

---

# UD01 - Introducció a HTML5, CSS i eines de preprocessat

---

## Índex

- Introducció
  - Història
    - Abans de HTML5
    - A partir d'HTML5
  - Ecosistema HTML 5
    - Llenguatge de Marques (Markup)
      - Elements (Tags)
        - Anatomia
        - Custom Elements
      - Atributs
        - Atributs "data-xxx"
      - Estructura de l'HTML
        - Elements de la Capçalera
        - Elements del Cos
      - Referències
    - Formularis
      - Nous tipus de camps a HTML5
      - Nous atributs per a camps de formularis a HTML5
      - Validació de Formularis
      - Referències
    - Javascript
      - ECMAScript
        - Strict Mode
          - Activació
        - Variables
        - Àmbit (Scope)
        - Closures
        - This
      - ES6+
        - Noves Funcionalitats
        - Compatibilitat cap Enrere
          - Polyfill
          - Transpiladors
      - Referències
    - Exercici 1:
  - Entorn de Treball
    - Editor de Text
    - Servidor Web i Motor de Plantilles PUG.
      - Prerequisits

- Procediment
- Referències
- Preprocessadors
  - Preprocessadors HTML
  - Preprocessadors CSS
  - Preprocessadors Javascript
  - Referències

# **Introducció**

Benvinguts al curs d'HTML5 de l'EBAP.

Aquest és un curs avançat. Motiu pel qual es pressuposen uns mínims coneixements previs d'HTML, CSS i Javascript.

En aquesta unitat didàctica repassarem breument els conceptes més importants, sense entrar en detalls, per assegurar que tots tenim, més o menys, els mateixos coneixements bàsics que ens seran clau per a poder seguir la resta del curs.

Si alguna cosa no l'entendem o ens resulta nova, és important preguntar al professor i/o investigar pel nostre compte a fi de poder entendre millor la resta d'unitats didàctiques.

# Història

## Abans de HTML5

- No hi havia un estàndard clar.
  - O millor dit: N'hi havia molts de discrepants.
- Imperava la "guerra dels navegadors".

## A partir d'HTML5

```
<!DOCTYPE HTML>
```

Al contrari que les versions anteriors, HTML5 és una especificació oberta que va creixent al llarg del temps.

Abans els navegadors aplicaven estrictament les normes imposades per l'especificació corresponent al DTD.

I si no hi havia DTD, reportaven un error per la consola (que l'usuari corrent normalment no veu) i, "feien el que podien".

Al final un s'adonava que de vegades era millor treure DTD perquè així solien funcionar totes les funcionalitats de les distintes especificacions.

**A partir d'HTML5, aquell "feien el que podien" passa a ser la norma:** Si el navegador no reconeix un tag, un atribut o es troba amb qualsevol altra circumstància que no sigui amb les especificacions que ell coneix, ha d'intentar resoldre-ho "de la millor manera possible" per tal de minimitzar l'impacte de cara a l'usuari.

**Per exemple:** un navegador que no conegui el tag `<progress>`, el tractarà com si en el seu lloc hi hagués un simple `<div>`

Això no vol dir que puguem fer el que vulguem ni que ja no hagem de fer servir validadors d'HTML:

- Primer perquè sempre és bo verificar que el nostre codi s'ajusta als estàndards actuals.
- I segon perquè és fàcil cometre errors quan escrivim html i, tot i que els navegadors intentaran endevinar "què volíem dir realment", sempre n'hi haurà que hi tindran més traça que d'altres..

# Ecosistema HTML 5

- Markup (HTML)
- Fulls d'Estil (StyleSheets)
- Javascript
- DOM
- HTML5 APIs

## Llenguatge de Marques (Markup)

### Elements (Tags)

#### Anatomia

Amb contingut:

```
<div class="menu">
  (contingut...)
</div>
```

Auto-tancats:

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

#### Custom Elements

Els elements de client o "custom" son aquells en els que el nom del tag està en minúscules i conté al menys un guió ("-"). Llevat d'algunes excepcions que no vénen al cas.

