UD01 - Introducció a HTML5, CSS i eines de preprocessat

Índex

Introducció

Benvinguts al curs d'HTML5 de l'EBAP.

Aquest és un curs avançat. Motiu pel qual es pressuposen uns certs coneixements previs.

Com veurem al llarg del curs, HTML5 no és només el llenguatge de marques (HTML), sino tot un ecosistema que comprèn aquest, els fulls d'estil (CSS), el llenguatge de programació ECMASCRIPT (Javascript) i tota una sèrie d'APIs que ens permeten des d'emmagatzemar dades als ordinadors de l'usuari (*LocalStorage*) fins a interactuar amb el sistema operatiu d'aquest ja sigui per obtenir les coordenades on es troba ubicat el dispositiu, la orientació de la pantalla, consultar els contactes, o fer vibrar el dispositiu, entre moltíssimes altres possibilitats.

Però el gran desconegut amb diferència i no perquè li manqui importància és CSS. Dominar bé els conceptes fonamentals de CSS pot marcar la diferència a l'hora de treballar amb HTML5.

Per aquest motiu, a més fer un breu repàs al HTML aprofundint només en les novetats que incorpora HTML5, aquest curs es centrarà principalment en aprofundir en els fulls d'estil (CSS).

Addicionalment, al final de cada unitat didàctica, veurem una mica de Javascript, el qual ens ajudarà a acabar d'arrodonir els nostres projectes en HTML5.

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'entenem 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.

Hi ha molta matèria, però la major part és informació que molts ja coneixereu i que, en qualsevol cas, es pot trobar molt fàcilment a Internet.

De fet, l'objectiu del curs no és que memoritzeu res: Sinó que assimileu els conceptes bàsics i sigueu capaços de trobar la documentació necessària quan la necessiteu.

Al final d'aquesta primera unitat didàctica, veurem com implementar un senzill servidor web amb *Node.JS* i *Express* que ens servirà per penjar-hi els exercicis que anem desenvolupant i poder contrastar així com es veu el resultat tant al nostre ordinador com, a través de la xarxa Wifi, també als nostres telèfons mòbils.

Història

Abans de HTML5

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

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

Per aquest motiu es simplifica el *doctype* que passa a ser simplement:

```
<!DOCTYPE HTML>
```

A partir d'HTML5, aquell "feien el que podien" d'abans passa a ser la norma: Si el navegador no reconeix un tag, un atribut o es troba amb qualsevol altra circumstància que no casi 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.
  - Referència completa: https://www.tutorialspoint.com/html5/html5_tags.htm.
- **Atributs HTML5:**
  - Referència competa d'atributs: https://www.w3schools.com/tags/ref_attributes.asp.
- **Altres:**
  - Viewport: 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 name="nom" type="text"
      placeholder="Escrigui el seu nom"
  >
  <input name="cognoms" type="text"
      placeholder="Escrigui els seus cognoms"
  >
  <button type="submit">
</form>
```

A l'exemple anterior probablement hi trobareu a faltar els atributs *action* o *method* i pot ser hageu 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.

Tampoc hem fet servir el tag `<label>`. Posar etiquetes és perfectament vàlid i, fins i tot recomanable en molts casos. Però de vegades, quan les dades son suficientment autoexplicatives, utilitzar només l'atribut *placehloder* per indicar què va a cada camp abans d'emplenar-lo, ens pot ajudar a assolir un aspecte molt més net.

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 fóssin `<input type="text">`.

Nous atributs per a camps de formularis a HTML5

placeholder	pattern	required
autocomplete	form	formaction
formenctype	formmethod	formnovalidate
formtarget	height	width
list	min	max
multiple	step	autofocus

Validació de Formularis

La regla d'or en quant a validació de formularis és que aquesta **sempre es fa costat del servidor**.

Qualsevol validació que puguem fer al costat del client no només depen de que les funcionalitats que facem servir estiguin disponibles i funcionin correctament al navegador i versió concrets que faci servir l'usuari. Sinó també de que aquest no ens vulgui enganar i colar-nos uns valors que no compleixin les regles que nosaltres hem imposat.

Dit això, obligar a l'usuari a emplenar tot un formulari per al final de tot dir-li que ho ha fet tot malament i que ha de tornar a començar és poc menys que suggerir-li que no torni. Especialment si en el procés li esborram les dades que havia introduït i el feim començar de zero.

Per això, **A MÉS** de la validació principal al servidor, és recomanable que el formulari reaccioni a les accions de l'usuari proporcionant-li un feedback constant perquè pugui saber si ho està fent bé o malament.

HTML5 ens proporciona vàries eines per fer just això:

Placeholder

D'acord: Tècnicament l'atribut *placeholder* no té res a veure amb la validació de formularis.

Però en una situació com la següent:

