizing —------------------------------------------------------------------------------ 70

**Agile Methodology**

To get a better understanding of this term we need to go back in time a little. Back in the days there was an approach (on some work areas still exist) called “**Waterfall**” // not related to the TLC’s song // we can say that this is more of a linear mindset of working where phases flow downwards and requires a project to be finished for the client to get a look on it and judge it. ***It focuses more on the successful delivery of the project.*** On the other hand, we have this other mindset to approach projects that is called “**Agile** ''. If we take a look at the definition of this word we are going to conclude that agile - *is something or someone that is able to move quickly and easily.*

The last approach mentioned focuses more on people, results, collaboration and flexible responses to change. Instead of planning for the whole project, it breaks down the development of the project into **sprints** with a **focus on the client's satisfaction.**

Sprint - A *sprint* in [Scrum](https://www.atlassian.com/agile/scrum) is a short period of time where a development team works to complete specific tasks. Breaks down the project into shorter blocks of time in which smaller goals can be accomplished.

Sprint Planning - Is an event in scrum that kicks off the sprint. The purpose of sprint planning is to define what can be delivered in the sprint and how that work will be achieved. Sprint planning is done in collaboration with the whole scrum team.

*// Product owner / manager - is a key actor in the development of the project, needs to have a clear vision of what is being built. //*

Daily Scrum - Also known as [*stand-up meetings*](https://www.workfront.com/project-management/methodologies/agile/daily-stand-up) it typically lasts no more than 15 minutes // standing up // and requires each team member to discuss what they have accomplished since the last meeting, what they will work on before the next meeting, and if any obstacles are standing in or blocking their way.

Sprint Retro & review - These both concepts may seem similar but they have completely different purposes. To summarize them, the sprint review is about the product, while the sprint retrospective is about the team. **While the *sprint review* helps you to regularly meet customer expectations, *retrospectives* allows scrum teams to become faster, smarter, and even happier in the next sprint.**

The Sailboat Retrospective - It's a visual way for your team to identify what pushed the project forward, as well as what held it back.

Grooming / Refinement - Helps **PO** to explain and understand the strategy behind the items in the backlog. *// It helps to get the whole team to be in the same boat //* At the end of this session the list of *user stories* will be prioritized. // this means that at the top of the list the user stories will be way more detailed getting more vague as the list goes down // The ideal length of these sessions is something around 45 minutes to 1 hour.

Velocity - **Is the amount of work a team can deliver during a sprint.** During *Sprint planning*, a team’s velocity is used to determine the number of product backlog items to tackle. This is calculated by taking the average story points of completed user stories for the last ~usually~ three sprints.

Agile estimation - Is just that, an estimation of the effort required to complete a task. It is a team sport. This means that it involves everyone. Each team member brings a different perspective on the product and the work required to deliver a user story.

Relative & absolute sizing/estimation - Relative means that we estimate how big or how much effort it will take a task to complete in relation to another task. The most used units of estimation are the ***Story Points.*** Whereas Absolute Sizing we estimate each item individually without comparing it.

Planning Poker -The most popular technique of gross level estimation is *Planning Poker* to provide an estimate in the form of a point value for each item. For agile estimation purposes, the numbers of the cards are: 1, 2, 3, 5, 8, 13, 20, 40, 100.



Story points - Is a number used to express the effort required to complete a task. This system takes into account the **amount of work** to do *(the folding paper example),* **Risk & Uncertainty**, (if what needs to be done is unclear then that uncertainty should be taken into account in the estimate), **Complexity** should be considered when providing a story point estimate *(folding paper airplane example).*

User Story - It’s an end goal. a few sentences in *simple language* that outline the desired outcome. User stories tell you what features and functions a product should have and should be a vocabulary that everyone understands to avoid misunderstandings. *User stories* describe the requirements and wishes for a project result to be created from the perspective of the customer.

*Structure of a User Story* =>

**WHO** (*role*), wants **WHAT** goal / wish **WHY** (*added value*)

*User stories* are added to sprints and *“burned down”* over the duration of the sprint. User stories are also the building blocks of larger agile frameworks like **epics** and **initiatives**.

Epics are large work items broken down into a set of stories, and multiple epics make an initiative.

Burndown chart - is a graphical representation of work left to do versus time. It is useful for predicting **when** all of the work will be completed. It helps monitoring the project scope creep as well as keeping the team running on schedule.

Spike - A Spike is **created when a user story or task cannot be estimated well enough until the team has done further research or investigation**.

**Version Control System**

**V**ersion **C**ontrol **S**ystem (**VCS**) records all the changes made to a file, set of files, projects, so a specific version may be called later if needed. The system makes sure that all the team members are working on the latest version of the file and more importantly all members can work simultaneously on the same project.

It allows us to revert selected files back to a previous state or even a whole project, if something is not working anymore, we can go back and recover it.

There are ***three*** types of *VCS*:

**Local VCS:** It is one of the simplest ways of keeping all the changes in a local database located on your own computer. One of the disadvantages of this is that you can easily forget the directory you are in and overwrite the wrong code for example.

**Centralized VCS:** In this case a remote server acts as the main repository storing every version of the project and only allows one person to contribute to the code at a time. The major disadvantage is that if the server goes down, no one can work on the project.

[**GIT**](https://git-scm.com/) is a **D**istributed **V**ersion **C**ontrol **S**ystem (**DVCS**) - On this one you get the complete codebase and its full version history mirrored or cloned on every team member’s computer, this copy has the full metadata of the original. // if something catastrophic happens, each team member’s local copy becomes a backup // Making it possible to ***commit***, ***branch*** and ***merge*** locally.The server does not have to store a physical file for each branch, just the differences between each commit made.

DVCS does not need internet connection except from ***pulling*** and ***pushing.***

*~ Let's take a look at some* ***GIT*** *commands and concepts ~*

git init - Is a one-time command. This command will create a new .git subdirectory in your current working directory and also a new main branch.

git clone [URL] - ***Retrieve*** // to locate and bring/copy/clone // an entire repository from a hosted location via URL.

// Uniform Resource Locator. A URL is nothing more than the address of a given unique resource on the Web. //

git status - show modified files in working directory, staged for your next commit

git add [file]- add a file as it looks now to your next commit. // a dot (git add .) adds all files. //

Git commit -m “[descriptiveMessage]” - commit your staged content as a new commit snapshot.

// **A commit is a snapshot in time**. Each commit contains a pointer to its root tree, representing the state of the working directory at that time. A commit with no parents is a **root commit** and a commit with multiple parents is a **merge commit.**//

git branch - list your branches. a \* will appear next to the currently active branch.

Git branch [branchName] - ***create*** a new branch at the current commit.

Git checkout - **switch** to another branch and check it out into your working directory.

Git merge [branch] - merge the specified branch’s history into the current one.

git log - show all commits in the current branch’s history.

git pull - Is used to ***fetch*** // to call for // and ***download*** content from a remote repository and immediately ***update*** the local repository to match that content.

git push - Is used to ***upload*** local repository content to a remote repository. Is how you transfer ***commits*** from your local repository to a remote repository.

git tag - This command is used to mark specific points in the repository’s history, usually release points *(v1.0, v1.1).* A tag once being created, does not change, it does not have a commit history, unlike branches.

* git tag -l / –list => list the tags in alphabetical order // no real importance in the order displayed //

There are two types of tags ***lightweight*** and ***annotated***:

Lightweight - It is just a pointer to a specific commit. To create one, all we need to provide is a tag name. We do not need to include any of the flags we will see with annotated tags below.

$ git tag v2.3-lightweight

$ git tag

v0.1

v0.5

v2.3-lightweight

V2.4

Running git show will display the commit **SHA**(***S****imple* ***H****ashing* ***A****lgorithm*), author, date and commit message.

$ git show v2.4-lightweight

commit ca82a6dff817ec66f44342007202690a93763949

Author: Emma Wedekind <wedekind.emma@gmail.com>

Date: Sat Mar 16 09:48:93 2918 -0700

// when we make a commit to save our work, git creates a unique ID (a.k.a. the "**SHA**" or "**hash**") that allows you to keep record of the specific changes along with who made them and when.

git's *checksums* include metadata about the commit including the author, date, and the previous commit SHA. //

git stash - // put aside // It temporarily saves the changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on.

**Note**: The stash is local to your git repository; stashes are not transferred to the server when you push.

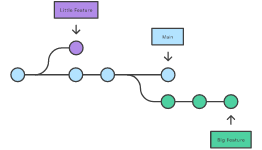
git hooks - These are scripts that Git automatically executes before or after events like ***commit***,***push***, or ***receive***. A case use could be enforcing a certain commit policy for example, if you want commit messages to have a minimum of fifty characters or to ***lint*** the code before it gets committed. // git hook can be thought of as a developer workflow tool for increasing productivity and enforcing best practices //

Git hooks can be classified in two:

client-side/local hook - Affects only the repository in which they are located.

server-side hook - These work just like local ones but they are located on the network hosting repository and they are prompted by events such as receiving *pushes* for example.

**Branching Strategy**

A branch is an independent line of development. When we want to add a new feature or fix a bug // no matter how big or how small // A new branch is created to encapsulate changes. These branches are usually merged back to a master branch once the work is finished.

**Main**

**Little feature**

**Big feature**

// By developing two features/code development in different branches, it’s not only possible to work on both of them in parallel, but it also keeps the main branch free from questionable code. We can see it as a kind of protection to the mainline //

Then we can say that a branching strategy is the *game plan* / *approach* that teams adopt when coding using a VCS. It is vital to have a set of rules that developers can follow to guarantee a correct interaction with a shared codebase.

A common strategy used is ***Gitflow*** which - *as its name suggests* - is a **workflow** applied to git.