Els elements "custom" no seran reportats com a "desconeguts" per un validador. I la seva finalitat és la de poder implementar components específics del costat del navegador.

```
<my-component>
  (contingut...)
</my-component>
```



React, Polymer i Angular son alguns dels frameworks més coneguts que en fan ús.

### Atributs

- Els noms del atributs és "case-insensitive" a HTML5.

#### Atributs "data-xxx"

De forma semblant als *Custom Elements*, els atributs el nom dels quals comenci per "data-", son considerats atributs d'usuari.

Aquests atributs es poden fer servir lliurement a qualsevol element sense que mai cap validador ens alerti de que no el reconeix i, **el que és molt més important:** sense el perill que nous atributs que s'estandarditzin en el futur hi puguin col·lisionar.

Resulten **molt útils** per atribuir informació semàntica a elements determinats que després podem fer servir tant des dels nostres controladors javascript com, si cal, des del full d'estil CSS.

## Estructura de l'HTML

- DTD
- Document (<html>)
  - Capçalera (<head>)
  - Cos (<body>)

### Elements de la Capçalera

- Títol (<title>)
- Codificació de caràcters: <meta charset='utf-8' />
- Viewport: <meta name="viewport" content="width=device-width, initial-scale=1">
- Fulls d'estil
  - Externs: <link rel="stylesheet" type="text/css" href="..." />
  - Interns: <style>...</style> (També poden anar dins el cos)
- Scripts (?)
  - Externs: <script src="..." /> (També poden anar dins el cos\*)
  - Interns: <script>...</script> (També poden anar dins el cos\*)



Avui en dia els scripts es solen ubicar al propi cos del document i, preferentment, al final per dues raons:

1. Per evitar retardar la descàrrega (i, com a conseqüència, també la renderització) del document.
2. Perquè si el script necessita capturar elements del DOM, no pot fer-ho fins que aquests s'hagin creat. **I aquesta és la millor manera d'assegurar-ho.**

### Elements del Cos

Els mes comuns son:

- <div></div>
- <span></span>
- <h1></h1>, <h2></h2>, <h3></h3>, ...
- <a href="..."></a>
- <p></p>
- <br />
- <pre></pre>
- etcètera...

Llistes:

- <ul></ul> / <ol></ol>
  - <li></li>

Taules:

- `<table></table>`
  - `<thead></thead>` / `<tbody></tbody>` / `<tfooter></tfooter>`
  - `<tr></tr>`
  - `<td></td>` / `<th></th>`

Nous a HTML5

- `<header></header>`
- `<footer></footer>`
- `<section></section>`
- `<article></article>`
- `<aside></aside>`
- etcètera...

## Referències

- **Elements HTML5:**
  - Nous elements HTML5: [https://www.w3schools.com/html/html5\\_new\\_elements.asp](https://www.w3schools.com/html/html5_new_elements.asp).
  - Referència completa: [https://www.tutorialspoint.com/html5/html5\\_tags.htm](https://www.tutorialspoint.com/html5/html5_tags.htm).
- **Atributs HTML5:**
  - Referència completa d'atributs: [https://www.w3schools.com/tags/ref\\_attributes.asp](https://www.w3schools.com/tags/ref_attributes.asp).
- **Altres:**
  - Viewport: [https://developer.mozilla.org/en-US/docs/Mozilla/Mobile/Viewport\\_meta\\_tag](https://developer.mozilla.org/en-US/docs/Mozilla/Mobile/Viewport_meta_tag).
- **Frameworks:**
  - React: <https://reactjs.org/>.
  - Polymer: <https://www.polymer-project.org/>.
  - Angular: <https://angular.io/>.

## Formularis

Exemple:

```
<form>
  <input type="text" name="nom" placeholder="Escrigui el nom">
  <input type="text" name="cognoms" placeholder="Escrigui els cognoms">
  <button type="submit">Enviar</button>
</form>
```



A l'exemple anterior probablement hi trobareu a faltar els atributs *action* o *method* i pot ser hegeu notat que hem fet servir un `<button>` en comptes d'un `<input type="text">`.

