

Chi Siamo



Marco Anisetti

Ricercatore

PhD @ Università degli Studi di Milano, Ricercatore presso il Dipartimento di Informatica.

Docente per il corso di Programmazione e Sicurezza dei Sistemi e delle reti Informatiche.

Lavora su sicurezza nella Cloud e tecniche di Computational Intelligence per estrarre e calcolare metriche su sistemi distribuiti.



Filippo Gaudenzi

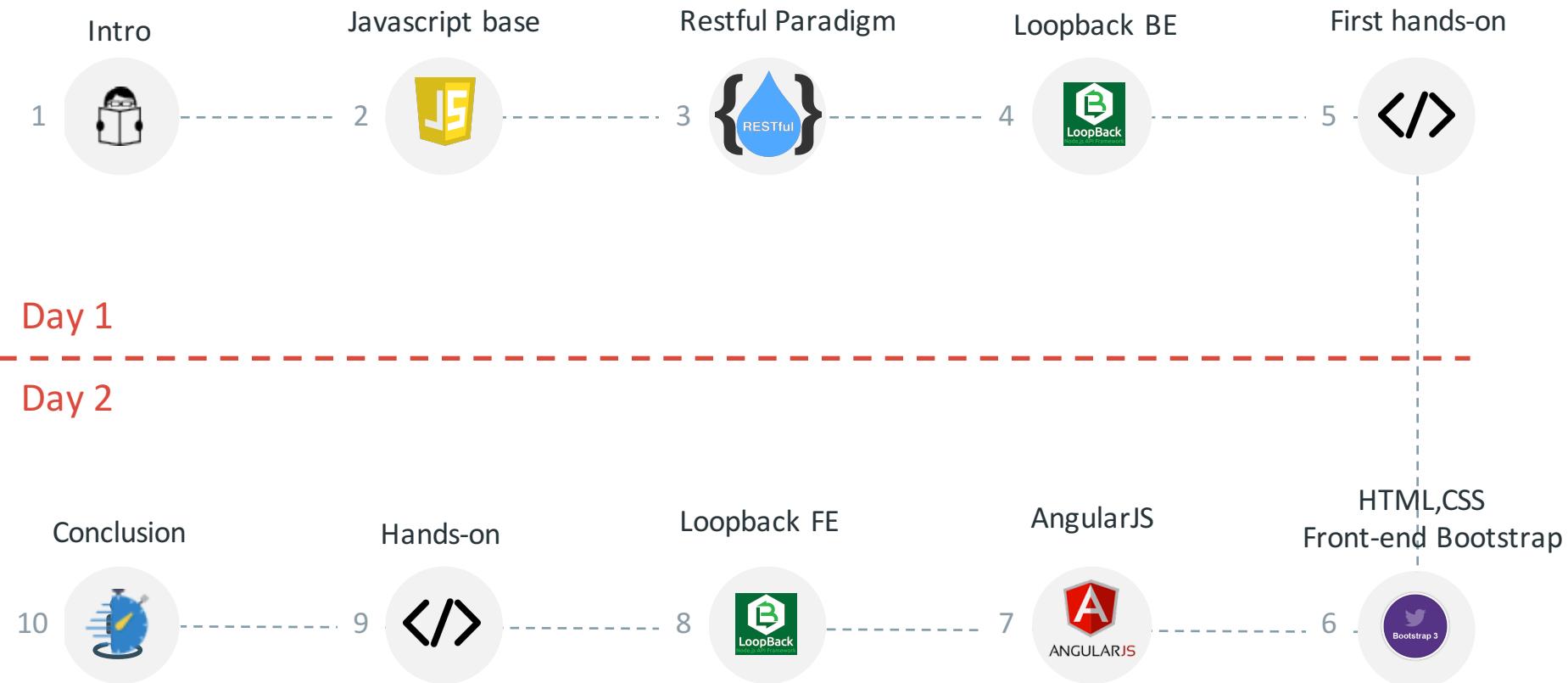
PhD Student

PhD student @ Università degli Studi di Milano, lavora principalmente su sistemi di valutazione per la sicurezza di servizi Cloud.

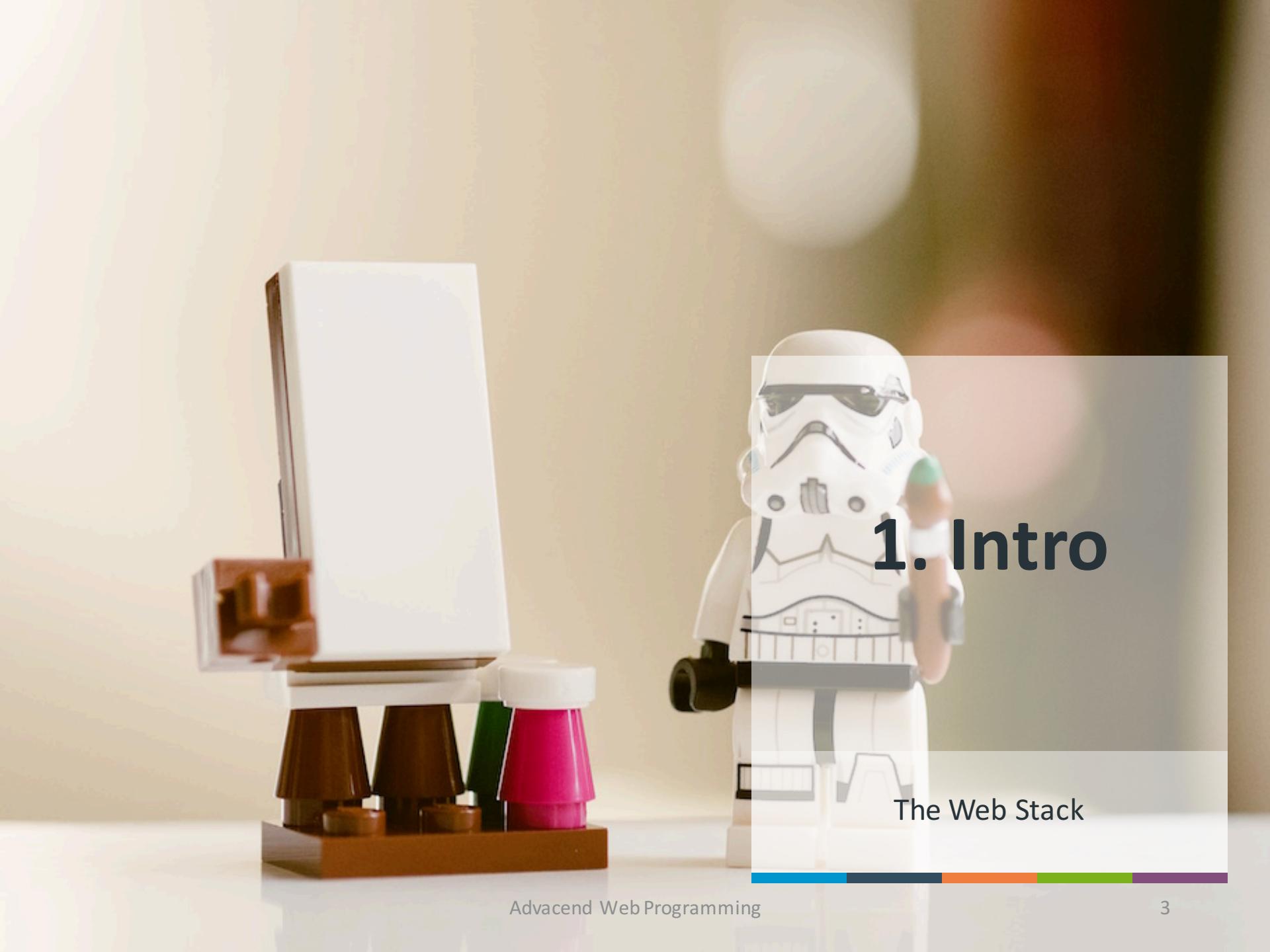
Esperienza 5+ nel campo della programmazione web con Javascript, Java, Python.