// **workflow** is a way of organization of the tasks and activities that must be done for a project to be finished. //

It is ideal for projects that carry iterative delivery planning. It allows *parallel* development through independent branches such as ***master***, ***develop***, ***feature***, ***hotfix***, ***release*** and ***bugfix***.

master - as the *main/master* branch contains the code developed and every stable version released.

develop - Contains the code in development of the next planned version released (*new features*). From this branch the featurebranches are born. The develop branch will be merged to master through the branch releases.

feature - These are the branches that we will be working on usually. As default it contains the prefix feature followed by the branch name. These branches are always born from the develop branch and die when they are merged.

// typically they exist on the local repository of developers //

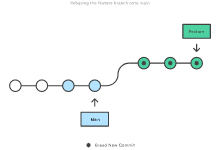
hotfix - Are the branches that arise from a bug and must be urgently resolved. // is like a bugfix but under pressure // It allows developers to keep working on their own changes while the bug is being fixed. These are born from the master branch and merged to master and develop to fix the bug.

release - Branches that we will use to create new versions to deploy a production. Is the mechanism through which we merge the new developments that we have in development to master.

bugfix - The idea behind this branch is that a bugfix can be any bug fix in a develop or release branch. // differs from hotfix; is specific for a fix to be directly applied to master //

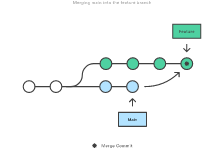
**Rebase & Squash**

git rebase - Rebase is a command that allows you to copy commits from one branch to another. In other words, *re-committing all commits of the current branch onto a different base commit.*The major benefit of rebase is that you get a much cleaner project history, resulting in a perfectly linear project history—*you can follow the tip of feature all the way to the beginning of the project without any forks.* This makes it easier to navigate your project with commands like git log, for example.

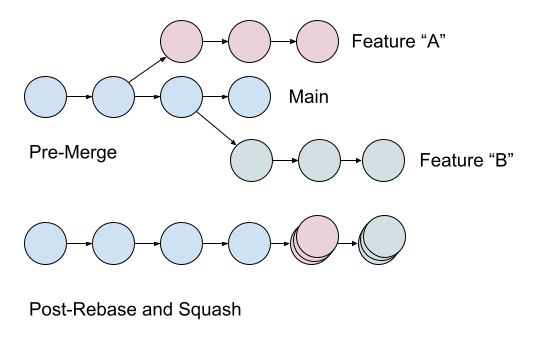
// The goal of a rebase is to change the start (or base) of a branch //

The golden rule of git rebase is to never use it on *public* branches.

merge - A commit that combines all changes of a different branch into the current. Merge is a non-destructive operation. But this also means that the feature branch could end up polluted and make it hard to read/understand.



squash - we can use squash to merge all of our commits from a feature branch into a single commit, which can then be added to the end of the main branch.



There is no right or wrong in when to use each option, to some extent it is preferential but as a general rule when merging a pull request from a feature branch with a messy commit history, squash should be applied to reorganize and have a clearer and easier to read git history.

// if a change took you x amount of commits and you do not want the people to know how many commits it took, squash can be applied //

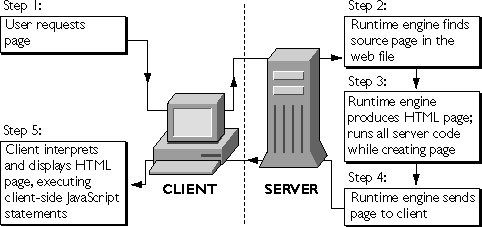
**Javascript Introduction**

*Javascript*, from now on **JS**, is a dynamic programming language. Basically you can build/create as much as your imagination **AND** knowledge of the language allows you.

It is well-known for web page creation, but it is widely used in numerous non-browser applications.JavaScript can be used for **c**[**lient-side**](https://www.geeksforgeeks.org/server-side-client-side-programming/) developments as well as **s**[**erver-side**](https://www.geeksforgeeks.org/server-side-client-side-programming/) developments.

**Client-side** means the browser is who processes the code (*the client’s device*) instead of the web server. This is used when we need to validate data before sending it to the web server, and respond to user events such as mouse clicks, form input, and page navigation among others. This is the *Front-End* side, what is visible to the user, client.

**Server-side** means that the processing takes place on the web’s server. This is the *Back-end* side and enables access to databases, file systems, and servers. The most used framework is *node.js.*



*To become a front-end developer is the aim of this program, so I am going to stick with the client-side.*

**Getting started**

There are **two** main methods that we can use to implement JS to our projects, hence, HTML files: **Internal** and **External**.

**Internal**, the script is written inside our **HTML** file in-between it’s corresponding tag

| *<script>  our set of instructions </script>* |
| --- |

**External**, we link our **JS** file in our **HTML** file. Using the same tags as the internal method but giving them an attribute src - It means *source* and contains the *location* of our **JS** file.

| <script src="OurJSFile.js"></script> |
| --- |

*This way our files are in separate folders resulting in a more organized project and a more easy-to-read code.*

The script tag (linking our JS file) should be at the bottom of our **HTML** file, just right above our closing body tag *</body>* to ensure that our **HTML** elements and **CSS** style are fully loaded before our **JS** file is.

| <script src="js/script.js"></script>  </body> </html> |
| --- |

Now we are ready to start writing **JS** but we need to understand some fundamentals

Syntax and Basic Constructs

*Syntax* refers to the set of rules that determines how **JavaScript** programs/code are constructed or written.

#### **1. Every statement should end with a semicolon - ;**

This good practice helps to understand that we are done with that statement and to avoid errors such as concatenate two statements for example.

let statement = prompt('end with a semicolon');

// concatenate: link things together in a chain or series //

**2. Statements -** are instructions that the browsers are going to execute

let statementOne = 1;

let statementTwo = 2;

let statementThree = statementOne + statementTwo;

alert("This is the statement 4!!");

Every line is a statement and ends with a semicolon.

**3. Comments -** Helps developers to document their codes, explain or give further information, the purpose of certain code and so on as well as team’s members to understand the intentions behind that code.

Comments are typed in-between double slash forward lines

// This is a comment //

**4. Whitespaces** - JS ignores whitespaces - This is the main reason a statement should end up with a semicolon. To visualize clearly when a statement starts and ends.

**5. Variables** - Variables are like containers used for saving values, data. Instead of repeating a value for different uses, you could just assign it to a variable.

let total = 0;

*For example, I am saying here that the variable total has an initial value of zero.*

In JS the keywords to declare a variable are the followings var and let:

var = used in the older versions of JS and with the implementation of 'let' it is considered a bad practice to use.

let = is the new way of declaring variables starting ES6 (ES2015) and is block scoped. // it'll be developed in depth in the following pages //

Variables names must start with a letter, underscore ( \_ ) or a dollar sign ($)

**CANNOT START WITH A NUMBER.**

**let a = 'hello';**

**let \_a = 'hello';**

**let $a = 'hello';**

**let 5a = 'hello'; // this gives an error**

There is another keyword const // introduced in ES62015 // whose value cannot be reassigned.

const constant = ES62015;

We cannot declare a constant without initializing

*const JS ; // ERROR - const declarations must be initialized. //*

*JS = 'Javascript';*

**6. JS Keyword:** The keywords are the reserved words that have special meaning in JavaScript.

var and let are the keywords used to define a variable.

function is the keyword which tells the browser to create a *function*.

And many more ~

**7. JS Functions:** Are groups of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again.

| function functionName(parameter, list, separated, by, commas) {    statements  statements  statements  }; |
| --- |

To invoke a function later in the script, we will simply need to write the name of that function as shown in the following code.

| functionName(); |
| --- |

**8. Case sensitive** - as it says we could have two variables

const number = 7;

const Number = 5;

These two variables are different from each other despite being the same word.

### 

### **9. Statements Versus Expressions** - **JS** has two major syntactic categories: *statements* and *expressions*:

**Statements -** “Do things.” A program is a sequence of statements. Here is an example of a statement, which declares (creates) a variable foo:

let statement = 'x';

**Expressions**

## Expressions are units of code that can be evaluated and resolve to a value. Expressions in JS can be divided in categories.

Arithmetic expressions

[String expressions](https://flaviocopes.com/javascript-expressions/#string-expressions)

[Primary expressions](https://flaviocopes.com/javascript-expressions/#primary-expressions)

[Array and object initializers expressions](https://flaviocopes.com/javascript-expressions/#array-and-object-initializers-expressions)

[Logical expressions](https://flaviocopes.com/javascript-expressions/#logical-expressions)

[Left-hand-side expressions](https://flaviocopes.com/javascript-expressions/#left-hand-side-expressions)

[Property access expressions](https://flaviocopes.com/javascript-expressions/#property-access-expressions)

[Object creation expressions](https://flaviocopes.com/javascript-expressions/#object-creation-expressions)

[Function definition expressions](https://flaviocopes.com/javascript-expressions/#function-definition-expressions)

[Invocation expressions](https://flaviocopes.com/javascript-expressions/#invocation-expressions)

// to be detailed //

**Scope**

Scope can be defined as the region of the execution, a region where the expressions and values can be referenced. // We can say the area where our code is written // Scope determines the accessibility (visibility) of variables.

JavaScript has 3 types of scope:

* **Block scope**
* **Function scope**
* **Global scope**

*Before ES6 (2015), JavaScript had only Global Scope and Function Scope.*

*ES6 introduced two important new JavaScript keywords: let and const. These two keywords provide Block Scope in JavaScript.*

Variables declared inside a { } block cannot be accessed from outside the block:

Global Scope: There's only one Global scope in the JS document. The area outside all the functions is considered the global scope and the variables defined inside the global scope can be accessed and altered in any other scopes.