No és que ja no hi siguin a HTML5. Però moltes vegades no seran la forma més idònia de gestionar els formularis:

- En comptes d'esperar a que l'usuari envii el formulari per a validar-lo, normalment preferirem advertir a l'usuari tot d'una que detectem alguna cosa que no estigui bé.
- Tampoc és ja gaire habitual enviar un formulari recarregant tota la pàgina. El més usual és enviar les dades via *Ajax* i realitzar alguna modificació al



document per a que l'usuari se'n adoni que la informació ha estat processada.

## Nous tipus de camps a HTML5

|       |        |                |
|-------|--------|----------------|
| color | date   | datetime-local |
| email | month  | number         |
| range | search | tel            |
| time  | url    | week           |



Els tipus de camps que no estiguin suportats en un navegador antic, es comportaran igual que si fossin `<input type="text">`.

## Nous atributs per a camps de formularis a HTML5

- **placeholder:** Ens permet especificar un text (o "pista") que apareixerà difuminat al fons dels camps que no estiguin emplenats.
- **pattern:** Ens permet especificar una expressió regular de validació.
- **required:** Ens permet especificar quan un camp és obligatori.
- **autofocus:** Marca un camp indicant que és ell qui ha de rebre el focus en carregar-se la pàgina.

Altres...

|              |            |                |
|--------------|------------|----------------|
| autocomplete | form       | formaction     |
| formenctype  | formmethod | formnovalidate |
| formtarget   | height     | width          |
| list         | min        | max            |
| multiple     | step       |                |

## Validació de Formularis

## Referències

## Javascript

Javascript és un llenguatge de programació **asíncron**:

- Orientat a events
- ...que son gestionats mitjançant *callbacks*
- ...que s'enqueuen al *event-loop*.

Va ésser desenvolupat en una setmana per encàrec de *Netscape* amb la intenció de dotar els seus navegadors de certa interactivitat.

Els programes en Javascript s'executen en un sol fil del processador. El que ens permet programar **gairebé** com si no existís concurrència.

El programari que interpreta el codi Javascript s'anomena "motor" (de l'anglès "engine"). El més utilitzat avui en dia és el "V8" de Google.





El motor, a més d'interpretar el codi Javascript, serveix de pont entre aquest i l'entorn en el que s'ha d'executar el nostre programa.

Així, en un programa que s'executi en un navegador, el nostre entorn serà el DOM, que ens dona accés a poder manipular totes les característiques del document i, fins i tot, algunes del propi navegador.

Per programes executats en un intèrpret com ara Node.JS, en canvi, el nostre entorn és el sistema operatiu i tots els serveis que aquest ens ofereix.

Per contra del que molta gent pensa, Javascript és un llenguatge fortament orientat a objectes. Fins al punt que fins i tot els seus tipus de dades més elementals com les cadenes de text o els valors numèrics, son en realitat objectes.



El que no és en realitat és un llenguatge "orientat a classes": No cal una *classe* per crear un objecte en Javascript.

També té herència. Però el que s'hereden son els *prototipus* i no les *classes*.

De fet tota la interacció amb el món exterior i fins i tot amb el propi llenguatge (com ara les llibreries de funcions matemàtiques, de temps, expressions regulars, etc...) es fa a través d'objectes (variables) predefinits que el motor ens presenta a mode d'interfície per a interactuar amb ells.

Així, a qualsevol entorn Javascript ens trobarem objectes com:

- Math
- Date
- RegExp
- Object
- String
- Number
- Array

...i d'altres només els trobarem en entorns determinats:

- Node.JS:
  - process
  - fs
- Navegador:
  - document
  - window

...un cas especial és el de l'objecte `console` que ens el trobarem sempre (pel que podriem dir que pertany al primer grup), amb els seus arxiconeguts mètodes `console.log()` i `console.error()`, però que el cito apart perquè, tot i presentar una interfície idèntica en tots els entorns, i realitzar una funció, efectivament idèntica. La forma en que es materialitza aquesta interacció pot ser radicalment distinta:

- A qualsevol intèrpret de terminal, com Node.JS: Ens mostrarà el resultat per la pantalla (consola).
- En un navegador, en canvi: Els missatges romanen ocults a no ser que nosaltres accediguem a la *consola del navegador* (que típicament s'obre amb la tecla F12).



