ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Framework Javascript

Terza parte

Fabio Vitali

Corsi di laurea in Informatica e
Informatica per il Management
Alma Mater – Università di Bologna

Oggi parleremo di…

Alcuni framework:

- *JQuery*
- *Node + Express*

- Il problema dei template
- Moustache e Handlebar
- AngularJS + Angular
- React
- Vue
- Web Components

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

I template HTML

I template HTML

- I siti web dinamici creano le loro pagine HTML come risultato di un'operazione computazionale.
- Nei siti web LAMP più evoluti, l'HTML viene generato come ultima operazione dopo la identificazione di tutti i valori variabili risultato della logica applicativa (Architettura a quattro livelli).
- Spesso si utilizza per questo un template, ovvero una pagina priva di contenuti ma completa dal punto di vista grafico e tipografico, in cui vengono inseriti appositi elementi, detti placeholder, che verranno sostituiti con un valore calcolato dall'applicazione.
- I placeholder hanno una sintassi speciale, per distinguerli dal contenuto normale, e delle regole di composizione e arricchimento funzionale.

I template HTML: i problemi

Ricorrere ad un linguaggio di programmazione per generare codice HTML però crea alcuni stupidi problemi sintattici solitamente ignorati dai linguaggi di programmazione tradizionali:

- Stringhe molto lunghe e con ritorni a capo interni
- Sovrapposizioni tra l'uso di alcuni caratteri (ad esempio, le virgolette) sia nel linguaggio di programmazione (delimitatori di stringa) sia nel linguaggio destinazione (delimitatori di valori di attributi)
- Facilità di interpolazione (cioè di embedding di variabili della computazione in posizioni speciali del template)

- Non sono problemi "veri", ovvero non sono impedite queste funzioni da nessun linguaggio di programmazione.
- E' un problema di comodità e convenienza: alcuni linguaggi rendono queste cose più facili, veloci e meno prone ad errori.

Interpolazione di variabili
soluzione 0: virgolette annidate

- Le virgolette sono caratteri ammessi in HTML, e sono caratteri fondamentali nel markup (ad esempio, attributi).
- HTML è esso stesso una stringa, che può essere prodotto programmaticamente. Le virgolette sono delimitatori standard delle stringhe.

```
var a ="<p class='"+obj.class+"'>"+obj.testo+"</p>"
```

- Se ho variabili da inserire nell'HTML, questo può portare a notevoli problemi di annidamento di stringhe in altre stringhe:

```
var b="<span onclick='doX(\""+obj.par+"\")'>"+obj.testo+"</p>"
```

Interpolazione di variabili

soluzione 1: funzioni di formattazione

Molti linguaggi di programmazione hanno funzioni e operatori per fare formattazione controllata di valori in una stringa sulla base dell'ordine in cui i valori vengono specificati:

- C/C++ : `sprintf(v, "%d + %d = %d\n", a, b, a+b);`
- Python: `v = "%d + %d = %d" % (a, b, a+b)` oppure
  `v= "{0} + {1} = {2}".format(a, b, a+b)`

A volte ci sono anche meccanismi per formattazione di valori sulla base dei loro nomi:

- Python: `v="{a} + {b} = {c}".format(a=a, b=b, c=a+b)`
- PHP:
```
$c= $a+$b;
$v="$a + $b = $c";
```

In PHP l'interpolazione delle variabili è automatica nelle stringhe delimitate con doppi apici.

Interpolazione di variabili
soluzione 2: embedding di istruzioni

PHP nasce come motore di template (PHP significa Hypertext Preprocessor). Il meccanismo (poi esportato in altri linguaggi) di permettere embed di comandi eseguibili dentro ad una pagina HTML è un chiaro esempio di template.

```
<html>
   <head>
      <title><? echo $page_title;?></title>
   </head>
   <body>
      <h1><? echo $page_title;?></h1>
      <? echo $page_content; ?>
   </body>
</html>
```

Interpolazione di variabili
soluzione 3: heredoc

- La sostituzione delle variabili però non è sufficiente, perché nella maggior parte dei casi un documento HTML è molto lungo e diventa scomodissimo forzarlo su una riga soltanto.
- Gli here document (*heredoc*) sono sintassi (dipendenti da linguaggio a linguaggio) per avere valori stringa lunghi più di una riga nel programma.