// Global scope //

let globalScope = 'Outside area'

console.log(globalScope);

function getGlobalScope(){

console.log(globalScope); //globalScope can be accessed from here

}

getGlobalScope(); // 'Outside area'

**Local Scope**: The variable can be accessed **within** a function where it is declared. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local scope can be divided into **function scope** and **block scope**.

//global scope

//global scope

//global scope

function funcionEjemplo(){

//local scope 1

function funcionEjemplo2(){

//local scope 2

}

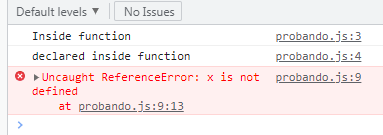
}

//global scope

//global scope

**Function scope:** It is the area of our function. A variable defined exclusively within the function cannot be accessed from outside the function or within another function

| function funcionEjemplo() {  var x = "declared inside function"; *// x can only be used in funcionEjemplo*  console.log("Inside function");  console.log(x); } *//global scope* funcionEjemplo(); *// calling the function* *//global scope* console.log(x); *// gives an error because 'x' is not a global variable* |
| --- |



**Block Scope**: Is the area within **‘if’**, **‘switch**’ conditions or **‘for’** and **‘while’** loops. // whenever we have {curly brackets}, it is a block. // In ES6, **const** and **let** keywords allow developers to declare variables in the block scope, which means **those variables exist only within the corresponding block.**

// global scope

function blockScope(){

// function scope

if(true){

// block scope

var fScope = 'Example One'; //exist in function scope

const bScope = 'Example Two'; //exist in block scope

let bScopeOne = 'Example Three'; //exist in block scope

// block scope

}

// function scope

console.log(fScope);

console.log(bScope);

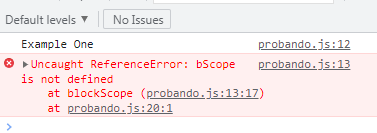
console.log(bScopeOne);

// function scope

}

// global scope

blockScope();



**Hoisting**

Hoisting is the default behavior in JavaScript, **which moves all declarations to the top of the scope before code execution (works with functions too).** It essentially gives us the advantage that no matter where functions and variables are declared, they will be relocated to the top of their scope, whether global or local.

**It allows us to call functions before even writing them in our code.**

*Note that only declarations, not initializations, are hoisted by JS*

*// Variable lifecycle*

*let a; // Declaration*

*a = 100; // Assignment*

*console.log(a); // Usage*

However, since **JS** allows us to *declare* and *initialize* our variables simultaneously, this is the most used pattern: .

*let a = 100;*

As said before **hoisting takes our** var **declaration to the top of its scope** and that’s why we are able to use it even though it gives us an undefined error. It is worth it to explain the difference between an **undefined error** and a **ReferenceError**.

The **undefined error** is when our variable is either not defined or defined as undefined (lol) whereas the **ReferenceError** is when we try to access an undeclared variable.

// hoisting

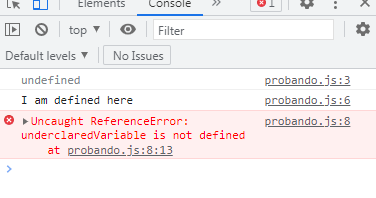
console.log(hoisting);

var hoisting;

hoisting = 'I am defined here';

console.log(hoisting);

console.log(underclaredVariable);



**Strict mode**

JS is referred to as a *forgiving* language because some of its syntax is optional and it can recover from some errors easily. This can result in an easier way to introduce bugs and make our code harder to read.

Here enters JS strict mode. This mode asks JS to not keep silent when it catches an error.

**Now how do we use it?**

We just simply write ‘use strict’ whenever we want to use this mode. At the top of our JS file, inside a function

Some benefits of using strict mode are:

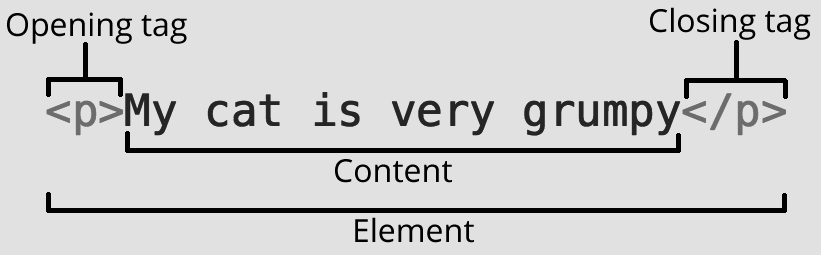
* Converts silent errors into ‘thrown’ errors which makes it easier to debug.
* It doesn’t allow duplicate properties or parameter values.
* When the variables are used without being declared or we call them “accidental globals”, strict mode throws an error.

***What is* HTML?**

**HTML** stands for***H****yper****T****ext* ***M****arkup* ***L****anguage*, is the code that is used to structure a web page and its content. We can make an analogy and say that HTML is like the skeleton of a body, it is what gives structure and support.

Let’s explore the anatomy of **HTML** - Consist primarily of *tags* that are enclosed inside angle brackets **< >** most *tags* are paired to indicate the start and the end of the element.

The *start* is marked by enclosing the tag inside angle brackets **<** tag **>** and to mark the end we need to add a forward slash at the beginning of our tag **</** tag **>**.



The types of tags used in the HTML document are responsible to tell a web browser to do something (follow the instruction) instead of just displaying text.

**Type of tags**

There are *three* types of tags:

Paired & Unpaired => As written before a paired tag is when the tag consists of an opening **<** tag **>** and a closing **< /** tag **>** . On the other hand, by deduction an unpaired tag does not require a closing tag. They are called Standalone or Singular tags as well. ~ *for obvious reasons, lol*. ~

<a></a>

<p></p>

<section></section>

<div></div>

<h1></h1>

<span></span>

Self-closing - These tags are just like the unpaired ones but the difference is that the main and important information is contained **within** the element as its *attribute*. A perfect example oh this is the img tag

<img src=”location of the img” alt=”brief description of the img”>

Utility-based - Depending on the purpose they server we can divide them into three sub-categories:

Formatting tags - These tags help us to format text characteristics such as the size, weight (bold), italic, and so on.

<b></b>

<u></u>

<strong></strong>

<i></i>

Tables, divisions, and span tags are also formatting tags but in this case they format a web page or document and set the layout of it.

| <table></table> <span></span> <section></section> |
| --- |

Structure tags - These are the tags that helps in structuring the HTML file such as html, head, body, etc. A basic HTML file should contain:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

</body>

</html>

Control tags - These are the script tags, radio buttons or checkboxes, form tags, etc. These are tags used to interact with the user.

<script></script>

<form action=""></form>

<input type="checkbox" name="" id="">

<input type="radio" name="" id="">

**Input**

As stated before the <input> tags are used to create interactive elements for web-based forms in order to accept data from the user.

This tag is one of the most powerful and complex in all of **HTML** due to the many number of combinations of input types and attributes.

<input type=” ”>

Inputs work differently depending on the value of its type attribute. Some types are:

<input type="button" value=""> <!-- This is a basic push button with no default behavior-->

<input type="checkbox" name="" id=""> <!-- A check-box allowing single values to be selected/deselected.-->

<input type="color" name="" id=""> <!-- A control for specifying a color; opening a color picker-->

<input type="date" name="" id=""> <!-- A control for entering a date (year, month, and day, with no time). Opens a date picker -->

<input type="datetime-local" name="" id=""> <!--A control for entering a date and time, with no time zone. Opens a date picker or numeric wheels for date- and time-components-->

<input type="email" name="" id=""> <!--A field for editing an email address. Looks like a text input, but has validation parameters-->

<input type="file" name="" id="" accept=""> <!--A control that lets the user select a file. Use the accept attribute to define the types of files that the control can select.-->

<input type="number" name="" id=""> <!--A control for entering a number. Displays a spinner and adds default validation when supported-->

<input type="password" name="" id=""> <!--A single-line text field whose value is obscured. Will alert user if site is not secure-->

<input type="submit" value=""> <!--A button that submits the form-->

The type attribute is the most important one but there are many more, these are some of them.

| Attribute | Type or types | description |
| --- | --- | --- |
| autocomplete | all | Autofill feature |
| disabled | all | Disabled, lol |
| form | all | Associates the control with a form element |
| name | all | Name of the form control. Submitted with the form as part of a name/value pair |
| required | all (almost) | A value is required or must be check for the form to be submitted |
| type | all | Type of form control |
| value | all | The initial value of the control |
| multiple | Email, file | Boolean, whether to allow multiple values |
| size | Email, password, tel, text, url | Size of the control |
| capture | file | Media capture input method in file upload controls |
| alt | image | Required for accessibility |
| height | image | Same as the height attribute for <img>. A vertical dimension |
| width | image | Same as the width attribute for <img> |
| pattern | Password,tel, text | The value must match to be valid |

**XHTML**

**XHTML** stands for **E**xtensible **H**yper**T**ext **M**arkup **L**anguage. Is a combination of **XML** and **HTML**. **HTML** is used for the *presentation of the data*, while **XML** is used for *carrying the data*.

**XML** (**E**xtensible **M**arkup **L**anguage) is similar to **HTML** but without predefined tags. ***You define your own tags according to your needs.***

Normal **HTML** works in most browsers, even if it is semantically poorly written, nowadays there is a huge number of browsers and includes smaller devices such as tablets, smartphones, etc. And their capacity to interpret a bad quality code is not the best.

**XML** is stricter than **HTML** in this aspect. A forgotten tag or an attribute without quotes makes the **file unusable** whereas in HTML such practice is often allowed or tolerated.

Both make use of *tags* (words bracketed by '<' and '>') and *attributes* (of the form name="value"), but while **HTML** specifies what each *tag* & *attribute* means (and often how the text between them will look in a browser), **XML** uses the tags only to delimit pieces of data, and leaves the interpretation of the data completely to the application that reads it.