## ECMAScript

Javascript es va estendre ràpidament per tots els navegadors de l'època. Si be cadascun en feia les seves pròpies variacions i, per això, durant molts d'anys va resultar caòtic programar en Javascript i que el nostre codi funcionàs bé a tots els navegadors de l'època.

L'associació **ECMA**: (European Computer Manufacturers Association) va voler estandarditzar-lo. Però el terme "Javascript" és una marca comercial registrada i l'aleshores titular, *Netscape*, no va permetre el seu ús. Motiu pel qual, el llenguatge va ésser estandarditzat sota el nom de "ECMAScript".

Avui en dia però, sempre que col·loquialment parlem de "Javascript" s'entén que en realitat estem parlant d'ECMAScript.

### Strict Mode

En els seus orígens Javascript era un llenguatge molt lax i, entre d'altres coses, no requeria declarar les variables: Si fèiem servir un identificador que no apareixia declarat al codi, automàticament es creava com a variable d'àmbit global.

Així, codi com el següent:

```
var Comptador = 10;
comptador--;
console.log(comptador);
```

...retornaria -1 (quan nosaltres probablement esperaríem un 9) en comptes de donar un error per variable no declarada, que és el que realment hauria de fer (en Javascript els identificadors són *Case-Sensitive*).

Per poder solucionar això i no rompre la compatibilitat amb codi antic, es va introduir el que es coneix com a "mode estricte".

Així, fins i tot els motors de Javascript més moderns funcionen en un mode compatible amb aquelles regles fins que nosaltres explícitament activem el mode estricte.

### Activació

Per activar el mode estricte basta amb posar la següent cadena (cometes incloses) al principi del nostre codi:

```
"use strict";
```

Quan els motors de Javascript moderns veuen aquesta cadena entenen que han d'aplicar les regles modernes del Javascript (ECMAScript).

A més, com que escriure un valor d'un tipus de dades donat (en aquest cas una cadena), encara que no l'assignem a cap variable ni en fem res d'especial és vàlid com a codi en qualsevol versió de Javascript. Si no utilitzam funcionalitats avançades no disponibles en ells, el nostre codi **també funcionaria amb motors antics**.



En resum: **SEMPRE** que ens disposem a escriure codi en Javascript, el primer que farem serà posar la cadena "use strict"; al principi.



Això ens ajudarà a fer millor codi i ens evitarà molts de problemes...

## Variables

Tenim tres formes de declarar variables:

```
var foo; // Function (closure) level scope
let bar; // Block ({ ... }) level scope
const foo; // Block-level, not de-referenciable.
```

- Només `var` funcionarà en versions antigues (però hi ha transpiladors que ens ho arreglen).
- Amb `let` i `const` no hi ha *hoisting*: (No poden ser referenciades abans de la declaració). Ni tampoc fora del bloc en el que s'han definit.
- Les variables en Javascript son *referències* a objectes: **Amb `const` l'únic que és constant és la referència**: El contingut de l'objecte sí pot canviar.



Com a bona pràctica (sempre que ens ho puguem permetre) farem servir preferentment *const*. Si no ens serveix *let* i, només com a darrer recurs *var*.

## Àmbit (Scope)

L'àmbit d'una variable determina a quines parts del codi aquesta és visible.