*This slide is using Font Awesome

Percorso



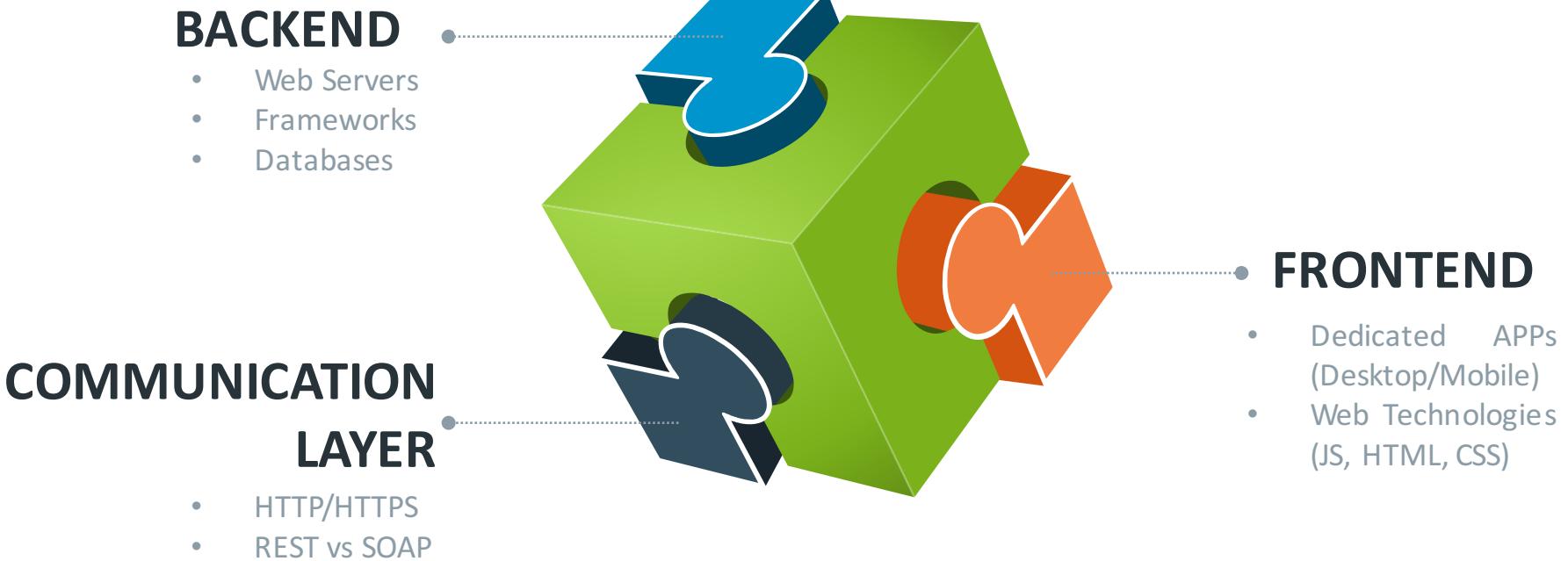
*This slide is using Font Awesome



1. Intro

The Web Stack

FULL STACK WEB



Backend

- Server side
riceve/processa/restituisce
dati al client
- Fornisce una serie di **risorse**
tramite routes (URI)
- Accede alle **base di dati**
- Restituisce **diversi tipi di dato**

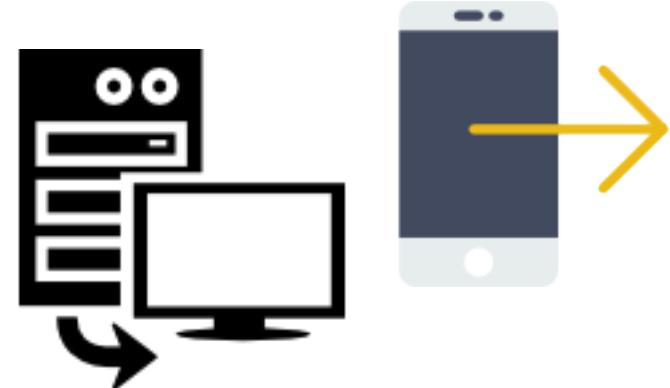


Frontend

Mostra il contenuto fornito dal backend.

Differenza fra:

- Client Web Application
- Server Side Web Application



Server Side Web Application

Classico approccio Web Site. Il server fa il rendering completo della view per il client che la interpreta per mostrarla all'utente.

Client Web Application

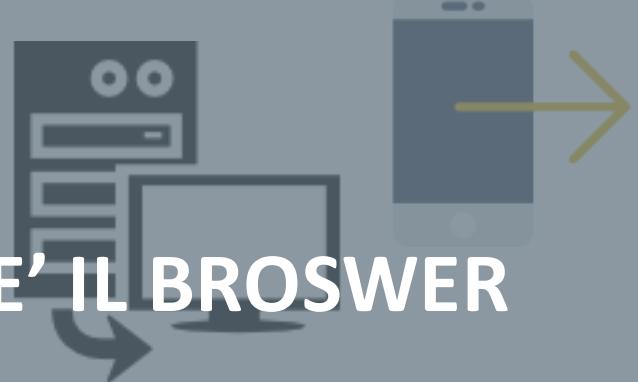
Il server fornisce una interfaccia CRUD e di elaborazione, mentre la logica di presentazione è gestita principalmente lato client

Frontend

Mostra il contenuto fornito dal backend.

Differenza fra:

- Client Web Application
- Server Side Web Application



IL CLIENT PER ECCELLENZA E' IL BROWSER WEB.

Server Side Web Application

Classico applicazione Web Site.
Il server fa il rendering completo
della view per il client che la
interpreta per mostrarla
all'utente.

Client Web Application

Il server fornisce una interfaccia
CRUD e la parte di elaborazione
data avviane lato client

DESKTOP e MOBILE.