**XHTML** is born from the combination of the *strength* of both, **XML** and **HTML** allowing the browser to interpret code more accurately and enhances the compatibility with other data formats.

**XHTML** makes viewing *mobile websites easier*, so it is mostly used for *mobile development*. It is super easy to convert an existing **HTML** file to a **XHTML** file with just a few changes.

We can take into consideration some advantages and disadvantages when deciding if to use XHTML.

Advantages

* **Extensibility**, as we define our own tags, we can implement new ideas as web communication and presentation logic emerges.

// In software development, presentation logic is concerned with how business objects are displayed to users of the software, e.g. the choice between a [pop-up](https://en.wikipedia.org/wiki/Context_menu) screen and a [drop-down menu](https://en.wikipedia.org/wiki/Drop-down_menu). //

* **Portability,** processing becomes simpler and painless for **XML parsers** since it adheres to **XML** standards. It allows web pages to be made simpler so that they can be handled by small devices. This is particularly relevant for mobile devices and compact devices with low-power processors. The advantage of portability means that we can create a document to meet a specific demand whenever it arises.

// To know the very basics, XML parsers are also known as XML Processor and it aims to check the syntax for a well- defined file. The main job of an XML parser is to access or modify the data in the document. The parser contains installed software packages for the client applications to interface with and also does the validation process of the XML Documents.To learn more follow this[**link**](https://www.educba.com/xml-parsers/) //

* **Easy to maintain**, as the rules are clear there is less margin of error. The structure is more apparent and syntax problems are easier to spot.
* **Ready for future release versions**, the documents will easily upgrade to the newest version.

Related to **disadvantages** we cannot really say there aren’t but…

* It does not solve all browser compatibility
* It is difficult to assimilate at the start because of its strictness and sometimes it is difficult to come up with a new name for a new element.

And some differences between both -

To say some, HTML is not case sensitive, it can make use of open tags such as <br>, all of the content can be included in the body element. On the other hand XHTML is case sensitive, all unclosed tags must be closed and the content must be put in blocks.

**DATA-\* ATTRIBUTES**

[data-\* attributes](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/data-*) allow us to store extra information on standard, semantic HTML elementsThe syntax is simple. Any attribute on any element whose attribute name starts with data- is a data attribute.

<h1>DATA- ATTRIBUTE LIST</h1>

<ul id="list">

<li data-name-any="4">ITEM 1</li>

<li data-name-name="35">ITEM 2</li>

<li data-name-something="cars">ITEM 3</li>

</ul>

Following the ‘data-’ comes the name with lowercase letters and only letters. Then the data value.

JAVASCRIPT ACCESS

Reading the values of these attributes out in [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript) is also very simple. You could use [getAttribute()](https://developer.mozilla.org/en-US/docs/Web/API/Element/getAttribute) with their full HTML name to read them, but the standard defines a simpler way: a [DOMStringMap](https://developer.mozilla.org/en-US/docs/Web/API/DOMStringMap) you can read out via a [dataset](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset) property.

To get a data attribute through the dataset object, get the property by the part of the attribute name after data- (**note that dashes are converted to camelCase**).

const dataAttribute = document.querySelector('#list');

dataAttribute.dataset.nameAny // "4"

dataAttribute.dataset.nameName // "35"

dataAttribute.dataset.nameSomething // "cars"

Each property is a string and can be read and written. In the above case setting *dataAttribute.dataset.nameAny = 6* would change that attribute to "6".

**Accessibility**

Accessibility is the concept of whether a product or service can be used by everyone. Translated to web development is the practice of making your websites usable by as many people as possible.Not just people with disabilities but those who use mobile devices or those who have a slow connection.

Accessibility is the right thing to do. To provide accessible sites is part of the law in some countries which can open up some significant markets that otherwise would not be able to use your services or buy your products.

We should consider the number and types of potential accessibility issues users will have such as ***visual*** (e.g., color blindness), ***motor/mobility*** (e.g.wheelchair-user concerns), ***auditory*** (hearing difficulties), **seizures** (especially photosensitive epilepsy), **learning/cognitive** (e.g., dyslexia), **incidental** (e.g., sleep-deprivation), **environmental** (e.g., using a mobile device underground) to name some.

According to [**IAPB**](https://www.iapb.org/learn/vision-atlas/) there are 43 million people living with blindness and 295 million people living with moderate-to-severe visual impairment worldwide.

People with visual impairments (blindness, low-level vision, and color blindness) use software zoom or screen readers (a software that reads digital text aloud).

These are some recommendations:

* Use alt text on content-enhancing images. A description of the image would be the aim as a screen reader will read the alt text aloud in place of the image.
* Have a link strategy (describe the link before inserting it “Read more about something, at [their website](http://interaction-design.org/).”)
* Offer visual cues (PDF icons), underline links and highlight menu links on mouseover.
* Improve visibility with careful color selection and high contrast.
* Reference shapes to help guide users (e.g., “Click the square button”).
* Offer transcriptions for audio resources, captions/subtitles for video.
* Make content easily understandable – simpler language reaches more users, as do effective information hierarchy, progressive disclosure and prompting.
* Try using your design without a mouse. It can be hard to scroll.
* Use tools such as WAVE and Color Oracle to test your design’s accessibility.

According to [**WWH**](https://www.bing.com/search?q=hearing+impairments+worlwide&cvid=a20310da00d44cfaa8f374aa2825b2e6&aqs=edge..69i57j0l8.8118j0j1&pglt=41&FORM=ANNTA1&PC=U531) over 400 million people worldwide have disabling hearing loss and hearing loss is rising fast globally. A text alternative must be provided to enable access. Videos should be manually subtitled and transcripts should be provided for audio content. Furthermore, due to the higher degree of language deficits in the DHH (deaf or hard of hearing) population, text simplification should be considered.

People with reduced mobility, which may involve purely physical problems (such as loss of limbs or paralysis), or neurological/genetic disorders that may cause weakness or loss of control in the limbs or even age rather than any specific trauma or condition also may result in hardware limitations. This often affects web development work as it requires controls to be keyboard accessible, e.g. using the Tab key to move between different controls on a web form.

Cognitive impairment refers to a broad range of disabilities, from people with intellectual disabilities who have the most-limited capabilities, to all of us as we age and have difficulty thinking and remembering. It also includes people with learning disabilities, such as dyslexia and attention deficit hyperactivity disorder. People with them experience a common set of functional problems. These include difficulty with understanding content, remembering how to complete tasks, and confusion caused by inconsistent webpage layouts.

A good foundation of accessibility for people with cognitive impairments includes:

* Delivering content in more than one way, such as by text-to-speech or by video.
* Easily understood content, such as text written using plain-language standards.
* Focusing attention on important content.
* Minimizing distractions, such as unnecessary content or advertisements.
* Consistent web page layout and navigation.
* Familiar elements, such as underlined links blue when not visited and purple when visited.
* Dividing processes into logical, essential steps with progress indicators.
* Website authentication as easy as possible without compromising security.
* Making forms easy to complete, such as with clear error messages and simple error recovery.

*Side note - Designing with* [*cognitive accessibility*](https://developer.mozilla.org/en-US/docs/Web/Accessibility/Cognitive_accessibility) *will lead to good design practices. They will benefit everyone.*

The [Web Content Accessibility Guidelines (WCAG)](https://www.w3.org/TR/WCAG21/?msclkid=e2bcda9ec81411ec9ed3e7c340bae41e) are technical standards that define 12 to 13 guidelines between them.These guidelines and criteria have been organized under four design principles: Perceivable, Operable, Understandable and Robust known as the **POUR** principles of accessibility.

**P**erceivable, are our five senses - *sight, smell, sound, taste*, and *touch* - that help us perceive what's going on around us. Our brain need information to be received by our senses in order to interpret, understand, or perceive it

We employ a combination of *sight*, *sound*, and *touch* to access electronic content. Users must have the option of using any combination of these three senses to access content - *visual*, *auditory*, or *touch* (via braille devices).

In this context, **it is the semantic meaning and HTML structure that is important.** Styling might enhance the visual presentation of the content but screen readers won't be able to see this.

**O**perable, people might use assistive technologies to access content, and it won't always be a keyboard and mouse. Users must be able to ***find***,***navigate***, and ***interact*** with content using any method available. To name a few, this could include a keyboard, a braille keyboard, sip and puff (**SNP** is an assistive technology used to send signals to a device using air pressure by "sipping" (inhaling) or "puffing" (exhaling) on a straw, tube or "wand." It is primarily used by people who do not have the use of their hands.), screen readers, and voice commands.

**U**nderstandable, by providing **plain** and **simple language**, we can facilitate users in comprehending the information offered. Users must also be able to comprehend the **functionality** and how to **execute a task properly**. This demands a **uniform presentation of information**. If there are any errors, users should be given **clear instructions** on how to fix them and continue on. *People want to be challenged by the content, not by the user interface, therefore keep that in mind.*

**R**obust, people use a variety of browsers, media players, assistive technologies, and gadgets to access web information. Some users may make use of cutting-edge technology. To put it another way, this principle includes verifying that the code underlying our website is well-formed (semantically valid) in order for it to work with the widest range of current and future technologies.

WCAG guidelines are put into three levels of conformance in order to meet the needs of different users. A being the lowest and AAA the highest. Level A sets a minimum level of accessibility and does not achieve broad accessibility for many situations. For this reason, UC recommends AA conformance for all Web-based information and it is not possible to satisfy all Level AAA success criteria for some content. For that reason the passing grade and the level expected is AA.

Here’s the [**link to meet the conformance requirements**](https://www.w3.org/TR/WCAG20/#conformance-reqs) in detail.

Semantic HTML

Semantic HTML was mentioned in the accessibility section of this handbook. But what does it really mean?