- Les variables *globals* son aquelles que son visibles des de qualsevol part del programa.
  - Fora del mode estricte les variables invocades sense haver estat prèviament declarades (ex.: `a = 23;`) es declaren automàticament a l'àmbit global. **Per això SEMPRE hem d'activar el mode estricte.**
  - En el mode estricte només es poden declarar variables globals amb `var` i exclusivament fora de qualsevol *clausura*.
  - En realitat, les variables d'àmbit global son propietats del *global object*. Així `var foo="bar"` (fora de qualsevol clausura) i `window.foo="bar"` serien declaracions equivalents.
  - El *global object* és `window` als navegadors i `process` a *Node.JS*. Existeix la proposta d'estandarditzar pròpiament `global` com a referència a `window` o `process`, respectivament.
- L'àmbit de les variables declarades amb `var` és el de la *clausura* dins la que han estat declarades o l'àmbit global en el cas que de no haver estat declarades dins cap.
- L'àmbit de les variables declarades amb `let` o `const` és el del *bloc* de codi en el que han estat definides.
  - Els *blocs de codi* es delimiten amb `{ }` ja sigui com a predicat d'una sentència (`if (x) { ... }`) o simplement posats expressament amb la finalitat de limitar l'àmbit de les variables del seu interior (`{let i=0; ...}`):
  - Un cas especial és el de la sentència *for* on al codi `for (let i=0; i<arr.length; i++) { ... }`, l'àmbit de `i` correspondria a l'interior el bloc `{ ... }`.

```
const arr = [1, 2, 3, 4]
for (let i=0; i<arr.length; i++) {
  arr[i] += 2;
};
console.log (arr); // [3, 4, 5, 6]
console.log (i); // Error!!
// Això amb var no passava...
```

## Closures

Les clausures delimiten l'àmbit de visibilitat d'una variable.

Es creen automàticament cada cop **que invocam** una funció (no en el temps de creació) pel que cada cop que invoquem de nou la mateixa funció s'en crearà una de nova totalment separada de la anterior.

En canvi, **els objectes que aquestes referncien, també poden ser referenciats des de l'exterior:**

```
function sumador(n) { // Això no crea cap clausura.
  let x = n; // (en realitat hauriem pogut fer servir 'n'
  let fn = function suma(m) {
    x += m;
    return x;
  };
  return fn;
};

let s1 = sumador(0); // Això sí crea una clausura.
let s2 = sumador(100); // I això una altra...

console.log (s1(3)); // 3
console.log (s1(5)); // 8
console.log (s2(5)); // 105
```

Les variables creades dins una clausura només son visibles dins d'aquesta **i dins les que es crein dins ella**. D'això s'en diu *Runtime Scope* o àmbit en temps d'execució.

Una utilitat molt pràctica de les clausures és fer-les servir per aïllar petits blocs de codi (o sub-controladors) de manera que les variables que ells declarin no puguin veure's afectades pel codi de l'exterior. Si be, avui en dia, pels casos més senzills, ens bastarà fer servir `const` o `let`:

```
// Petit controlador secundari:
(function(model) {
  ... implementació ...
})(model.submodel.foo.bar);
```



El patró anterior es coneix pel nom de *IFE*: (*Immediately Invoked Function Expression*).

La seva forma més senzilla és `(function(){...})()`. Però podem fer les variacions que vulguem: com ara fer servir paràmetres per evitar referenciar variables externes des de l'interior, igual que a l'exemple anterior, fer que ens retornin un valor que podem assignar a una variable i/o assignar-li un nom a la



```
funció: const myCoolObj = (function coolObjBulder(options)
{...;return obj})(config);
```

## This

*this* és la propietat més àmpliament incompresa de Javascript.

*this* implementa el *Call Site Scope* (en contraposició amb el *Runtime Scope* que acabam de veure).



Explicar a fons *this* queda més enllà dels objectius d'aquest curs, però per a qui tingui interès en aprofundir, us deixo a les referències un enllaç a l'article "Understanding Javascript OOP".

.....

## ES6+

### Noves Funcionalitats

- **Let and Const.**
- **Template literals.**
- **Arrow Functions.**
- **Destructuring.**
- **class:** (△ L'herència continua essent de prototip)



A l'apartat de *referències* teniu alguns articles sobre aquests temes.

### Compatibilitat cap Enrere

Quan desenvolupam codi que ha de ser interpretat per un navegador, tenim el problema afegit que no sabem com d'antic pot arribar a ser aquest.

Com més compatibles vulguem ser, menys noves característiques del llenguatge podem fer servir...

...o **NO**.

En Javascript tenim dues eines molt potents que ens permeten fer servir fins i tot les més recents funcionalitats del llenguatge encara que el nostre codi s'executi en motors antics.

Aquestes eines son els *polyfills* i els *transpiladors* (o "transpilers").