```
<label for="data">Data</label>
<input name="data" type="text">
```

...afegir un atribut de l'estil de `placeholder="dd/mm/aaaa"` pot ser un bon començament.

Evidentment, fer servir `type="date"` seria una solució molt millor. Però amb navegadors (bastant) antics no funcionarà. En canvi, si combinam les dues solucions, probablement els usuaris dels navegadors més antics ens ho agrairan.

Nous tipus de camps

Considerem el següent exemple:

```
<label for="color">
    Indiqui quin és el seu color favorit
</label>
<input name="color" type="text">
```

Això podríem dir que `placeholder="Ex. #ff0000"` no ens ho arregla *massa*, precisament...

Afortunadament, en HTML5 tenim el tipus de camp *color* que ens incrustarà un flamant *color-picker* per a que l'usuari pugi triar el que més li agradi.

També, com hem dit, tenim el tipus *date* per a dates.

Altres que tenim son:

`time`	`datetime-local`	`month`
`week`	`number`	`range`
`search`	`tel`	`url`
`email`		

La forma d'implementar cadascun d'ells depen del navegador:

- En general alguns d'ells, com `color` o `range` ens presentaran un widget totalment personalitzat de manera que no podrem introduir valors invàlids de cap manera (tret que, per exemple, només vulguem acceptar dates dins d'un determinat període).
- Altres, com `tel`, `email`, etcètera... es mostraran com a camps de text normal permetent-nos editar el contingut lliurement, però mentre el valor introduït no sigui vàlid, presentaran un estil diferent (que per defecte depen del navegador però, com veurem més endavant, el podem personalitzar des del full d'estil).

 A l'apartat de *Referències* podeu trobar l'enllaç a la llista actualitzada completa.

Atributs de validació

pattern:

Desgraciadament no podem tenir un tipus de camp específic per a cada tipus de dades que eventualment puguem voler sol·licitar en un formulari.

A l'exemple següent:

```
<label for="url">Url</label>
<input name="url" type="url">
<label for="ip">Adreça IP</label>
<input name="ip" type="text" placeholder="Ex: 192.168.1.1">
```

...el *placeholder* és un bon intent. Però la realitat és que si introduïm una url invàlida ens en adonarem de seguida. Però si introduïm "257.30.25.13" (o simplement "Sancho Panza"...) al camp d'adreça IP, no hi haurà res que ens indiqui que el valor és invàlid.

Per solucionar això, l'atribut *pattern* ens permet especificar una expressió regular associada al camp de manera que, sempre que el contingut del mateix no la compleixi, adoptarà l'estat de "no validat".

```
<label for="ip">Adreça IP</label>
<input name="ip" type="text"
    placeholder="Ex: 192.168.1.1"
    pattern="((^|\.)((25[0-5])|(2[0-4]\d)|(1\d\d)|([1-9]?\d
>
```

 Les *expressions regulars* no acostumen a brillar per la seva llegibilitat i la de validar una IP, encara que sigui de versió 4, no és tampoc de les més senzilles.

Però un cop les dominam son un recurs impagable a l'hora de validar dades.

A la secció de *Referències* teniu dos enllaços al respecte:

- Un és un interessant recull d'*expressions regulars* comunes en la validació de formularis HTML5.
- L'altre, per aquells que estiguin interessats en aprendre'n més, és un portal on trobareu tota la informació que pugueu necessitar per dominar les *expressions regulars*.

required:

L'atribut *pattern* ens permet dotar al formulari de la capacitat de diferenciar per ell mateix si el valor d'un camp és vàlid o no.

Però no sempre és requerit emplenar tots els camps. De vegades alguns son opcionals i, si no estan emplenats difícilment podran complir el patró especificat.

Per això l'atribut *required* ens permet identificar aquells camps que realment son requerits. De manera que el formulari pugui saber quan ens ha de marcar com a error el fet que, eventualment, no estigui emplenat.