Backend Technologies

Frameworks & Programming Languages



JAVASCRIPT	NodeJS, Express, Strongloop/Loopback, Seneca...
PYTHON	Django, Flask, Tornado
PHP	Laravel, Symfony, Zend

Databases

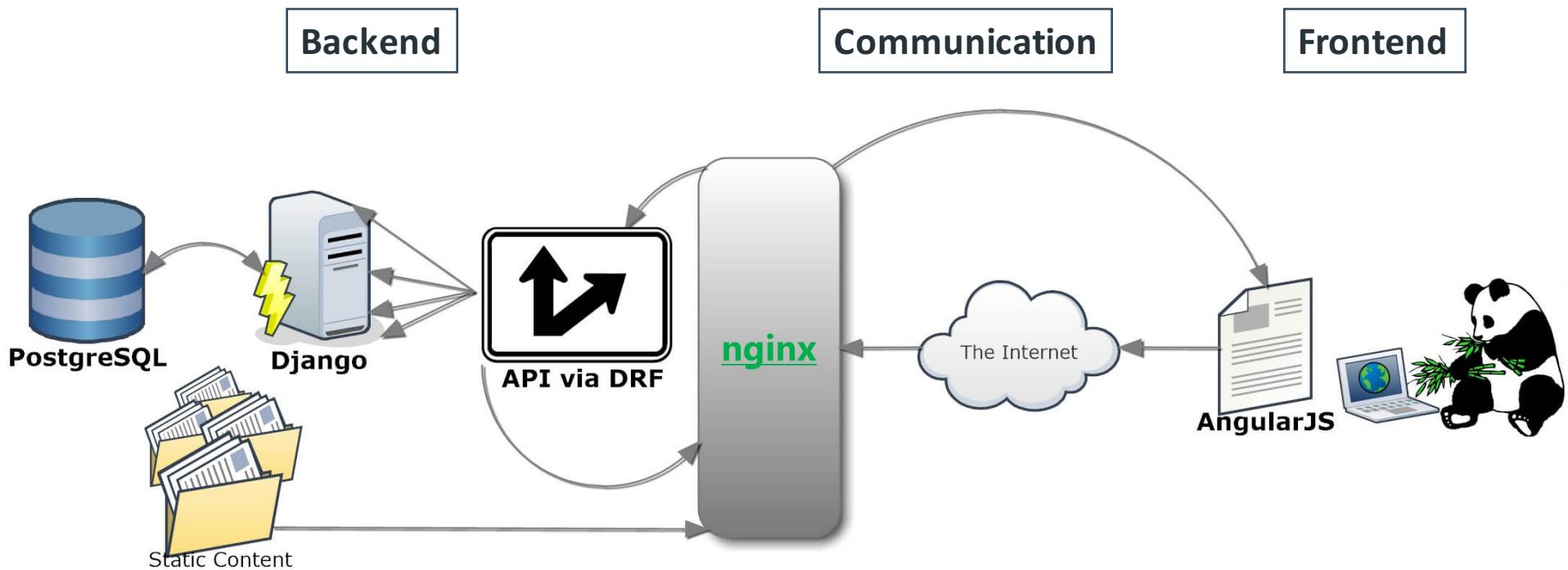


SQL	MariaDB, PostgreSQL, MySQL, OracleDB, SQL Server...
NO-SQL	MongoDB, Cassandra, Redis...

Web Servers

Nginx, Apache (..), IIS, Oracle Web Tier...