### Polyfill



Modernizr

Transpiladors

Babel

## Referències

- Understanding Javascript OOP: <https://robotlolita.me/2011/10/09/understanding-javascript-oop.html>.
  - 4 Modern ES6+ Features You Should Be Using Now:  
<https://programmingwithmosh.com/javascript/essential-modern-javascript-features/>.
  - Modernizr: <https://modernizr.com/>.
-

## Exercici 1:

Redactar una carta de presentació, estil currículum en HTML5. Ha de contenir com a mínim:

- El vostre nom i llinatges.
- Els vostres coneixements i experiència en HTML i/o altres tecnologies.
- Motiu que vos ha impulsat a fer aquest curs.

Heu d'aplicar el que heu après fins ara. Però podeu anar més enllà si en sabeu (incloure una fotografia, aplicar estils, etcètera...).

---

# Entorn de Treball

Per poder seguir aquest curs necessitarem principalment tres coses:

1. Un editor de text que suporti ressaltat sintàctic.
2. Un servidor web per poder visualitzar els exercicis que anem fent no només al nostre ordinador, sinó com a mínim també al nostre smartphone.
3. També farem servir un motor de plantilles HTML. Concretament el *Pug* (abans conegut com a "Jade"). Això ens permetrà:
  - Generar HTML molt més ràpidament.
  - Amb menys propensió a errors (perquè la sintaxi és més clara i perquè si cometem errors greus el compilador els detectarà).
  - Fer servir models de dades externs (el que seria separar la vista de les dades en un model MVC).



# Editor de Text

Podeu fer servir el que vulgueu si el teniu disponible.

Als ordinadors de l'aula trobareu preparat el Sublime Text.

Si el voleu fer servir també a casa, heu de saber que Sublime Text no incorpora per defecte el ressaltat sintàctic per a fitxers *Pug*. Però se li pot afegir mitjançant el seu sistema de paquets:

- Menú Preferences -> Package Control
- Teclejam 'pi' i sel·leccionam "Install Package"
- Teclejam 'Pug' i sel·leccionam "Pug"

# Servidor Web i Motor de Plantilles PUG.

Com a Servidor Web, farem servir *Express*. Que és un framework per *Node.JS* disponible al repositori de paquets *NPM*.

Explicar *Node.JS* i el funcionament de *Express* està fora dels objectius d'aquest curs. Simplement els farem servir perquè son la forma més ràpida i senzilla d'obtenir un entorn de treball amb les eines que necessitam.

Així que, seguidament, ens limitarem a relacionar les passes a seguir per crear un projecte *Express* per poder començar a treballar:

## Prerequisites

- *Node.JS* i *NPM*: Els ordinadors de l'aula ja els tenen instal·lats.
  - A l'apartat de [referències](#) trobareu més informació si voleu instal·lar-vos-els a casa.
- *Express-generator*: És una eina en línia de comandes que ens permetrà crear projectes *Express* funcionals només executant una comanda.
  - Si no el tenim ja instal·lat, l'instal·larem amb la comanda `npm install -g express-generator`.

## Procediment

Obrirem una terminal i seguirem els passos següents:

1. Cream un directori, per exemple de nom "exercicis", on emmagatzemar els nostres exercicis i projectes i ens situam en ell.

```
u@m:~$ mkdir exercicis
u@m:~$ cd exercicis
u@m:~/exercicis$
```

2. Repetirem el pas anterior per crear a dins d'aquest un subdirectori, per exemple "cursHTML5" per al nostre primer projecte.

```
u@m:~/exercicis$ mkdir cursHTML5
u@m:~/exercicis$ cd cursHTML5/
u@m:~/exercicis/cursHTML5$
```

3. Fem servir la comanda `express` (l'*express-generator*) per crear un projecte buit.

Farem servir el paràmetre `--view=pug` per indicar-li que volem fer servir específicament aquest motor de plantilles:

```
u@m:~/exercicis/cursHTML5$ express --view=pug
```

```
create : public/  
create : public/javascripts/  
create : public/images/  
create : public/stylesheets/  
create : public/stylesheets/style.css  
create : routes/  
create : routes/index.js  
create : routes/users.js  
create : views/  
create : views/error.pug  
create : views/index.pug  
create : views/layout.pug  
create : app.js  
create : package.json  
create : bin/  
create : bin/www
```

```
install dependencies:  
$ npm install
```

```
run the app:  
$ DEBUG=curshtml5:* npm start
```