Semantic HTML elements are those that clearly describe their meaning such as <header>, <footer>, <section> they accurately describe their purpose and the type of content we can expect inside of them.

The internet was originally created to share scientific documents but people started to share other things and the web started to get complex and look nicer.

As there weren’t many tags to give a complex design to a web page, developers developed (lol) some hacks to make this happen using tags that had different purposes. For example, to use the **<table></table>** to position other elements rather than to describe information.

The use of generic tags as **<div>** by giving them a *class* or an *id* and using them as *headers*, *footers*, *navbars*, etc.

Elements such as **<header>**, **<nav>**, **<section>**, **<article>**, **<aside>**, and **<footer>** act more or less like a **<div>** element - *they group other elements together into page sections -* however where a **<div>** tag could contain any type of information, it is easy to identify what sort of information would go in a semantic **<header>** region.

The benefits of writing semantic code is that it is much easier to read and since developers spent most of the time reading rather than writing results in a much easier work, it helps tons with accessibility as developed before.

The new semantic elements of HTML5 are:

<article></article>

<section></section>

<aside></aside>

<details></details>

<figcaption></figcaption>

<figure></figure>

<footer></footer>

<header></header>

<main></main>

<mark></mark>

<nav></nav>

<summary></summary>

<time></time>

The **<section>** and **<article>** tags are similar and interchangeable. To decide which of these should we use, we have to keep in mind the following:

1. An article is intended to be independently distributable or reusable.
2. A section is a thematic grouping of content.

Using the newspaper analogy, the sports **section** will contain **articles** that are about sports, with each individual piece having its own *heading* and story and being entirely self-contained.

The **<aside>** element is intended for content that is not part of the flow of the text in which it appears, however still related in some way. Contains only the content related to the main web page.

Before HTML5, web pages’s menus were created using *<ul>’s* and *<li>’s* tags. Now, together with these, we can separate our menu items with a **<nav>** tag, for navigation between our pages. We can have any number of **<nav>** elements on a page, for example, it's common to have **global navigation** across the top (in the **<header>**) and l**ocal navigation** in a sidebar (in an **<aside>** element).

The **<footer>** tag defines the footer (duh) of a web page or a section. It contains copyright, authorship and contact information, back to top links, sitemap, related documents, social media links, etc.

We can have several **<footer>** tags on a web page. For example, we can place a footer inside the [**<article>**](https://www.w3docs.com/learn-html/html-article-tag.html) tag to store information related to the article (links, footnotes, etc.).

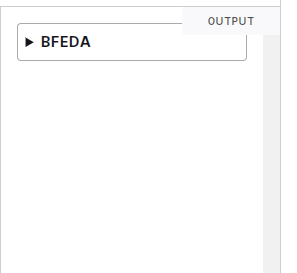
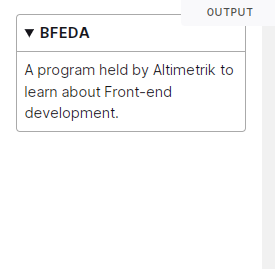
The **<details>** tag creates a disclosure widget in which information is visible only when the widget is toggled into an *"open"* state. A summary or ‘title’ must be provided using the [**<summary>**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/summary) element.

<details>

<summary>BFEDA</summary>

A program held by Altimetrik to learn about Front-end development.

</details>



This is how it looks once the browser renderized it. When it is ‘closed’ it shows the summary tag content and when it is open, shows the rest.

The **<figcaption>** tag represents a caption or legend describing the rest of the contents of its parent [**<figure>**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/figure) element.

The **<figure>** tag represents a unit of content, optionally with a caption, that is self-contained and typically referenced as a single unit. *Usually this is an image, an illustration, a diagram, or a code snippet that is referenced in the main text.*

<figure>

<img src="../../../BFEDA/ALTIMETRIK\_LOGO.jpg"

alt="Altimetrik's logo">

<figcaption>This is Altimetrik's logo</figcaption>

</figure>



The **<main>** tag represents the dominant content of the [**<body>**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) of a document. A document mustn't have more than one **<main>** element.

The **<mark>** tag It marks (duh) a part of the text which has relevance. It can be used to **highlight** text for **showing emphasis**, **highlight search terms** in search results to provide context; or d**istinguish new content** added.

The **<header>** tag defines the header (duh) of a page or a section. *It usually contains a logo, search, navigational links, etc.*

It doesn’t present a new section in the outline.It commonly contains the heading (an [**<h1>**–**<h6>**](https://www.w3docs.com/learn-html/html-h1-h6-tags.html) element) of a surrounding section. However, this is not required.

There are from **<h1>** to **<h6>** *heading tags* - The **<h1>** tag is the most important element of a webpage, and search engines use this element to understand the topic of the page.

**<h1>** was always crucial for **SEO** and still continues to be a major ranking factor. Using the primary keyword in the **H1** tag helps the page to rank highly for that keyword. *Headings are crucial for text and content organization, and we should take it seriously. Utilizing the heading tags certainly adds up to the architecture of the content.*

For **search engines**, *it’s easier to read and understand the well-organized content than to crawl through structural issues.*

For **users**, *headings are like anchors in a wall of text, navigating them through the page and making it easier to digest.*

**<h1>** tag defines the most important heading on the page, while **<h6>** defines the least important heading.

*We can compare* ***<h1>*** *tags with newspaper headlines. If they attract you, then you read the entire article. If they don’t, you ignore them.*

<h1> tags best practices:

* Should include one main keyword (do not over stuff our heading with tons of keywords)
* It is recommended to use just one <h1> tag per page and then the others according to their relevant order.
* An exact match of the keyword and the <h1> tag drastically improves the ON-Page SEO of your page.
* <h1> tags often get displayed as a snippet in the search engine result pages. If your <h1> tag is click-worthy and engaging, more people click on it.

**META TAGS & SEO**

Meta tags are elements within our **HTML** code. It is not visible to the user but plays a crucial role in our **SEO** performance. By using these tags we make sure that browsers and search engines behave the way we want them to.

All meta tags are defined using the **<meta>** tag (<title> tag is the exception) and they are placed in the **<head>** of our **HTML** file.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name=””

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Michael Armesto - BFEDA 2022</title>

<meta name="description" content="I am Michael Armesto a trainee front-end developer and this is my very first project. Following Altimetrik’s program Become a Front-End Developer With Altimetrik">

<meta name="keywords" content="html, css, javascript, js, developer, frontend, web developer, michael, armesto">

</head>

<body>

</body>

</html>

**<title>** tag is the first element that specifies our **HTML** file for search engines and users. It appears as the clickable headline for the search result.

**<meta name=”description” content=””>** It is a description that summarizes our web page. This is shown (normally) under our title in **s**earch **e**ngines **r**esults **p**age (**SERP**)

(under our clickable title when we search for something - about 120 - 150 characters)

**<meta name=”keywords” content=””>** It is a concept that has become *obsolete* but that was previously vital for **SEO** positioning. Through **keywords** we could tell the search engine what was the *content* or *theme* of our website but currently it is the content of the website itself and how these keywords are distributed, the meta tags title and description on which the weight of SEO positioning falls.

**<meta charset="utf-8" />** This defines the character encoding for your website. It tells browsers and search engines how to interpret the code/content of your website at the base level.***UTF-8****, also known as Unicode, is the character encoding that covers almost all of the characters and symbols in the world.*

***<meta name="viewport" content="width=device-width, initial-scale=1.0">*** it sets the **pixel-width of the page** to be the same as the **pixel-width of the device** it is viewed on, as well as setting the **starting zoom-level to default**. This is important because in *responsive web-design* you often want to *display content based on the device width.*

**What is SEO?**

Stands for **S**earch **E**ngine **O**ptimization, which is a set of good practices designed to improve our positioning in the **SERP.**

Let’s take a look at some concepts:

**Organic search results**: It refers to the non-paid search results from a search engine. These results can’t be bought or influenced by advertisers; they’re the ones the search engine deems most relevant to the user’s search query.

**Organic traffic**: It refers to the number of users who reach your site via organic search results. Users are [far more likely](https://moz.com/blog/new-google-survey-results#:~:text=interactions%20with%20organic%20results) to click on search results that appear near the top of the **SERP**, which is why it’s important to use your **SEO** strategy to rank relevant pages as highly as you can. When users type a query in a search engine, they have a very specific purpose. If our website can provide a user with a solution, we will mostly gain a new customer. If not a conversion, then maybe a new follower or an email subscriber. Moreover, **organic traffic increases website trust.** Websites that organically rank in the top positions of search engines are trusted by users.

**We can follow some tips for a good SEO performance:**

* First, a correct use of our meta tags is a crucial component. Using the right meta tags, the right way. As said before our <title> element is our main and most important. The page’s title is the first thing for a searcher to see in SERPs and decide if the page is likely to answer the search intent. A well-written one may increase the number of clicks and traffic, which have at least some impact on rankings.
* We should give each page a unique title that describes the page’s content accurately.
* Keep title around 50-60 characters long
* Make use of our brand name in the title (even if it’s not shown on the SERP) still makes a difference
* **Meta description tags**, it occupies the largest part of a **SERP** snippet and invites searchers to click on your site by promising a clear and comprehensive solution to their query. *That’s why the description must be as realistic as it is inviting and distinctly reflect the content.*
* IMG alt attribute, this is good to implement for two reasons:
* Alt text is displayed to visitors if any particular image cannot be loaded (or if the images are disabled). And to gain accessibility to your page if a user needs to use a screen reader.
* Alt attributes provide context because search engines can’t “see” images. Helping search engines understand what the images are about and how they go with the rest of the content may help them serve a page for suitable search queries.