THE FLOW



Tecnologie Adottate

Frontend

HTML



CSS

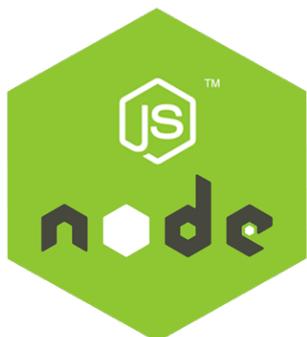


JS



ANGULARJS

Backend



Tecnologie Adottate - Frontend

VIEW



HTML



CSS



MODEL/CONTROLLER

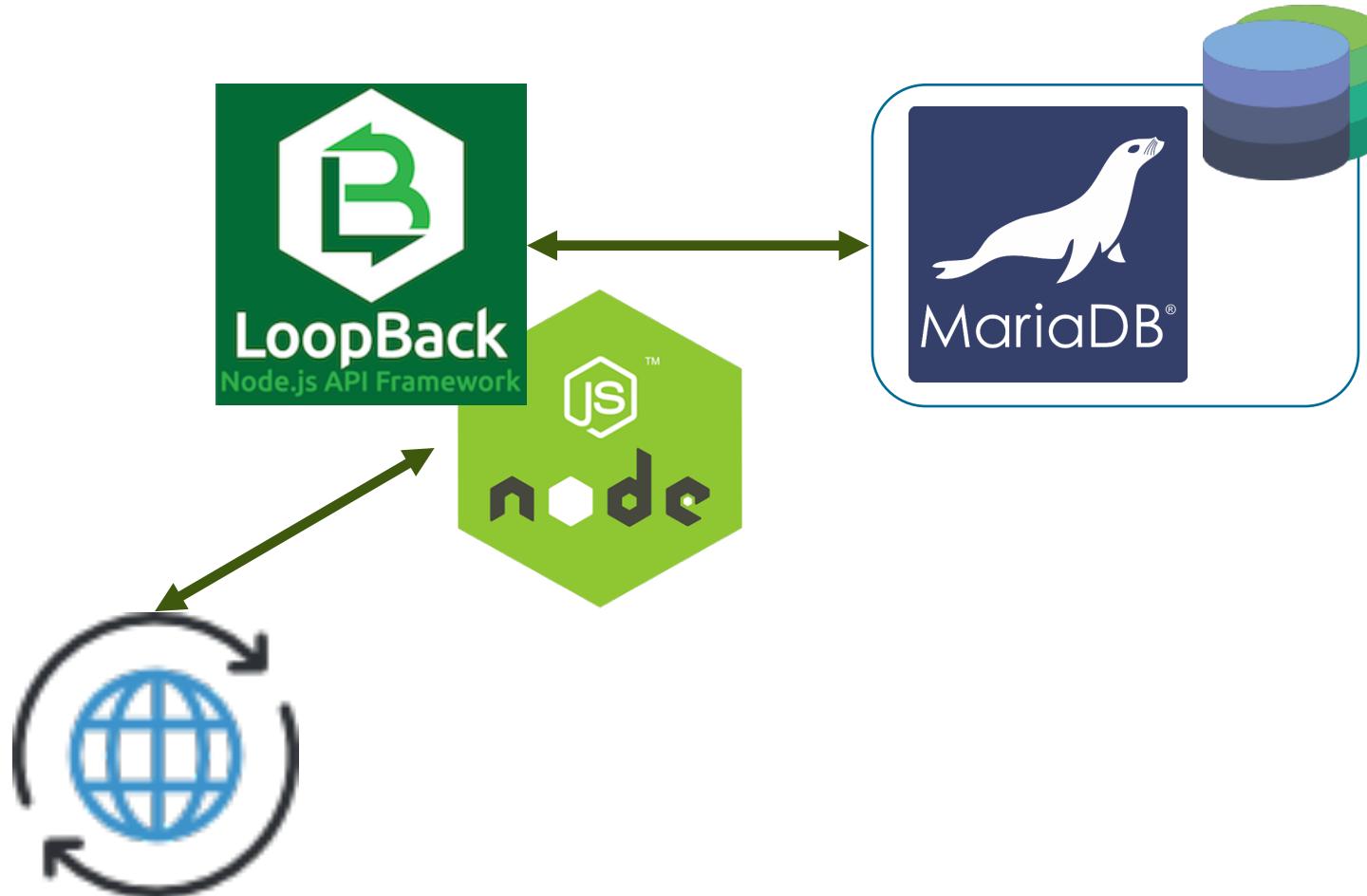


ANGULARJS

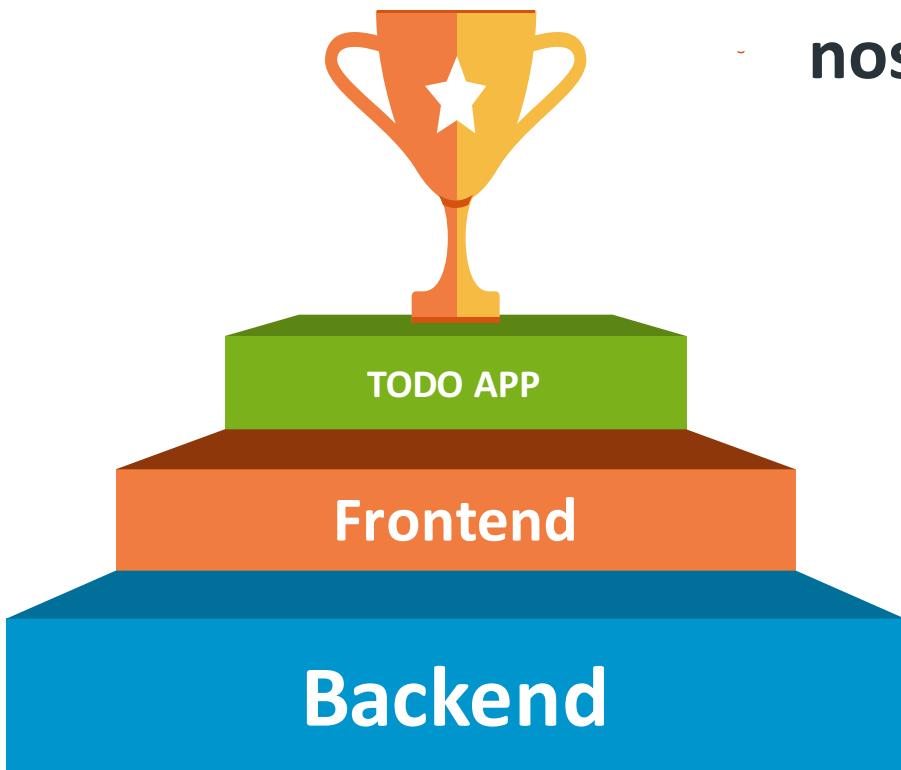
JS



Tecnologie Adottate - Backend



Scopo Finale



Uscire da questo corso con la nostra prima applicazione WEB:

TODO List



3. Javascript Base

the right way

Prof. Ardagna, Anisetti, Gaudenzi

Javascript: Caratteristiche

- **LOOSE TYPING**
- **SCOPING AND HOISTING**
- **FUNCTIONS AS FIRST-CLASS OBJECTS**
- **OBJECT ORIENTED**
- **ANONYMOUS FUNCTIONS**
- **FUNCTION BINDING**
- **CLOSURE FUNCTION**

JavaScript

- JavaScript è un linguaggio di scripting, tipicamente utilizzato client-side
- Nonostante la somiglianza nel nome, è un linguaggio completamente distinto da Java
- Come tutti i linguaggi di scripting, è interpretato
 - Il sorgente non deve essere compilato per essere eseguito
- L'interprete di JavaScript è generalmente contenuto all'interno del browser

JavaScript

- Storia
 - Definito da Netscape (LiveScript)
 - Nome modificato in JavaScript dopo accordo con Sun nel 1995
 - Microsoft lo chiama JScript (differenze minime)
 - Standard di riferimento: ECMAScript 262

JavaScript

- A differenza di HTML, JavaScript è case-sensitive
 - "ciao" è diverso da "Ciao"
- Purtroppo possono esserci incompatibilità e differenze tra i diversi browser
 - A volte si comportano in maniera diversa o non funzionano
- Si basa su due concetti principali:
 - DOM (Document Object Model)
 - Eventi (script event-driven)

Tipi

- JavaScript è un linguaggio debolmente tipato
 - Il tipo delle variabili (e dei parametri/argomenti delle funzioni) non viene dichiarato esplicitamente, ma definito implicitamente al primo assegnamento
 - `numero = 7;` numero è di tipo Number
- JavaScript converte automaticamente i tipi durante l'esecuzione (quando possibile)
- Tipi principali in JavaScript
 - Number: interi e decimali (virgola mobile); ad es: 7, 7.7
 - Boolean: valori booleani (vero/falso); ad es: true, false
 - String: sequenze di caratteri; ad es: "ciao", "claudio"

Variabili e istruzioni

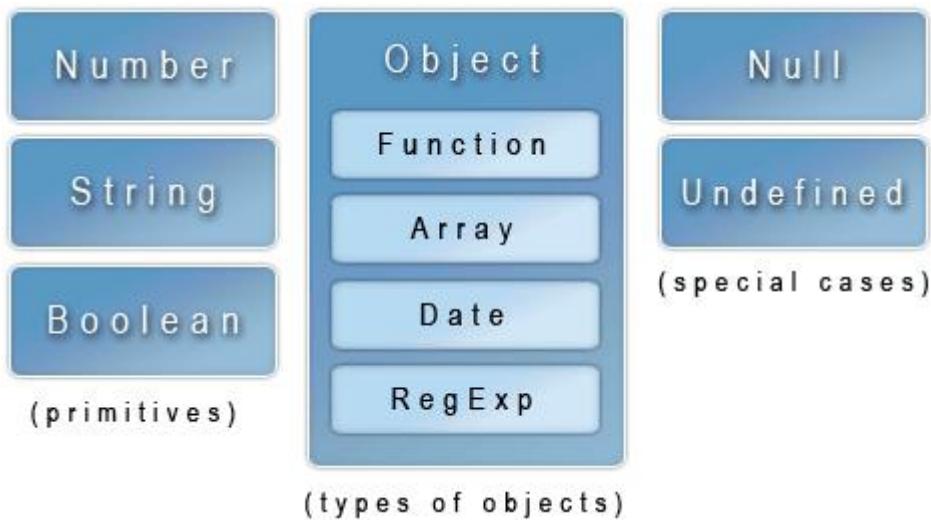
- Le dichiarazioni delle variabili globali non sono obbligatorie, ma si possono fare con la keyword var
 - Sono senza tipo
 - var variabile;
- Le dichiarazioni delle variabili locali sono obbligatorie, e devono essere fatte con la keyword var
 - Sono senza tipo
 - var prezzo_scontato;
- Le inizializzazioni sono facoltative (ma è buona norma farle all'atto della definizione e prima di utilizzare le variabili)
- Tutte le istruzioni JavaScript devono terminare con punto-e-virgola

Variabili

- Loosely typed
 - È possibile assegnare a una stessa variabile prima un valore stringa, poi un numero, poi altro ancora
- Ad esempio
 - alfa = 10
 - beta = "Claudio"
 - alfa = "Erika" // tipo diverso!!
- Sono consentiti incrementi, decrementi e operatori di assegnamento estesi (++, --, +=, ...)