Python

```
V = """\
<html>
<body>
  <h1>Hello {name}!</h1>
</body>
</html>
""".format(name="John")
```

PHP

```
$V = <<<EOS
<html>
<body>
  <h1>Hello $name!</h1>
</body>
</html>
EOS
```

Interpolazione di variabili
soluzione 4: template server-side

- I template server-side servono per generare un documento pronto a partire da un ambiente di computazione e un documento HTML vuoto con variabili da interpolare.
- Esistono anche per Javascript/Node, il più famoso è Pug (in precedenza chiamato Jade), che ha una sintassi semplificata per la parte HTML:

HTML (sort of...)

```
doctype html
html(lang='en')
 head
  title #{title}
 body
   h1 #{title}
   div.container
    p #{message}
```

Javascript – Express.js

```
app.set('view engine', 'pug')
app.get('/', function (req, res){
  res.render('index.pug', {
    title: 'Ciao a tutti',
    message: Prima prova!'
  })
})
```

Interpolazione di variabili

Ok. Ma client-side?

Javascript non aveva né un meccanismo di interpolazione di variabili, né una sintassi per stringhe lunghe (*heredoc*).

Posso usare una stringa normale con i soliti problemi di lunghezza della riga e mancanza di interpolazione.

```
var str1  = "<p>prima riga di una stringa lunga,\n ";
    str1 += "seguita da una seconda riga,\n ";
    str1 += "e poi una terza riga.</p>";
```

Oppure posso bloccare la chiusura di una stringa con un backslash e proseguire sulla riga successiva:

```
var str1  = "<p>prima riga di una stringa lunga,\n \
             seguita da una seconda riga,\n \
             e poi una terza riga.</p>";
```

Interpolazione di variabili
soluzione 5: nascondere i template

- ***Idea***: nascondere il testo del template nell'HTML stesso, ma inerte e inattivo, e poi quando serve prenderlo a inserirlo nel DOM in una parte visibile.
- Tuttavia debbo ricordarmi di proteggere il template HTML dalla sua visualizzazione, esecuzione, e/o indicizzazione prima del tempo o nel posto sbagliato.
- Per quanto riguarda il frammento, nel tempo si sono suggeriti vari meccanismi di protezione:
  - *div nascosti via CSS o posizionati fuori schermo.*
    - L'utente normale non lo vede, ma se il CSS viene disattivato lo vede. Inoltre un motore di ricerca lo indicizzerà lo stesso. Inoltre le immagini verranno comunque caricate e gli script eseguiti.
  - *<script type="text/template"> ... </script>*
    - Un truccaccio, ma funziona.

Interpolazione di variabili

soluzione 6: il tag `<template>`

Nell'HTML living standard è previsto un nuovo tag <template>.

Il contenuto di <template> è inerte e non fa parte del documento:

- Le immagini non vengono caricate,
- gli script non vengono eseguiti,
- Il contenuto non occupa spazio nel layout, ecc.

Al momento utile, il template viene attivato importando il suo contenuto nel DOM come discendenti di un nodo del DOM esistente. In quel momento i contenuti si visualizzano, le immagini si caricano, gli script vengono eseguiti.

```
var t = $('#tpl').content
$('img', t).src='fig1.gif'
var c= document.importNode(t)
$('#out').appendChild(c)
```

```
<template id="tpl">
   <p>Testo</p>
   <img src=""/>
</template>
```

Interpolazione di variabili
ok, ma la interpolazione?

- Adesso abbiamo i template, ma come facciamo per la sostituzione di variabili dentro ad un testo?
- Javascript ha la funzione replace(), ma è complicata e sostituisce una cosa alla volta:

```
"$1+$2=$3".replace("$1",a).replace("$2",b).replace("$3",a+b)
```

- JQuery non ha niente di accettabile.

C'è qualcosa di meglio?

Interpolazione di variabili
soluzione 7: Template literals (ES 2015)

Una nuova sintassi per definire stringhe multi-linea con interpolazione di variabili.

```
var firstName = 'Jane';
var x = `Hello ${firstName}!
How are you
today?`;
```

```
x vale "Hello Jane!
How are you
today?"
```

Tre elementi fondamentali: :

- Backticks come delimitatori: `
- I new line fanno parte della stringa
- Interpolatori: `${varName}`

Attenzione! L'interpolazione avviene solo durante l'assegnazione del template literal, e non può essere controllata né posticipata.