```
<input name="nom" type="text" placeholder="Nom*" required>
```

min, max i step:

En camps de tipus *number* i *range* ens permeten establir els valors mínim, màxim i la resolució d'increment/decrement.

minlength i maxlength:

En camps de text ens permeten establir una longitud mínima i màxima.

Referències

- Tipus de camps en HTML5: https://www.w3schools.com/html/html_form_input_types.asp
- Repositori d'expressions regulars comunes per HTML4: http://html5pattern.com/Miscs
- Tot sobre expressions regulars: https://www.regular-expressions.info/.
- Més sobre validació de formularis: https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation.

Javascript

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

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

La seva primera versió la va desenvolupar *Brendan Eich* en només 10 dies per encàrrec 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 serveïs 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.

Per aquest motiu, podem fer coses com:

```
> "hello world".length
11
> "hello world".split(" ")
[ 'hello', 'world' ]
```

> 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 origens 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 son *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 activam 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 poguem 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 `{` i `}` ja sigui com a predicat d'una sentència (`if (x) {...}`) o simplement posats expréssament 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* ón 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.

Les variables declarades 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.

En canvi, **els objectes que aquestes referencien, 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
```

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 *IIFE*: *(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 objectíus d'aquest curs, però per a qui tingui interés en aprofundir, us deixo a les referències un enllaç a l'article "Understanding Javascript OOP".

ES6+

Noves Funcionalitats

L'arribada de ES6 va suposar una fita històrica per Javascript (tècnicament ECMASCRIPT).

Amb ella arribaren noves característiques molt desitjades. Però el que és millor: Es va canviar la forma en que s'alliberaven les noves versions de l'especificació per un sistema molt més eficient i dinàmic:

Ara cada any s'allibera una nova versió. Per exemple, la d'enguany serà la ES2018.

Això és possible gràcies a que les noves funcionalitats poden ser provades molt abans gràcies a l'ús de polyfills i transpiladors que ens permeten executar codi que fa servir les darreres funcionalitats en motors que encara no les suporten.

Les següents son algunes de les noves característiques més interessants:

- **Paràmetres per defecte:** `function inc(step=1){this.counter += step}`.
- **Template Literals:**
  - Fins ara per definir cadenes de text literals podiem fer servir cometes simples o dobles. Ara s'afegeix el caràcter d'accent greu o *backtick* (`).
  - Les cadenes així definides s'anomenen *template literals* i entre d'altres característiques més avançades, son multi-línia (ja no cal posar `\n`, etc...) i permeten la interpolació de variables o expressions.
- **Destructuració:**
  - En objectes: `let {foo, bar} = {foo: "Hello", bar: "world!", baz: "lost"}`.
  - I arrays: `let [a, b] = [1, 2], [a, b] = [b, a]`
- **Arrow Functions:** `let double = n=>n*2`
  - Molt convenients per a implementar callbacks senzills (augmentant la legibilitat).
  - Al temps que **absolutament desaconsellables** en la resta de casos: Per a funcions llargues empitjoren la legibilitat i, a més, el seu comportament no és igual (hereten el *this* de la funció on han estat invocades).
- **Promeses** Que tractarem més endavant en abordar el tema de la *programació asíncrona*.

- **`let` i `const`:** De les que ja hem parlat.
- **El keyword *class*:** Que proveeix una sintaxi millor a l'hora de crear objectes, tot i que pot ser l'elecció del nom no fora la més adequada, doncs indueix a pensar que pugui funcionar com el keyword *class* d'altres llenguatges.
- **Mòduls ES6.**

 A l'apartat de *referències* us deixo un article que les explica en més profunditat.

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

Un *polyfill* és un bocí de codi Javascript que implementa una funcionalitat *futura* que encara no està disponible a la versió del motor que l'executa.

D'aquesta manera ens asseguram que el nostre codi, tot i fer ús d'una nova funcionalitat, s'executarà sense problemes fins i tot en motors que encara no la suportin.

Els *polyfills* apliquen una tècnica que s'anomena *detecció de funcionalitat* per saber si s'han d'aplicar o no. Així, si incloem un *olyfill* al nostre programa i aquest és executat per un motor que ja no el necessita, el *polyfill* s'autoinhibirà de manera que, de forma transparent, passarem a fer servir la implementació nativa.

Per exemple, un *polyfill* per assegurar la disponibilitat de *Promeses* tindria una forma semblant al següent:

```
(function(global) {
    if (! global.Promise) {
        global.Promise = class {
           ...
        }
    };
})(window || process);
```

 La tècnica de *detecció de funcionalitat* no només es fa servir per als *polyfills*: La podem fer servir també nosaltres sempre que ens enfrontem a la

situació de voler fer servir una funcionalitat que no estem segurs estarà disponible en tots els casos.

D'aquesta manera podem oferir una funcionalitat alternativa o simplement ometre-la però sense que afecti a l'estabilitat de la resta del codi.

Per exemple, algunes APIs d'HTML5, com la que ens permet activar la vibració del mòbil, difícilment estaran disponibles en navegadors PC.

Una de les llibreries de *Polyfills* més extensament utilitzada és *Modernizr*, que podeu trobar enllaçada a les *referències*.