Loose typing

- Variabili dichiarate senza tipo



- Coercione di tipo:
 - `7 + 7 + 7; // = 21`
 - `7 + 7 + "7"; // = 147`
 - `"7" + 7 + 7; // = 777`
- Coercione durante la comparaison
 - “`==`” vieta la coercione
 - “`!!`” casta a boolean

Variabili e Scope

- Due possibili scope
 - Globale, per le variabili definite fuori da funzioni
 - Locale, per le variabili definite esplicitamente dentro a funzioni (compresi i parametri ricevuti)
- ATTENZIONE: un blocco NON delimita uno scope!
 - Tutte le variabili definite fuori da funzioni, anche se dentro a blocchi innestati, sono globali
- ```
x = '3' + 2; // la stringa '32'
{
 { x = 5 } // blocco interno
 y = x + 3; // x denota 5, non "323"
}
```

# Scoping e hoisting

- Lo scopo non viene delimitato da blocchi quali if while ecc.
  - Non esiste il **block-level scope**
- L'unico delimitatore di scopo è la funzione
  - Si parla di **function-level scope**
- **Hoisting**: la buona pratica di dichiarare tutte le variabili all'inizio del loro scopo
  - Per evitare errori dovuti a variabili valorizzate prima della dichiarazione
  - Questo movimento è come se venisse fatto in modo invisibile dall'interprete Javascript.
    - Hoisted solo la dichiarazione non eventuali valorizzazioni o assegnamenti

# Tipo dinamico

- Operatore `typeof` ritorna il tipo di una espressione
  - Risolve le variabili incluse
    - `typeof(10/2)` = number
    - `typeof("stringa")` = string
    - `typeof(false)` = boolean
    - `typeof(document)` = object
    - `typeof(document.write)` = function
- Il tipo è dinamico: rappresenta il tipo in quel momento temporale e corrispondente al valore attuale della variabile (o dell'oggetto...)
  - `variabile = 10;`
    - `typeof(variabile)` = number
  - `variabile = "UNIMI";`
    - `typeof(variabile)` = string

# Stringhe

- Delimitate sia da virgolette sia da apici singoli
- Per annidare virgolette e apici, occorre alternarli
  - `document.write('<IMG src="image.gif">')`
  - `document.write("<IMG src='image.gif'>")`
- Concatenazione con +
  - `document.write("paolino" + 'paperino')`
  - Concatenazione fra stringhe e numeri comporta la conversione automatica del valore numerico in stringa
- Le stringhe JavaScript sono oggetti dotati di proprietà (ad es., `length`) e metodi (ad es., `substring(first,last)`)

# Array

- Lista rappresentata come array con indice a partire da 0
  - Le celle di un array JavaScript non hanno il vincolo di omogeneità in tipo: ogni cella può contenere indistintamente numeri, stringhe, oggetti, altri array, ...
- Creazione e inizializzazione di un array
  - Array vuoto
    - `var lista = new Array();`
    - `lista[0] = "Marco";`
    - `...`
  - Array inizializzato in fase di definizione
    - `lista = new Array("Marco", "Claudia", "Samuele");`

# Array

- Inserimento valori
  - I singoli elementi sono referenziati con l'usuale notazione a parentesi quadre: ad esempio, lista[x]
  - `lista[i] = "stringa";`
- Lettura contenuto in posizione i
  - `var elem = lista[i];`

# Array

- Lunghezza di un array (attributo length)
  - `var lunghezza = lista.length;`
- Per (sovra)scrivere l'ultimo elemento dell'array:
  - `lista[lunghezza-1] = "stringa";`
  - Ogni scrittura sovrascrive l'elemento che era memorizzato in precedenza
- Per leggere l'ultimo elemento dell'array
  - `var elem = lista[lunghezza-1];`

# Array - Costruzione Alternativa

- A partire da JavaScript 1.2, anche per gli array esiste un modo alternativo di costruzione: basta elencare la sequenza, racchiusa fra parentesi quadre, di valori iniziali separati da virgole
  - `vett = [ 1, -2, "tre" ]`

# Cicli

- Strutture di controllo per l'accesso sequenziale a un set di elementi (ad es., lista)
- I cicli sono generalmente basati sul concetto di indice (i): l'indice scorre lungo la lista indicando, via via, posizioni successive
- JavaScript supporta `for`, `while`, `do/while`, `for... in...`, `with`

# Code Style: Convenzioni

- La convenzione più usata è la
  - [Google Code Style Guide for JavaScript](#)
- Una convenzione interessante e di natura community
  - [Idiomatic.js](#)
    - Lista di best practice volte a far in modo che un codice seppur scritto da più mani sembri scritto da un unico programmatore
    - Include menzione su test framework
- **Linting:** analizzare il programma per verificare eventuali errori
  - lint utility originalmente creata (unix 1979) per programmi in C.
  - [JSHint](#) for javascript