Interpolazione di variabili

soluzione 8: interpolation dei poveri

```
String.prototype.tpl = function(o) {
   var r = this ;
   for (var i in o) {
      r = r.replace(new RegExp("\\$"+i, 'g'),o[i])
   }
   return r
}
```

Per usarla con un array:

```
var t = "$0+$1=$2";
var a = [a, b, a+b];
var r = t.tpl(a)
```

Per usarla con un object:

```
var t = "<p>Io sono il $tit $nome
   $cognome, ma potete chiamarmi
   $nome.</p>";
var o = {
   tit: "prof.",
   nome: "Fabio",
   cognome: "Vitali"
} ;
var r = t.tpl(o);
```

Interpolazione di variabili

soluzione 9: tutto insieme

Se uso <template>, e String.prototype.tpl(), posso ottenere risultati piacevoli in modo molto comodo:

```
<template id="tpl1">
  <p class="$c">$t</p>
</template>
<template id="tpl2">
  <span onclick='doX("$parameter")'>$testo</p>
</template>
...
$('#out').append($('#tpl1').html().tpl({c:"c1", t:"Prova"}));
$('#out').append($('#tpl2').html().tpl({
        parameter:"pippo.html",
        testo:"clicca qui"
}));
```

Interpolazione di variabili
soluzione 10: template client-side

In alternativa, posso ricorrere a framework appositi.

Negli ultimi anni ne sono nati tre di grande successo e con una ragionevole somiglianza tra loro:

- Angular (poi evoluto in una vera e propria piattaforma applicativa con poca attinenza con il templating)
- ReactJS
- VueJS

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

I primi linguaggi di template per Javascript

Mustache.js

- Mustache è una semplice sintassi per template testuali con interpolazione di variabili
  - *logic-less*: nessun flusso di controllo, nessuna istruzione per cicli o condizioni;
  - *Separation of logic and presentation*: nessun modo per fare embedding di logica applicativa nel template.
- Mustache è disponibile su decine di linguaggi di programmazione; Mustache.js è la versione per Javascript.
- L'unico modo per fare cicli è utilizzare dei blocchi speciali.

```
Ciao {{name}} {{surname}}!
Hai vinto {{value}} euro!
{{#taxed}}
  Beh, {{net}} euro, con le tasse.
{{/taxed}}
```

```
{
  "name": "Fabio",
  "surname": "Vitali",
  "value": 10000,
  "net": 10000*0.78,
  "taxed": true
}
```

```
Ciao Fabio Vitali!
Hai vinto 10000 euro!
Beh, 7800 euro, con le tasse.
```

Handlebar.js

Handlebar.js è una semplice estensione di Mustache.js che permette di accedere agli oggetti annidati dentro alle variabili da interpolare e di ottenere l'effetto di cicli e istruzioni condizionali usando bocchi contestuali.

```
<div>
  <h1>{{title}}</h1>
  <h2>by
   {{#each authors}}
    <b>{{#if title}}{{title}}{{/if}}
       {{name}} {{surname}}
    </b>{{#unless @last}},{{/unless}}
   {{/each}}
  </h2>
  <div class="body">{{body}}</div>
</div>
```

```
{
  title: "My new post",
  authors: [{
    name: "John",
    surname: "Smith",
    title: "Dr."
  },{
    name: "Mandy",
    surname: "Brown"
  }],
  body: "This is my first post!"
}
```

My new post

by Dr. John Smith, Mandy Brown

This is my first post!

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Framework a componenti

Angular.js

Angular

React

Vue

Web Components

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Angular.js e Angular

Angular.js

- Nasce nel 2009 come progetto privato di un dipendente Google e un suo amico. Diventa un progetto Open Source nel 2010, ufficialmente supportato da Google. Il team di sviluppo era composto da tre persone.
- E' un framework per la semplificazione della realizzazione di Single Page Application, basato sui pattern Model-View-Controller (MVC) e Mode-View-ViewModel (MVVM).
- Nel 2015 evolve al di là e al di fuori del motore di template e diventa una vera e propria piattaforma applicativa.
- Ogni parte di una pagina web può diventare la *view* di un *model*, e via Javascript si crea il *controller* che ne gestisce bi-direzionalmente la creazione e lo sviluppo.