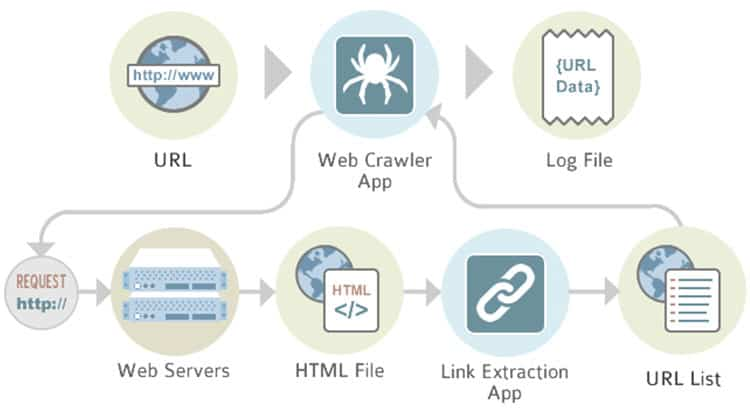
This[**page**](https://www.searchenginejournal.com/important-tags-seo/156440/#close) covers in depth a wider strategy to perform even better at SEO

**Web Crawler**

Web crawlers are also called bots, robots or spiders. Search engines do not automatically know which websites are out on the Internet. They need to have programs to crawl and index sites before delivering the correct pages for phrases and keywords or the words that people use to find a helpful page.

Without crawlers, the internet would be a jumbled, scattered mess of information.

Search engine crawlers need to have a starting place – a link – before seeing the following link and the subsequent pages.



The search engines visit or crawl websites by passing between the various links associated with different websites. However, if your website is brand new without links connecting your website’s pages with others, you can request that the search engines crawl your site.

Google, for instance, allows you to submit your website to their crawlers via a tool called [Google Search Console](https://search.google.com/search-console/about).

While web crawlers are on a page, they collect information about it, such as the meta tags and the copy. The crawlers then store these pages in their index so the search engine algorithms can sort them for their contained words to retrieve and rank for its users later on.

The index is a database of web pages that the crawler has discovered. This database is what a search engine like Google pulls its search results from.

Although it’s important to give the crawler proper access to your site so you can appear in search results, it’s not always necessary to have the crawler access every page of your site and by blocking the crawler’s access to pages, you save the [crawl budget](https://www.seoclarity.net/pillar/crawl-budget).

**What exactly is a crawl budget?** *A search engine bot's time and resources are finite, so the crawl budget defines how many pages a bot will crawl within a specific amount of time.*

By limiting the access to non-important pages, you extend your crawl budget to further guarantee that your important pages will make it into the index.

*There are a few ways to optimize the crawl budget:*

* **“Noindex” Tag**: A “noindex” tag informs search engine bots of which pages to not include in their index. The implementation of this tag will remove pages from the index. This is often used for pages such as thank you pages, admin and login pages, internal search results, etc.

<meta name="robots" content="noindex">

* **Canonical Tag**: These tags will inform Google that a grouping of similar pages has a preferred version that you wish to be shown in the SERPs. By adding it we define which URL is the original URL of a page. If we have a page with both a mobile and a desktop version, Google sees these as duplicate versions of the same page.

<link rel="canonical" href="URL goes here">

* **Robots.txt**: [Robots.txt](https://www.seoclarity.net/blog/understanding-robots-txt) is a file that a search engine bot will read before it crawls your site. This file sets parameters on which pages are and are not to be crawled.

All major search engines have web crawlers, with the big ones having multiple crawlers with specific focuses to be specific Googlebot is Google’s main crawler, which does both desktop and mobile crawling but there are also several other bots, including AdsBot, Googlebot News, Googlebot Videos, and Googlebot Images.

**Geotagging**

Geotagging is particularly useful for local SEO goals.

A single point or location is referred to as a **geo**. There is a distinct **geo-location** coordinate for wherever you are on the earth, and if you travel five feet away from that spot, there is a different coordinate for that location therefore **Geotagging pinpoints the exact location of your company.**

**Geotagging** adds global location data to a website or image discovered online, such as specific latitude and longitude and search engines scan, catalog, and analyze this data when they crawl the web looking for data, but our website users don't see it. *Search engines are integrating the person's present location into the search results they offer as more potential customers search for information from their mobile devices.*

**Geotagging** is indeed vital for local search engine results ranking. If our cleaning services company is based in Montevideo, Uruguay, locating customers in Paris, Virginia, or France is unimportant. We need local results from web searches to grow our local business. We must locate clients in our close surroundings and provide them with information when they require it.

*How do we add this information?*

Making sure all of our photographs were taken using a current *GPS-enabled* smartphone, which is the first and easiest approach to ensure they all have **geotagging** information. This can be done using smartphones or digital cameras. Simply upload them to your website and include the additional information for the picture filename and alt-text as described above, and we'll be on our way to bettering our **SEO**.

*Manually editing the photo's geotag metadata is possible. This information, along with other technical data, is kept in the Exchangeable Image File (EXIF) data of any JPEG shot. We can open the images in a variety of programs and add or edit the geotag information.*

**DOM**

The **D**ocument **O**bject **M**odel (**DOM**) is the data representation of the objects that make up a web document's structure and content.

And **HTML** is our markup language that allows us to represent a certain type of **DOM**. // With our HTML and its tags we can make differents DOMs with different structures such as index.html or profile.html //

It describes a page so that programs can alter the structure, style, and content of the document. The Document Object Model (DOM) represents the document as nodes and objects, allowing programming languages to interact with them.

The **DOM** is a logical **tree** that describes a document. **Each branch of the tree ends in a node, which stores objects.**

**Tags are the backbone of an HTML document** and according to the **DOM**, every **HTML** tag is an object. Nested tags are “children” of the enclosing one. The text inside a tag is an object as well.

All these objects are accessible using **JS**, and we can use them to modify the page.

<!DOCTYPE HTML>

<html>

<head>

<title>A basic example of DOM</title>

</head>

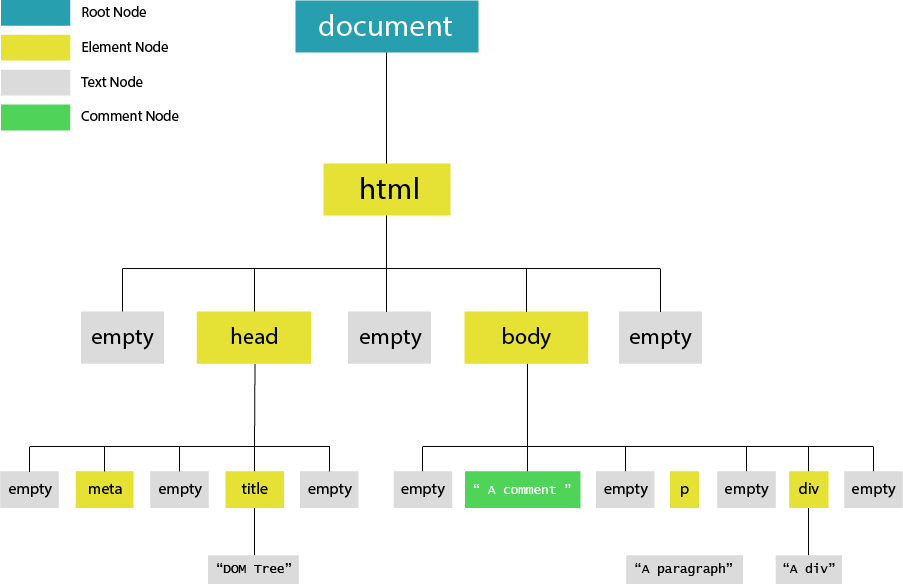
<body>

<h1> BFEDA 2022 </h1>

</body>

</html>

The DOM represents HTML as a tree structure of tags. Here’s how it looks:



All items in the **DOM** are defined as **nodes**. There are many types of nodes, but there are *three main ones that we work with most often*:

* **Element nodes**
* **Text nodes**
* **Comment nodes**

When an **HTML** element is an item in the **DOM**, it is referred to as an **element node**.

Any line text outside of an element is a **text node**, and an HTML comment is a **comment node.** In addition to these three node types, the document itself is a **document node**, which is the root of all other nodes.

*It is like an ancestral family tree, which consists of parents, children, and siblings. The nodes in the DOM are also referred to as parents, children, and siblings, depending on their relation to other nodes.*

The process of modifying the content and structure of the DOM is known as DOM manipulation (document object model). By adding, removing, or changing items and their characteristics, the DOM can be changed.

The following is a list of actions that we can achieve with DOM manipulation methods:

1. **Creating new element**
2. **Appending new element**
3. **Removing element**
4. **Replacing element**
5. **Changing style**
6. **Adding attribute**
7. **Removing attribute**
8. **Adding event listener**

**How to manipulate the DOM with JS**

For **creating a new element** we need to use the **document.createElement()** method and pass the name of the element as an argument.