# Coffee Break

10 minutes

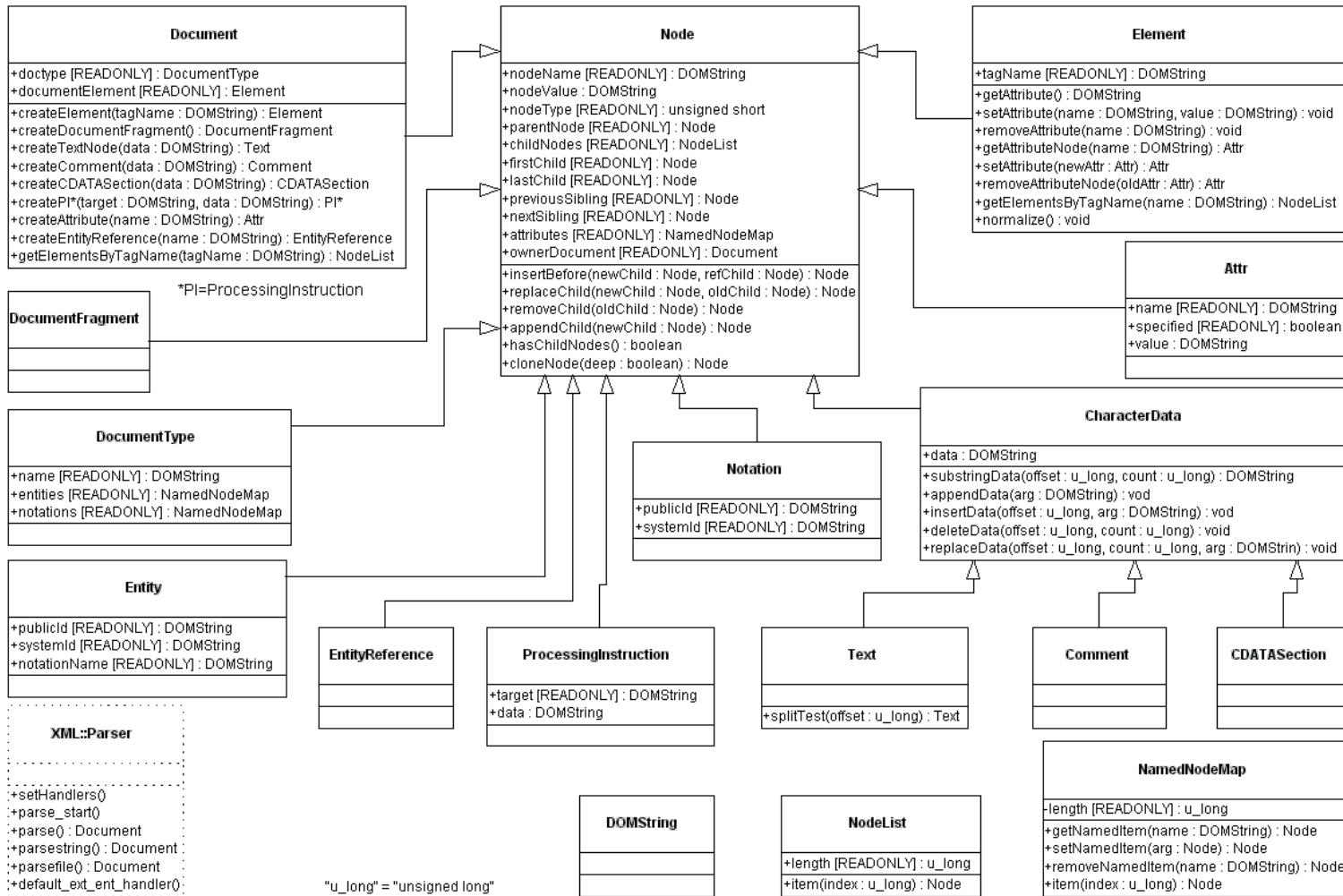


# Oggetti DOM, eventi, finestre, nodi

## Logical View

### DOM (Core) Level One

Invoke "getName" to read instance variable `name` when using XML::DOM or XML4J



# Document Object Model (DOM)

- È uno standard W3C (World Wide Web Consortium)
- Definisce uno standard per accedere documenti
  - "*The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.*"
- Separato in tre parti
  - Core DOM – modello standard per tutti i tipi di documenti
  - XML DOM – modello standard per documenti XML
  - HTML DOM – modello standard per documenti HTML

# HTML DOM

- È uno standard object model e programming interface per HTML
- Definisce
  - Elementi HTML come oggetti
  - Proprietà degli elementi HTML
  - I metodi per accedere agli elementi HTML
  - Gli eventi per tutti gli elementi HTML
- In altre parole, è uno standard che definisce come ottenere, cambiare, aggiungere o modificare elementi HTML

# HTML DOM

JavaScript

*core language*

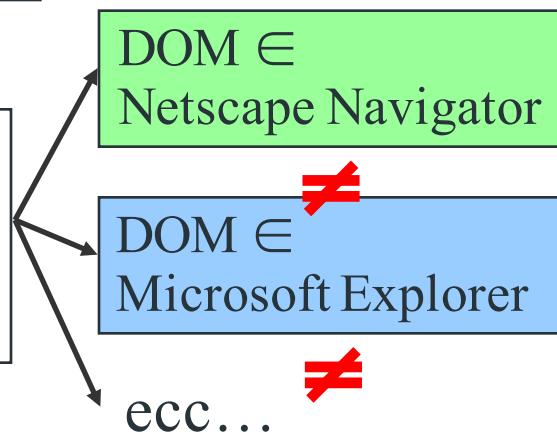
+ variabili, funzioni,  
ecc. definiti dall'utente



unico



**DOM** (*Document Object Model*):  
insieme di **oggetti predefiniti** che si  
riferiscono alla pagina Web, oppure  
al Web Browser



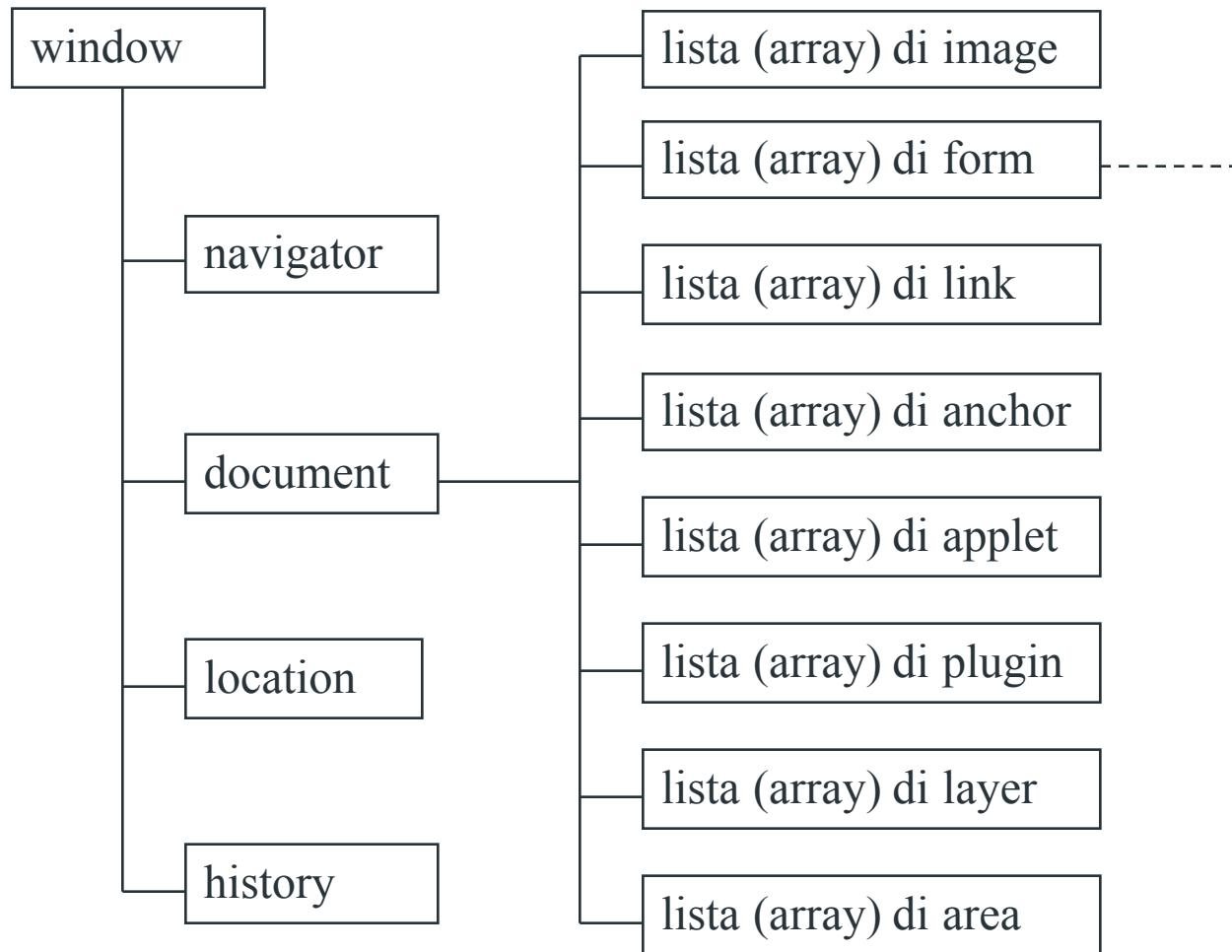
- JavaScript usa HTML DOM per modificare tutti gli elementi di una pagina web
  - Quando una pagina è caricata il browser crea il DOM della pagina
  - Pagina rappresentata come un albero
  - DOM è definito dal browser
    - Definizione è fatta separatamente per Explorer, Mozilla, ...
    - Possono nascere incompatibilità