Angular.js: model-view-controller

- Il *model* è una struttura dati (tipicamente espressa in JSON o come oggetto Javascript) che può esistere nella pagina web, essere caricata dinamicamente via Ajax, o essere calcolata a seguito di computazioni client-side.
- La *view* è un frammento HTML nella pagina principale, che viene decorato con speciali attributi (che iniziano con ng-) e placeholder (con la sintassi {{nome}}).
- Il *controller* è il codice Javascript che lega la view al model.
- Il *module* è il contenitore di tutte le parti che costituiscono un'applicazione AngularJS.
- La *directive* è un struttura di markup (elemento, attributo, classe) a cui associare un comportamento specifico.

Angular.js: Bi-directional binding e dependency injection

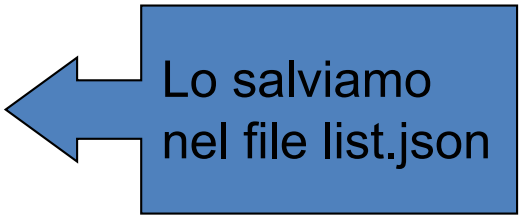
Angular.js introduce due concetti importanti che troveremo altrove:

- *bi-directional binding*
  - *legame mono-direzionale*: se un elemento del model è associato ad uno o più elemento HTML allora cambiando il valore nel modello cambia automaticamente anche la visualizzazione dell'HTML
  - *legame bi-direzionale*: se l'elemento HTML è modificabile (ad es., è un input di qualche tipo), allora cambiando il value dell'elemento si cambia anche il valore relativo nel modello, e viceversa.
- Dependency injection
  - Per evitare effetti collaterali imprevisti, ogni modulo lavora in un ambiente isolato senza accesso all'ambiente esterno, es. variabili globali.
  - Per fornire accesso a questo o quell'oggetto esterno, si deve passarglielo esplicitamente tra i parametri del controller (*iniezione degli oggetti utilizzati*, o *dipendenze*).

Angular.js: il model

- Il *model* è una struttura dati (tipicamente espressa in JSON o come oggetto Javascript) che può esistere nella pagina web, essere caricata dinamicamente via Ajax, o essere calcolata a seguito di computazioni client-side.
- Ad esempio:

```
{
  "users" : [{
    "name": "Fabio Vitali",
    "tel": "051 2094872",
   "studenti": 120
  },{
    "name": "Angelo Di Iorio",
    "tel": "051 2094889",
   "studenti": 60
  }]
}
```



Angular.js: la view

- La *view* è un frammento HTML nella pagina principale, che viene decorato con speciali attributi (che iniziano con ng-) e placeholder (con la sintassi {{nome}}).
- Nella view viene indicato il controller da utilizzare nel frammento:

```
<html>
  <body ng-app="simpleApp">
    <h1>Un semplice test Angular</h1>
    <table border="1" ng-controller="listCtrl">
      <tr>
       <th>Nome</th><th>Tel</th><th>studenti</th>
      </tr>
      <tr ng-repeat="person in list">
        <td>{{person.name}}</td>
        <td>{{person.tel}}</td>
        <td align='right'>{{person.studenti}}</td>
      </tr>
    </table>
  </body>
</html>
```

Angular.js: il controller

- Il module è il contenitore di tutte le parti che costituiscono un'applicazione AngularJS. Il controller lega la view al model.
- Il controller esplicitamente fa riferimento agli oggetti che vuole vedere. Questa si chiama *dependency injection*.
- In questo caso facciamo riferimento ai seguenti oggetti:
  - $scope: lo scope di visibilità della view
  - $http: l'oggetto che si occupa di comunicazioni via Ajax.

```
var simple = angular.module('simpleApp', []);
simple.controller('listCtrl', function($scope, $http){
     $http.get('list.json').success(function(data) {
        $scope.list = data.users;
     });
  });
```