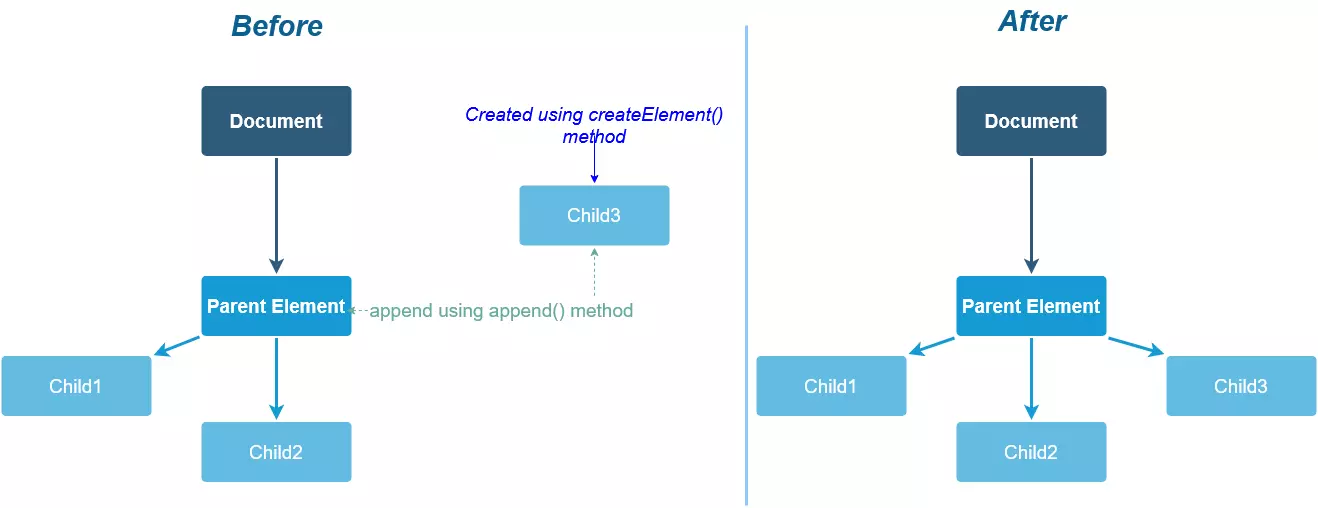
let createFirstElement = document.createElement('p');

let createSecondElement = document.createElement('img');

let createThirdElement = document.createElement('li');

We created the element and saved its reference in a variable, however the formed element is floating aimlessly because the DOM was unaware of it till now. We have to attach the element to the document using:

* **append()** - It appends the node objects or DOMString (text) object to the parent element.
* **appendChild()** - it can append only node objects to the parent element.



.

<div id="parent">

<p>This is the first Bendición.</p>

</div>

<button onclick="addBendicion()">Agregar Bendición</button>

<script>

function addBendicion() {

const parent = document.getElementById("parent"); // selecting parent

const bendi = document.createElement("p"); // creating a child 'p' element

bendi.innerHTML = "This is the second Bendición"; // adding some content to the 'p' element we created.

// appending child to parent

parent.append(bendi);

// we can also use appendChild()

}

</script>

We not only want to add or create elements, we also will need to remove them.

The **removeChild()** allows us to do it.

// selecting parent and child

const parent = document.getElementById("parent");

const Bendi = document.getElementById("laBendicion");

// removing child from parent

parent.removeChild(Bendi);

*It is needed to call this method from the parent otherwise it will throw an error.*

Following the same logic we can manipulate the **DOM** by replacing an element with another element: **replaceChild()**

parentElement.replaceChild(newChild, oldChild);

* **parentElement** - Parent element whose child is to be replaced.
* **newChild** - Element which will replace another one.
* **oldChild** - Element which will be replaced.

We can even change the **style** of an element controlling any CSS property of the element.

element.style.property = value;

* **element** - The element whose style is to be changed.
* **property** - The property of the element whose value is to be changed.
* **value** - The value of the property.

To add HTML attributes to an element is also possible using **setAttribute()**

element.setAttribute(attribute, value);

* **element** - The element whose attribute is to be added.
* **attribute** - The attribute of the element.
* **value** - The value of the attribute.

The same way to remove attributes we use **removeAttribute()**. We need to call this method on the element and pass the attribute as the argument.

element.removeAttribute("title");

Adding event listeners to elements is also possible **addEventListener()** (I will go in depth into this topic in the following week). An event listener is a function that is called when an event occurs on an element.

element.addEventListener(event, function(){

});

* **element** - The element on which the event is to be added.
* **event** - The event which is to be added.
* **function** - The function which is to be called when the event occurs.

An **event** is a *‘thing’* that happen to an **HTML** element and **JS** can *‘react’* to it some common events are:

* onchange - An HTML element has been changed
* onclick - the user clicks an HTML element
* onmouseover - Similar to CSS pseudo-class **hover.** When the user moves the mouse over an HTML element
* onmouseout - the inverse of onmouseover
* onkeydown - the user pushes a keyboard key
* onload - the browser finished loading the page

**CSS**

CSS stands for **C**ascading **S**tyle **S**heets. Previously we made an analogy that HTML was like the skeleton or structure of the body keeping the same analogy we can now think of CSS like the skin and flesh of that body. So with a little makeup and plastic surgeries we can make it look what society defines as ‘pretty’. (lol)

CSS brings style to our page by interacting with HTML elements. To do so we need to call the element with a selector.

Selectors define the elements which are going to be applied CSS rules.

*The fact that it is called ‘Cascading’ means that it reads the rules from the top to the bottom. So if I select an element and apply certain rules to it and later on I call it again and apply the same set of rules but with different values, the last one is the one that is going to end up being applied to the element.*

We have an **universal selector** ‘ \* ’ an asterisk that will call all of the elements of our HTML and apply the rules given.

\* {

margin: 0;

padding: 0;

box-sizing: border-box;

outline: none;

border: none;

text-decoration: none;

font-family: 'Source Code Pro', monospace; }

These rules are applied to every element inside the HTML file that the CSS file is linked to.

Then we have a **type selector** which is going to select all tags that match its name. For example, if I type the letter p in CSS it is going to select all of my <p> tags in my HTML document and apply the rules I gave to it.

a {

text-decoration: none; }

All of my <a> tags (and only <a> tags) will have a decoration of none.

We also have a **class selector**. The syntax of this selector is easy. To select a class just need to put a dot before the class name given to the element. For example if i gave to an element the class=”text” I will select that class on CSS like this .text

HTML element:

<p class="text">Hey there Ariadna!</p>

CSS:

.text {

color: #f8f7ffd3;

line-height: 1.5em;

font-weight: lighter;

margin-bottom: 0;

text-align: center; }

**Id selector** selects the element based on their id value. **There must be only one unique element per id.**

HTML:

<p class="text" id="example">Hey there Ariadna!</p>

CSS:

.text {

color: #f8f7ffd3;

line-height: 1.5em;

font-weight: lighter;

margin-bottom: 0;

text-align: center; }

#example{

color: blue;

}

While **.text** rules are applies to all of my elements with that class **#example** is only applied to that particular element.

In the last example I am selecting the same element with two different selectors and a different set of rules. Which one will it be applied to? Here enters the **specificity.**

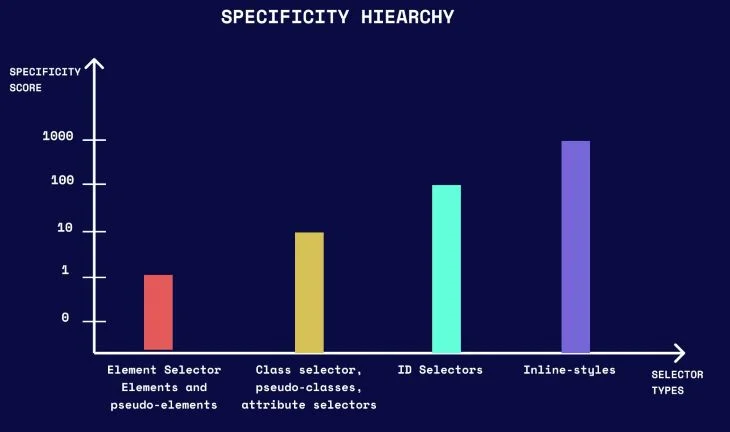
**Specificity**

**Specificity** implies assigning a score to selectors in order to rank or compare them. The style rules of the selector with the highest score will be applied to the element.

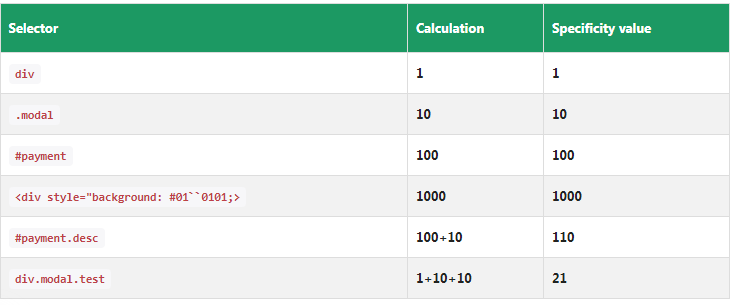
A four-part specificity hierarchy determines a selector's specificity score:

* **Inline** style selectors are **style rules defined in an HTML document using the style attribute**. These selectors are given a specificity **score of 1000** and, as such, take precedence over other selector types
* **ID selector** like #text, have a specificity **score of 100**
* **Class, pseudo-class, and attribute selectors** have a specificity **score of 10.** .text, :active, and [attr] are examples of class and pseudo-class, and attribute selectors, respectively
* **Element and pseudo-element selectors** have a specificity **score of one**. div and :after are examples of element and pseudo-element selectors, respectively

The **universal selector** has a specificity of **zero**; therefore, *its style rules will be overridden whenever there are conflicting selectors.*



We can even add up specificity like in the following example:



**Linking a .css file**

There are **three primary** **methods to include CSS code in our HTML** file. Might be **external**, **internal**, or **inline**. **External** can be used to control the design of an entire website using just one file *(rather than adding individual instances of CSS code to every HTML element you want to adjust).* To use an external style sheet, we must add a header section in our.html files that connects to the **external** style sheet and looks like this:

<head>

*<meta charset="UTF-8">*

*<meta http-equiv="X-UA-Compatible" content="IE=edge">*

*<meta name="viewport" content="width=device-width, initial-scale=1.0">*

*<title>Michael Armesto</title>*

*<meta name="description" content="I am Michael Armesto a trainee front-end developer and this is my very first project">*

*<meta name="keywords" content="html, css, javascript, js, developer, frontend, web developer, michael, armesto">*

<link rel="stylesheet" href="styles/css/style.css">

</head>

**The internal** CSS code is the one that appears inside the HTML document's head section. When you want to give a specific page its own custom look, this is very useful. There is no difference between CSS code that is placed inside an external file and CSS code that is placed inside an internal file. The only thing to keep in mind is that it must be enclosed within the <style> tag.