Això ens haurà creat una estructura de fitxers i directoris com aquesta:

```
.  
├── app.js  
├── bin  
│   └── www  
├── package.json  
├── package-lock.json  
├── public  
│   ├── images  
│   ├── javascripts  
│   ├── stylesheets  
│   └── style.css  
├── routes  
│   ├── index.js  
│   └── users.js  
└── views  
    ├── error.pug  
    ├── index.pug  
    └── layout.pug
```

Les dues següents passes ja ens les ha suggerit el propi express-generator...

4. Executem `npm install` per tal que s'instal·lin totes les dependències:

```
u@m:~/exercicis/cursHTML5$ npm install  
npm notice created a lockfile as package-lock.json. You should  
add it to your git repository  
added 118 packages from 174 contributors and audited 247 packages  
found 0 vulnerabilities
```

i...

5. Arrancam el nostre nou servidor web. Podem fer servir la comanda que s'ens o, en el nostre cas, amb `npm start` serà suficient.

```
u@m:~/exercicis/cursHTML5$ npm start

> curhtml5@0.0.0 start /home/usuari/exercicis/cursHTML5
> node ./bin/www
```

Ara hauríem de poder accedir al nostre servidor web a través del port 3000 del nostre PC. És a dir: a la url <http://localhost:3000>.

- També, si tenim el nostre mòbil connectat a la xarxa inalàmbrica i coneixem l'adreça IP del nostre ordinador (que podem obtenir amb les comandes `ifconfig` o `ip addr list`), hi podrem accedir canviant "localhost" per la nostra IP a l'enllaç anterior.

Si ara desam el fitxer que hem creat a l'exercici anterior sota el directori "public" i amb el nom "curriculum.html", veurem que podem accedir a ell sota al url <http://localhost:3000/curriculum.html> (o amb la IP corresponent si volem poder accedir-hi també des del nostre smartphone).

# Referències

- *Sublime Text*: <https://www.sublimetext.com/>.
- Instal·lació Node.JS i NPM:
  - Via *NVM* (recomanat): <https://github.com/creationix/nvm/blob/master/README.md#install-script>.
  - Amb el gestor de paquets del Sistema Operatiu: <https://nodejs.org/es/download/package-manager/>.
  - Instal·lació manual: <https://nodejs.org/en/download/>.
- Altres:
  - Motor de JavaScript V8 de Google: [https://ca.wikipedia.org/wiki/V8\\_\(int%C3%A8rpret\\_JavaScript\)](https://ca.wikipedia.org/wiki/V8_(int%C3%A8rpret_JavaScript)).

# Preprocessadors

Els preprocessadors son eines que ens permeten transformar un fitxer de codi escrit en un llenguatge modificat o totalment distint al que entén compilador o intèrpret al que estan destinats.

Per exemple, el motor de plantilles *Pug* és un preprocessador que transforma una plantilla *Pug* en un document HTML vàlid que pot llegir qualsevol navegador.

Al projecte *Expres* que hem creat anteriorment, podem veure com a dins el directori *views* tenim tres fitxers:

error.pug index.pug layout.pug

## **layout.pug:**

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

## **index.pug:**

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

## **error.pug:**

```
extends layout

block content
  h1= message
  h2= error.status
  pre #{error.stack}
```

## **Definicions:**

- shims
- polyfills
- transpilers

# Preprocessors HTML

# Preprocessors CSS

- PostCSS: <https://postcss.org/>.
  - Autoprefixer: <https://github.com/postcss/autoprefixer>.



# Preprocessors Javascript

- Babel

# Referències

- Motors de Plantilles
  - Pug (Jade): <https://pugjs.org>
  - Handlebars: <https://handlebarsjs.com>
- ECMAScript:
  - Modernizr: <https://modernizr.com/>.

- 
- 
- Frameworks · jQuery · [You might not need jQuery](#)