Angular.js: bi-directional binding

Ovvero legame bidirezionale: se un elemento HTML è modificabile (es., un input di qualche tipo), allora cambiando il value dell'elemento si cambia anche il campo relativo nell'oggetto scope, e viceversa.

```
<tr ng-repeat="person in list">
  <td>{{person.name}}</td>
  <td>{{person.tel}}</td>
  <td align="right">{{person.studenti}}</td>
  <td><input type="number" ng-model="person.studenti"/>
    <button ng-click="add(person,10)">+10</button>
    <button ng-click="add(person,-10)">-10</button></td>
</tr>
...
var simple = angular.module('simpleApp', []);
simple.controller('simpleAppCtrl',function($scope, $http){
  $http.get('list.json').success(function(data) {
    $scope.list = data.users;
    $scope.add = function(person,n){person.studenti+=n};
  });
});
```

Angular.js: dependency injection

Per evitare effetti collaterali imprevisti, ogni controller lavora in un ambiente isolato privo di accesso all'ambiente esterno, in particolare le variabili globali.

Per fornire accesso a questo o quell'oggetto esterno, è necessario passarglielo esplicitamente tra i parametri della funzione del controller (*iniezione degli oggetti utilizzati*, o *dipendenze*).

Questo isola in un punto solo l'identificazione delle dipendenze dell'algoritmo, e permette di realizzare test sofisticati modificando il tipo e i permessi degli oggetti iniettati.

```
simple.controller('simpleCtrl',function($scope, $http){
   $http.get('list.json').success(function(data) {
      $scope.list = data.users;
      $scope.add=function(person,n){person.studenti+=n}
   });
});
```

Angular.js: le direttive

- Una direttiva è un nuova struttura di markup (elemento, attributo, classe) a cui associare un comportamento specifico.
- E' un modo per estendere HTML con nuovi tag e attributi specifici all'applicazione.

```
<tr ng-repeat="person in list">
   <td>{{person.name}}</td>
   <td>{{person.tel}}</td>
   <td align="right">{{person.studenti}}</td>
   <td><aggiungi></aggiungi></td>
</tr>
...
simple.directive('aggiungi', function() {
  return {
    restrict: 'E',
    template: '<input type="number" ng-model="person.studenti"/>\
               <button ng-click="add(person,10)">+10</button>\
               <button ng-click="add(person,-10)">-10</button>'
    }
})
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Angular

Angular.js e Angular

- Alla fine del 2014, gli sviluppatori di Angular fecero un annuncio drammatico:

  La nuova release di AngularJS sarebbe stata drasticamente differente, non compatibile all'indietro e senza un percorso di evoluzione dalla versione precedente.

- Per sottolineare il cambiamento, si richiese che d'ora in poi si intendesse con AngularJS la versione 1.x fin lì utilizzata, e con Angular e basta le versioni successive, senza fare riferimento alla versione utilizzata (siamo ora alla versione 9).
- Angular non è più un sofisticato motore di template HTML basato su Model, View e Controller, ma un ambiente di produzione di applicazioni web basato sul concetto modulare di componente che include ***anche*** dei template HTML per la loro presentazione a schermo.

I principali concetti di Angular

Angular evolve da AngularJS rinunciando ad alcuni dei suoi concetti più importanti e rivoluzionandone altri:

- Di passa da MVVM a CBA (Component Based Architecture)
- AngularJS prende una pagina HTML già organizzata e la arricchisce con contenuti dinamicamente generati da Javascript su direttive e template; Angular costruisce da zero il DOM finale a partire da frammenti HTML e CSS indipendenti tra loro, inseriti ed integrati in appositi moduli JS/Typescript.
- La sintassi AngularJS usa il prefisso ng- per ogni direttiva nel codice HTML. Angular usa decoratori sintattici ((click) invece che ng-click, e però *ngFor invece che ng-repeat).
- Si abbandonano i controller e gli scope espliciti, sostituiti dall'incapsulamento dato dai componenti.
- Si adotta il linguaggio Typescript, che viene compilato in Javascript a runtime (non è obbligato, ma fortemente suggerito).

Angular setup e bootstrap

- AngularJS ha un documento HTML che contiene le direttive fondamentali per l'attivazione della applicazione, ng-app e ng-controller.
- Angular invece ha una Command Line Interface (CLI) attraverso cui creo un progetto, che automaticamente genera una struttura di directory e di file predefiniti, di cui andare a modificare solo quelli rilevanti.
- Ogni componente viene posto in una directory indipendente che contiene un frammento HTML, una dichiarazione di classe in Typescript, e uno o più file CSS associati, ottenendo una totale autonomia ed incapsulamento del codice del componente

Concetti di Angular

- *Moduli*: ***NON*** quelli di Javascript, ma frammenti complessi composti da HTML, CSS e codice (JS o TS) pensati per essere incapsulati, autonomi e autosufficienti (il flusso di dati da modulo a modulo è rigidamente controllato e limitato).
- *Direttive*: componenti speciali che cambiano il comportamento del markup e il DOM che viene generato. Introdotte dal decoratore @Directive.
  - Direttive strutturali: cambiano il contenuto del DOM: ad es. *ngFor,*ngIf
  - Direttive di attributo: arricchiscono il DOM con comportamenti: ad es. [ngModel].
- *Componenti*: un tipo di direttiva a cui corrisponde un template HTML e un selettore (i.e. un nuovo elemento di markup) che andrà a sostituire nel DOM finale. Introdotto dal decoratore @Component.
- *Routing*: l'applicazione può passare da uno macro stato all'altro (ad esempio elenco articoli, vista articolo, vista carrello, ecc.) attraverso un meccanismo di routing interno che modifica anche l'URI visibile e gestisce scoping e passaggio dati da parte a parte dell'applicazione

Angular: un esempio

Creiamo un nuovo componente

- da CLI:

```
> ng generate component hello-world
```

- la CLI crea una cartella pre-popolata:

  `src/app/hello-world`

  - `hello-world.component.ts` -> codice Typescript del component
  - `hello-world.component.css` -> codice CSS del component
  - `hello-world.component.html` -> template HTML del component
  - `hello-world.component.spec.ts` -> codice di test del component

- I template degli altri componenti possono usare il nuovo tag:

```
<app-hello-world> ... </app-hello-world>
```

- I template possono contenere direttive Angular e interpolazione di dati passati dall'applicazione.

```
<p>Hello {{name}}! </p>
```

Un esempio

app-component.html

```
<h1>
  <app-hello-world>
  </app-hello-world>
</h1>
```

hello-world.component.ts

```
@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})

export class HelloWorldComponent
         implements OnInit {

  name:string;

  constructor() {
     this.name="Fabio"
  }

  ngOnInit() { }
}
```

hello-world.component.html

```
<p>
    Hello {{name}}!
</p>
```

Angular.js vs. Angular

- Angular ha una parabola interessante: nata come side project di programmatori di Google, ha avuto un grande successo di immagine per qualche anno, poi offuscata dalla nascita di React e Vue.
- ***Violentemente opinionato***, Angular richiede un po' di tempo per abituarcisi, e forma e vincola grandemente il progettista al modello applicativo implicito nella libreria.
- ***Dependency injection*** e ***unit testing*** rappresentano le basi più astratte su cui è stata fondata una libreria di template, che si è evoluta rapidamente.
- Gli stessi sviluppatori hanno ritenuto questi due concetti molto più importanti del ***templating*** e del ***Model-View-Controller***, tanto da lasciarli indietro nelle nuove versioni e basarle su concetti completamente diversi.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

React.js

React.js

- Nasce nel 2011 all'interno di Facebook, e ha la prima installazione importante con Instagram nel 2012. Diventa open source nel 2013.
- Inizialmente solo un motore di template, poi un application framework per i browser, infine con React Native (2015) un ambient di progettazione integrato che comprende anche gli ambienti mobili.
- React Native continua ad usare Javascript come motore fondamentale, ma non usa HTML o il DOM come destinazione dei comandi, ma view native del sistema operativo di destinazione.

Principi base di REACT (1)

Virtual DOM

- L'applicazione REACT non cambia il DOM, ma cambia una copia del DOM sotto il controllo della libreria. Questa controlla propagazioni e effetti collaterali e aggiorna quando serve il DOM vero modificando ciò che vede l'utente *(reconciliation).*

One-way data binding

- Ogni elemento del DOM è associato ad uno store, e quindi ogni modifica viene effettuata allo store. Lo store chiederà al VDOM di rieseguire il rendering dell'elemento associato, e da lì propagata al DOM.

Universal rendering

- L'uso del VDOM implica che la mia applicazione non debba per forza interagire con elementi HTML di una pagina web, né che debbo per forza risiedere sul browser stesso.
- Attraverso renderer diversi (e.g., React Native) posso ottenere la stessa applicazione funzionante nativamente su device mobili. Attraverso Next.js posso creare Server-Side Rendering (SSR) precompilati per ottimizzare SEO e velocità dell'applicazione.

Principi base di REACT

JSX

- Un mix di javascript, XML e CSS per semplificare la realizzazione di template HTML che includono computazioni e calcolo di espressioni Javascript / Typescript.
- In particolare la coesistenza di HTML e JS permette di semplificare la coesistenza dei separatori di stringhe (virgolette) e nuovo markup
- Babel è il componente di React che controlla la coesistenza.

Components

- Un'unità indipendente di codice, markup e stile, richiamabile con markup inventato e riusabile e configurabile via *props*.

React.js: modello di processo (1)

- Un component in React è una funzione che restituisce codice HTML o JSX.
- La funzione render() contiene il template da utilizzare per la sua visualizzazione.
- Per semplificare la vita del programmatore, il programma è scritto secondo una sintassi speciale, JSX, che mescola comandi Javascript e markup HTML