# Oggetti (DOM)

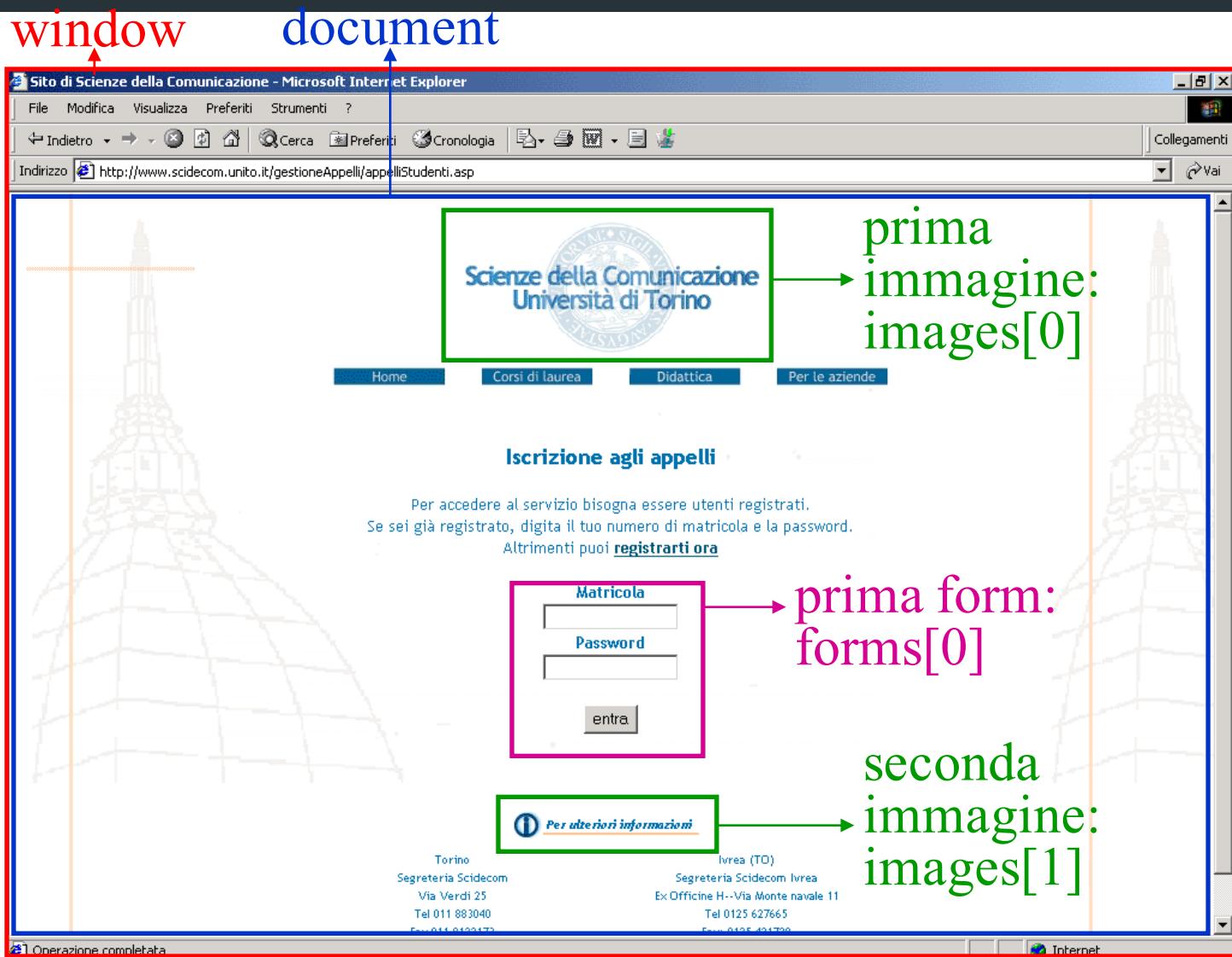
- Tramite il modello DOM a oggetti, JavaScript può
  - Cambiare tutti gli elementi e attributi HTML di una pagina
  - Cambiare tutti gli stili CSS
  - Rimuovere elementi e attributi HTML
  - Aggiungere elementi e attributi HTML
  - Reagire a eventi nella pagina
  - Creare nuovi eventi

# Oggetti (DOM): Liste di oggetti

- Il DOM è organizzato secondo una gerarchia:



# Oggetti (DOM)



# Oggetti (DOM): Window

- **Window (this)** = la finestra corrente del browser
- Figli dell'oggetto window
  - **navigator** = il browser (in quanto applicazione)
    - Ad esempio, per sapere quale browser si sta utilizzando (Explorer/ Netscape)
  - **document** = il contenuto della finestra
  - **location** = informazioni sull'URL corrente
    - Ad esempio, per caricare nella finestra un URL differente
  - **history** = elenco degli URL visitati di recente
    - Ad esempio, per tornare alla pagina Web precedente

# Document

- Rappresenta il documento corrente: il contenuto della pagina web attuale
  - Da non confondere con la finestra corrente!
- Componenti principali di document
  - forms (array) = lista dei moduli (form) contenute nella pagina
  - elements (array di Buttons, Checkbox, etc etc...)
  - anchors (array)
  - links (array)
  - images (array) = lista delle immagini contenute nella pagina
  - applets (array)
  - embeds (array)

# Document

- Diversi metodi disponibili
  - Metodo write stampa un valore a video: stringhe, numeri, immagini, ...
- Esempi (`document_write.html`)
  - `document.write("paperone")`
  - `document.write(18.45 - 34.44)`
  - `document.write('paperino')`
  - `document.write('<IMG src="image.gif">')`
  - `document.write = this.document.write`
- Altro esempio: file HTML con immagine con nome "image\_1":
  - `document.image_1.width` si riferisce alla larghezza dell'immagine
  - `document.image_1.width = 40`

# Oggetti: proprietà e funzioni

- Oggetto è una collezione di
  - proprietà (variabili): sono a loro volta oggetti
  - funzioni (metodi, operazioni)
- Per accedere alle proprietà di un oggetto

```
window.status = 'hello!';
```

nome oggetto   nome proprietà

- Per invocare le funzioni di un oggetto

```
window.print();
```

funzione

```
window.document.write("<p>Ciao!</p>");
```

funzione

nome oggetto

- Per accedere agli oggetti contenuti in una lista (array)

```
window.document.images[0].src = 'sole.gif';
```

la proprietà *src* della prima immagine della pagina

# Oggetti: proprietà e funzioni

- L'invocazione di una funzione apparentemente senza alcun oggetto si riferisce all'oggetto window
  - `prompt("Come ti chiami?", "boh");`
  - `window.prompt("Come ti chiami?", "boh");`
- Un riferimento all'oggetto `document` non preceduto da un riferimento all'oggetto `window`, è equivalente al caso in cui `document` è preceduto da `window` (implicito)
  - `document.write("<p>Ciao!</p>");`
  - `window.document.write("<p>Ciao!</p>");`