Transpiladors

Promise és un objecte que, com hem vist, podem implementar nosaltres mateixos en Javascript i aplicar-lo, només si cal, en temps d'execució (*polyfill*).

Però quan es tracta però de construccions del llenguatge, com les sentències `let` i `const` o com la *destructuració* d'objectes, no queda altra opció que reescriure el codi de manera que, amb les eines disponibles, faci el mateix (o, cas que no sigui posible, presenti la funcionalitat més semblant possible) que hauria fet l'original en un intèrpret que la suportàs.

En aquest cas no podem simplement carregar una llibreria, sinó que és necessari transformar el codi original en una versió modificada que sigui compatible.

Les eines que ens permeten fer aquesta transformació de forma automatitzada es diuen *transpiladors*.

El més conegut és el *Babel*, que teniu també enllaçat a les referències.

Referències

- Understanding Javascript OOP: https://robotlolita.me/2011/10/09/understanding-javascript-oop.html.
- Top 10 ES6 Features Every Busy JavaScript Developer Must Know: https://webapplog.com/es6/.
- Modernizr: https://modernizr.com/.
- Babel: https://babeljs.io/.
- Altres:
  - Motor de JavaScript V8 de Google: https://ca.wikipedia.org/wiki/V8_(int%C3%A8rpret_JavaScript).

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).

 A les *Referències* trobareu l'enllaç al portal web del motor de plantilles *Pug* on s'explica perfectament la seva senzilla sintaxi.

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

Si l'acabam d'instal·lar, no trobarem aquesta opció. Per activar-la:

- Teclejarem `Ctrl+Shift+P`.
- Ens sortirà un camp de text al que escriurem "install package control" i pitjarem `Enter`

- 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:

Prerequisits

- *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 *Expres* 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. Executam `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 sho
added 118 packages from 174 contributors and audited 247 pa
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


> curshtml5@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 nostre 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).

Plantilles *Pug*

Ja tenim un servidor web amb el que podem penjar tota mena de fitxers estàtics. Ja siguin html, css, imatges, javascript...

Però fer feina amb fitxers html directament resulta molt dificultós i propens a errors.

Exprés ens permet fer servir motors de plantilles, com el *Pug*, que no només ens faciliten molt la feina, sinó que també ens eviten moltes hores cercant errors estúpids com tancar un tag malament (o simplement no fer-ho) ja que sabem que el HTML que ens generarà serà sempre, com a mínim estructuralment, correcte.

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}
```

Si be la sintaxi és prou autoexplicativa, a l'apartat de *Referències* hi trobareu també l'enllaç a la plana web de *Pug* on podreu aprendre com li podeu treure més suc.

En aquest cas però, aquestes plantilles no es publiquen automàticament, sinó que estan pensades per a ser utilitzades des d'un programa més complex.

El responsable de que en accedir a http://localhost:3000 vegem la plantilla *index.pug* renderitzada és la següent *ruta* que podem trobar al fitxer *routes/index.js*:

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

Com que l'objectiu del curs no és aprendre *Express*, simplement afegirem aquesta nova ruta al mateix fitxer:

```
/* Get any (root-level) pug template. */
router.get('/:baseName.html', function(req, res, next) {
  const bName = req.params.baseName;
  var model = {};
  try {
    model = require("../models/"+bName + ".js");
  } catch (err) {
    console.warn("Error processant el model: models/" + bNa
  };
  res.render(bName, model);
});
```

Amb ella, i un cop arrancat de nou el servidor amb la comanda `npm start`, qualsevol fitxer *Pug* que desem al directori *views*, estarà disponible sota la ruta del mateix nom però extensió `.html`.

Exemple: `views/cursHTML.pug` -> `http://localhost:3000/cursHTML.html`.


Aquesta ruta, a més, intentarà trobar un fitxer, també del mateix nom, però amb extensió *.js* i sota el directori *models* que carregarà com a *model* de dades per a la nostra plantilla.

Així, si cream aquest directori (`mkdir models`) i, seguint amb l'exemple anterior, hi desam un fitxer `cursHTML.js` amb un contingut semblant al següent, hi podrem accedir des de la nostra plantilla evitant-nos així haver de mesclar aquesta amb les dades a mostrar:

```
module.exports = {
  title: "Curs HTML 5",
  someTable: [
    {
      name: "Manel",
      birthdate: "21/04/1983",
      address: "A ca seva",
    },
    ...
  ],
}
```

Referències

- *Sublime Text*: https://www.sublimetext.com/.
  - Instruccions Ubuntu: http://tipsonubuntu.com/2017/05/30/install-sublime-text-3-ubuntu-16-04-official-way/.
- 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/.
- Motor de plantilles Pug: https://pugjs.org.