```
class Hello extends React.Component {
  render() {
    return <h1>Hello, {this.props.name || "world"}!</h1>;
  }
}
```

- Il componente React a questo punto permette una struttura di markup riutilizzabile basato su nuovi tag:

```
<body>
  <Hello name="Fabio"> </Hello>
  <p>Come va? Tutto bene?</p>
</body>
```

React.js: modello di processo (2)

- JSX è un formato XML che mescola in maniera sensata codice Javascript, frammenti HTML e elementi XML inventati ad hoc.

```
function Hello(props) {
  return <h1>Hello, {props.name || "world"}</h1>;
}

function Many() {
  return (
    <div>
      <Welcome name="Fabio" />
      <Welcome name="Alice" />
      <Welcome />
    </div>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Many />);
```

React.js: JSX e Babel

- JSX è un'estensione di Javascript che ammette al suo interno anche frammenti non protetti di codice di markup.
- L'interprete Babel esamina sistematicamente e ricorsivamente il codice JSX.
  - Ogni espressione JavaScript viene valutata ed eseguita.
  - Per ogni elemento per cui esiste una funzione con lo stesso nome, viene eseguita la funzione e l'output sostituisce l'elemento.
  - Per ogni elemento per cui esiste una classe con lo stesso nome, viene eseguita la funzione render() e l'output sostituisce l'elemento.
- Appena non ci sono più sostituzioni da fare, l'interpretazione è conclusa. Il metodo ReactDOM.render() inserisce nel nodo specificato il codice JSX interpretato.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Vue.js

Vue.js

- Nato nel 2014 all'interno di Google come costola semplificata di Angular:
  - "I figured, what if I could just extract the part that I really liked about Angular and build something really lightweight without all the extra concepts involved?" (Evan You)
- Possiede template, sintassi handlebar, espressioni Javascript arbitrarie, binding bidirezionale, direttive (chiamate *componenti*), una struttura molto semplice di applicazione.
- Non possiede: scope, dependency injection, controller.

Vue.js: un semplice esempio

```
<body>
   <div id="app">
      <button @click="count++">
         Hai cliccato {{count}} volte.
      </button>
   </div>

   <script type = "text/javascript">
      import { createApp } from 'vue'
      createApp({
         data() {
            return { count: 0 }
         }
      }).mount('#app')
   </script>
</body>
```

- ***Interpolazione***: Una struttura unica mantiene tutti i dati visibili all'applicazione in un'unica struttura dati. Ogni formula di interpolazione nel documento accede a questa struttura dati in maniera diretta.
- ***Reattività***: ogni interazione dell'utente permette di attivare una funzione di callback che interagiace direttamente con i dati e aggiorna il DOM subito.
- ***Integrabilità*** in HTML tradizionale.

Vue.js: Single File Component

Composition API