**Inline** styles are snippets of CSS written directly into the HTML element. For example:

<h1 style=”font-size:40px;color:violet;”>This is for BFEDA 2022</h1>

**Critical CSS**

When working on big projects our css file could end up rather long and heavy causing our web to be slow when loading. This is when the critical CSS strategy comes in handy. In this strategy we're looking for the bare minimum CSS, or crucial CSS, to make the website appear to the user.

To start working with the critical CSS we need to change our approach to the way we handle the CSS – *this means splitting it into two files. For the first file, we extract only the minimum set of CSS required to render the first section/thing the user is going to see, and then we write it as an internal CSS between the <style> tags.* And for the second file we are using the **loadCSS();** function to asynchronously load the remaining, non-critical CSS. This is important because we are essentially off-loading the bulkier *(non-critical)* CSS and injecting it into the web page in the background.

<!doctype html>

<head>

<style> /\* inlined critical CSS \*/ </style>

<script> loadCSS('non-critical.css'); </script>

</head>

<body>

...body goes here

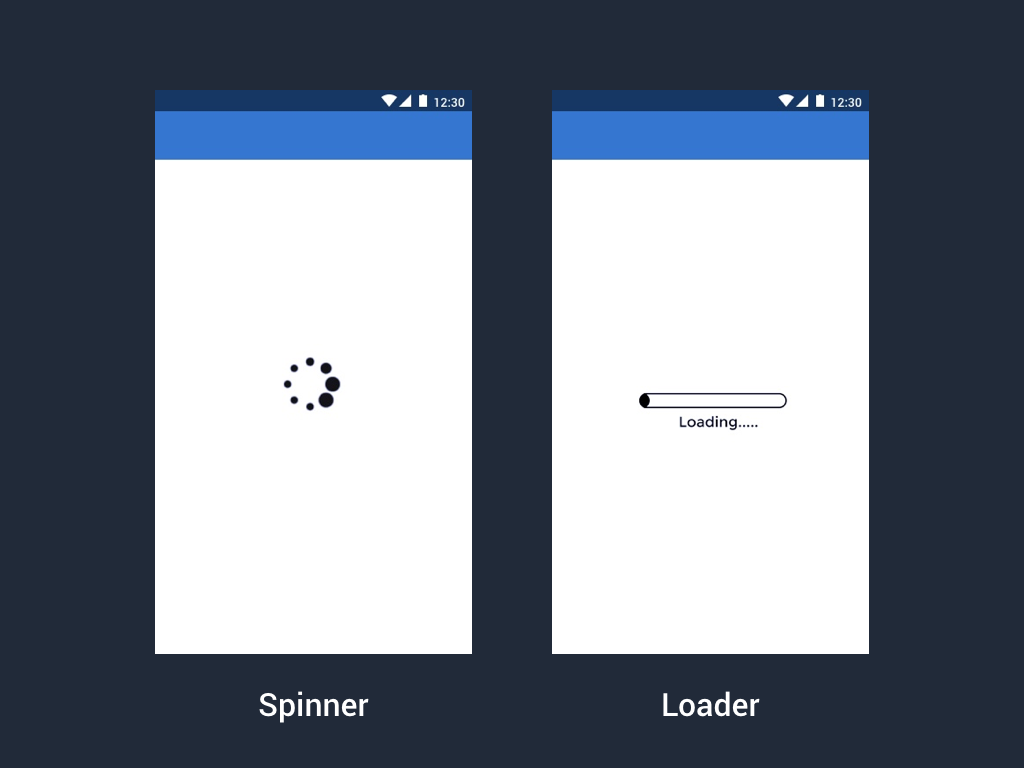
</body>

</html>

Luckily this can be automated with a node.js package but it is not the aim of the second week so I might skip it for now. But we can find more info about it [here](https://www.smashingmagazine.com/2015/08/understanding-critical-css/).

There are some other options such as loaders, spinners or skeleton screens -

Because the loading time is unclear, using a **loader/spinner** generates a period of uncertainty for the user. It emphasizes the fact that the user is currently waiting. The screen abruptly transitions, leaving the user perplexed and raising several queries in his or her mind.



**Skeleton screens** are a low fidelity UI into which information is gradually loaded. It gives users a visual cue that the content is being loaded into each UI element.



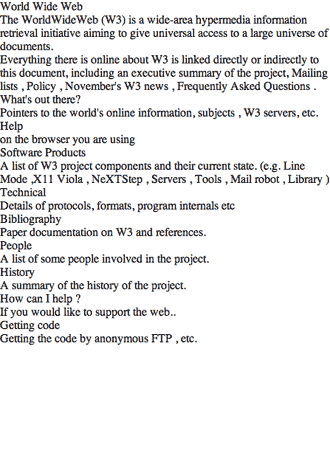
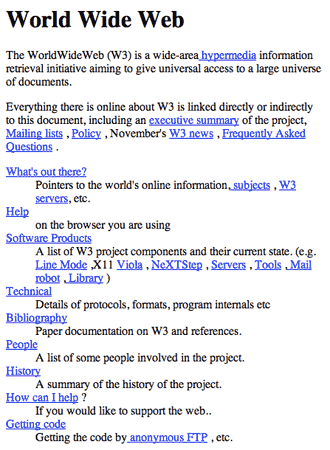
* Skeleton Screen when used as a feature in the product makes people perceive that the app/website is loading fast.
* By designing the skeleton screen that leverages motion(L to R follows the natural eye movement) helps in decreasing the perceived duration time.
* Further using a skeleton screen with progressive loading is thought of benefit for the user. In progressive loading an individual element becomes visible as soon as it is loaded, it helps in displaying the content that is exactly loaded and what is yet to be loaded. **Using these features all together act as a clear indication of progress.**

**CSS RESET**

A **CSS Reset** (or "Reset CSS") *is a set of CSS rules that resets the styling of all HTML components to a consistent baseline.*

Each browser has its own 'user agent' stylesheet that utilizes to make unstyled websites look more readable. Most browsers, for example, make links blue and visited links purple by default, add a border and padding to tables, apply various font sizes to H1, H2, H3, and practically everything, and apply a specific amount of padding to almost everything.

Example of the first web created, without and with CSS reset.



*This causes developers a lot of grief because they can't figure out how to make their websites look the same in every browser.*

*Developers can use a CSS Reset to cause every browser's styles to be reset to null, minimizing cross-browser differences as much as possible.*

**Normalize.css**

**Normalize.css** is an alternative to **CSS resets**. Is an open source .css file that [*Nicolas Gallagher*](http://necolas.github.io/normalize.css/) made on GitHub. It allows us to apply a "reset" to our code in order to let most modern browsers use our CSS.

To use it, we need to link it to our HTML like we would any other CSS, but making sure it's above our main stylesheet so that the "reset" is applied before our own styles.

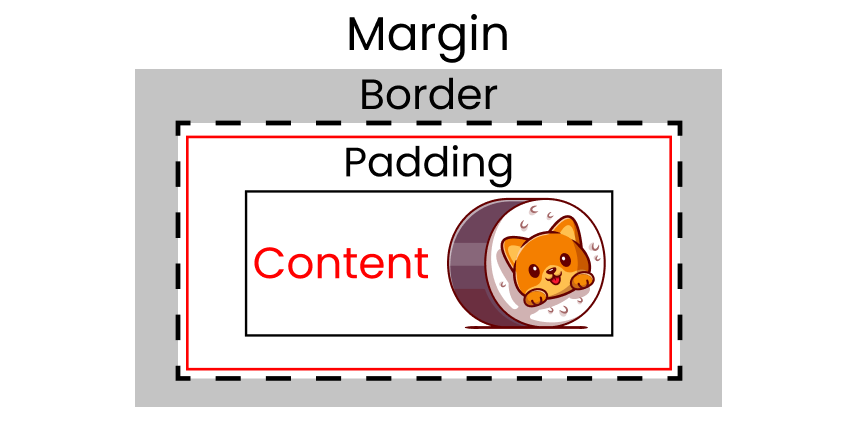
**Box model**

The CSS box model is a container that contains multiple properties including borders, margin, padding, and the content itself. It is used to create the design and layout of web pages. It can be used as a toolkit for customizing the layout of different elements.

Everything in CSS has a box around it, and understanding these boxes is key to being able to create layouts with CSS, or to align items with other items.

Box model defines how the different parts of a box — **margin**, **border**, **padding**, and **content** — work together to create a box that we can see on a page.

* **Content box**: Is the area where our content is displayed.
* **Padding box**: The padding sits around the **content** as white space. Unlike margins, cannot have negative values, hence the value must be 0 or positive. Padding is commonly used to push content away from the edge of a page.
* **Border box**: The border box wraps the **content** and any **padding**.
* **Margin box**: The margin is the outermost layer, wrapping the **content**, **padding**, and **border** as whitespace between this box and other elements. Margin values might be positive or negative. When we employ a negative margin on one side of our box, it may overlap other elements on the page.



**Box sizing**

It applies to all of the elements that accept width and/or height.

The width and height we provide to an element in the CSS box model are applied exclusively to the element's content box by default.

*Let’s say we set the width of an element to 500px, the content box will be 500px wide, and the width of any border or padding will be added to the final rendered width, making the final element wider than 500px.* This is the default behavior **(content-box)**.

On the other hand we have **border-box** which tells the browser to take any border and padding into consideration when calculating the width and height of an element.

*When we set the width of an element to 500px, that width includes any border or padding you apply, and the content box shrinks to accommodate the extra width worth taking in notes that do not include the margin since it is outside the box.*

Javascript

<https://www.educative.io/edpresso/what-are-expressions-in-javascript>