```
<script setup>
const count = 0

function increment() {
  count++
}
</script>

<template>
  <button @click="increment">
      Hai cliccato {{ count }} volte.
   </button>
</template>
<style scoped>
   button {
      color: red;
   }
</style>
```

Come tutti i framework, anche Vue suggerisce una scomposizione della pagina in componenti isolati ed indipendenti

Essi contengono:

- I template HTML da inserire nel DOM
- Gli stili CSS locali da non condividere
- I dati su cui effettuare le operazioni
- I metodi visibili solo localmente

Ci sono due sintassi principali per la realizzazione di componenti, dette Options API (antica) e Composition API.

Componenti e binding

- Un componente può essere associato ad un elemento di markup.

```
import { createApp } from 'vue'
const app = createApp({})

app.component('modal', {
   props: ['id', 'title'],
   template: `<div class="modal fade" :id="id"
              style="display: none;">
       <div class="modal-dialog">
          <div class="modal-content">
             <div class="modal-header">
                <h4 class="modal-title">{{title}}</h4>
                <button type="button" class="close"
                   data-dismiss="modal">×</button>
             </div>
             <div class="modal-body big">
                <slot></slot>
             </div>
          </div>
       </div>
   </div>`
})
```

```
<modal id="info" title="Informazioni">
   <p>Un po' di informazioni</p>
</modal>
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Web Component

WebComponents

Prendendo ispirazione dai sistemi a componenti come Angular, React e Vue, anche la sintassi standard di HTML ha introdotto i web component.

- *Custom element*: posso estendere il vocabolario degli elementi di HTML con quelli che più mi piacciono.
- *Shadow DOM*: posso creare un mini-DOM protetto a cui non si applicano script, stili e eventi del documento principale
- *HTML template*: posso associare ad un custom element un frammento HTML che va automaticamente a rimpiazzare il custom element di riferimento.

WebComponents

Per esempio:

```
<my-modal id="m1" title="Hello world">Some content</my-modal>
```

può essere usato con un template come:

```
<template id="modalTpl">
<div class="modal" tabindex="-1">
  <div class="modal-header">
    <h5 class="modal-title"><slot name="title"></h5>
    <button type="button" class="btn-close"></button>
  </div>
  <div class="modal-body"><slot></slot></div>
  <div class="modal-footer">
    <button type="button">Close</button>
  </div>
</div>
</template>
<script>
customElements.define('my-modal',
  class extends HTMLElement {
    constructor() {
      super();
      let template = document.getElementById('modalTpl').con
      const shadowRoot = this.attachShadow({mode: 'open'});
      shadowRoot.appendChild(template.cloneNode(true));
    }
  }
);
</script>
```

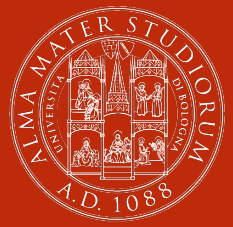
Differenze con altri framework:

- Debbo registrare il componente nell'array customElements.
- Per ora (?) posso subclassare solo alcuni elementi
- Lo shadow DOM è un DOM non visualizzato in cui vado a posizionare cose che non voglio (ancora) visualizzare.
- L'integrazione dello shadow DOM nel DOM vero avviene esplicitamente chiamando la funzione attachShadow()

- La separazione tra pagina e componenti è facoltativa.
- L'uso dei template è facoltativo.
- L'uso dello shadow DOM è facoltativo.
- Si usano <slot> per annidare sotto componenti

Conclusioni: il nuovo web

- Alcuni esempi su `http://www.fabiovitali.it/TW/2023/components/`
- Il sorgente di tutti è disponibile su virtuale
- La rivincita del modello computazionale su quello dichiarativo
- La ***Component Based Architecture*** è qui per rimanere.
- HTML e CSS come l'assembler
- Il ruolo delle aziende su WHATWG e W3C
  - Librerie e linguaggi proprietari: TypeScript, JSX, ecc.
  - CDN e posizionamento fisico delle librerie
  - La conquista delle teste degli sviluppatori



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Fabio Vitali

Dipartimento di Informatica – Scienze e Ingegneria
Alma mater – Università di Bologna

Fabio.vitali@unibo.it

www.unibo.it