# Eventi

- I programmi JavaScript sono tipicamente "guidati dagli eventi" (event-driven)
  - Eventi sono scatenati da azioni dell'utente sulla pagina Web
    - Ogni volta che l'utente scrive qualcosa in una casella, preme un pulsante, ridimensiona una finestra ecc... genera un "evento"
  - Un programma JavaScript deve contenere un gestore di eventi (event handler), che sia in grado di ricevere e interpretare le azioni dell'utente (eventi)
  - Il DOM fornisce una serie di gestori di eventi predefiniti
    - L'accadere di un evento nella pagina Web mette automaticamente in azione il corrispondente gestore di eventi

# Eventi

- <a href="#" onClick = "window.print()">
  - Attributo onClick: evento click innesca il gestore
  - window.print() è un codice JavaScript che viene eseguito dal gestore
  - href
    - "#" resta nella pagina in cui si trova (salta al Top)
    - URL va alla pagina indicata (carica la nuova pagina)
    - <A HREF="#" onClick="JavaScript:window.print(); return false;"> il browser resta nella pagina corrente, senza saltare al Top

# Eventi

- Un'istruzione JavaScript può essere inserita all'interno di un tag HTML, (anziché essere racchiusa nei tag <SCRIPT>...</SCRIPT>)
- In questi casi, il gestore di evento invocato farà riferimento al tag in cui si trova l'istruzione
  - <A HREF="#" onClick="window.print()">
    - Quando l'utente fa click (onClick) sul link (<A...>)
  - <FORM NAME="modulo" onSubmit="alert('Ciao!');">
    - Quando l'utente invia (onSubmit) il modulo (<FORM...>)
  - <INPUT TYPE="text" NAME="login" onFocus="...;">
    - Quando l'utente porta il cursore (onFocus) nel campo di testo (<INPUT TYPE="text"...>)
  - <BODY onLoad="jump()">
    - Quando l'utente carica (onLoad) la pagina (<BODY...>)

# Eventi

- Gli eventi intercettabili su un link: onClick, onMouseOver, onMouseOut
- Gli eventi intercettabili su una finestra: onLoad, onUnLoad, onBlur
- Esempio:
  - <BODY onLoad = "alert('caricato')" >  
  <FORM name="myform">  
    <INPUT type="button" name="bottone"  
      value="Premi qui"  
      onClick = "document.write(sum(1,13))" >  
  </FORM>  
</BODY>

# Gestione eventi

- Per sfruttare il valore restituito da confirm, prompt, o qualsiasi altra funzione JavaScript occorre inserire come valore dell'attributo onClick un programma JavaScript (una sequenza o una chiamata di funzione)
- Esempi:
  - `onClick= "x = prompt('Cognome e Nome:') ;  
document.write(x)"`
  - `onClick= "ok = confirm('Va bene così?') ;  
if(!ok) alert('ATTENTO...')"`

# Accesso agli oggetti nella pagina

- Lista dei moduli (<FORM...>) contenuti in una pagina
  - `window.document.forms[i]`
- |- Lista delle immagini (<IMG...>) contenute in una pagina
  - `window.document.images[i]`

# Accesso agli oggetti nella pagina

- I campi di testo sono oggetti dotati di nome posti all'interno di un oggetto form pure esso dotato di nome
  - Come tali sono referenziabili con la "dot notation":  
document.nomeform.nomeTextField
- Il campo di testo è caratterizzato dalla proprietà value
- Esempio:
  - <FORM name="myform">  
  <INPUT type="text" name="cognome" size=20>  
  <INPUT type="button" name="bottone" value="Show"  
    onClick="alert(document.myform.cognome.value)">  
</FORM>

# Accesso agli oggetti nella pagina

- Attributo NAME:
  - <FORM NAME="modulo">  
    <INPUT TYPE="text" NAME="codice\_fiscale">
    - window.document.modulo.codice\_fiscale.value =...
  - <IMG SRC="claudio.gif" NAME="claudio">
    - window.document.claudio.src=...
- Oppure l'attributo ID:
  - <INPUT TYPE="text" ID="codice\_fiscale">
    - window.document.getElementById("codice\_fiscale")=...
  - <IMG SRC="claudio.gif" ID="claudio">
    - img=window.document.getElementById("claudio");  
    img.src=...

# Eventi (form)

- JavaScript permette di
  - Intercettare eventi che "accadono" nei campi di un modulo
  - Modificare i campi di un modulo

# Eventi (form)

- Un form contiene solitamente campi di testo e bottoni

```
<FORM name="myform">
```

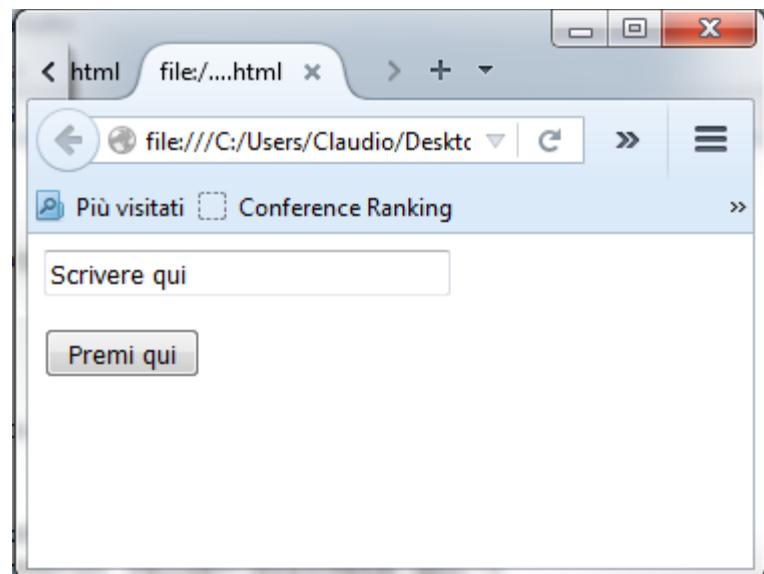
```
 <INPUT type="text" name="campoDiTesto"
 size=30 maxlength=30 value="Scrivere qui">
```

```
 <P>
```

```
 <INPUT type="button" name="bottone"
 value="Premi qui">
```

```
</FORM>
```

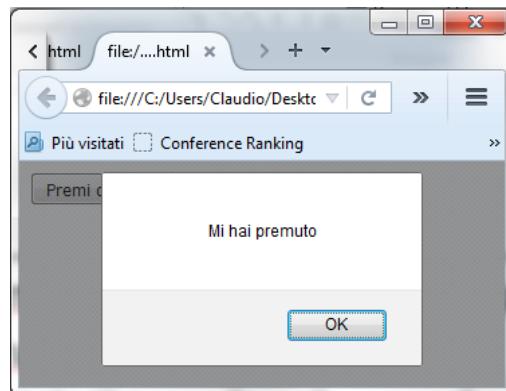
- Quando il bottone viene premuto è possibile invocare una funzione JavaScript



# Eventi (form)

- Quando si preme il bottone, l'evento bottone premuto può essere intercettato mediante l'attributo onClick

```
<FORM name="myform">
 <INPUT type="button" name="bottone"
 value="Premi qui"
 onClick = "alert('Mi hai premuto')" >
</FORM>
```

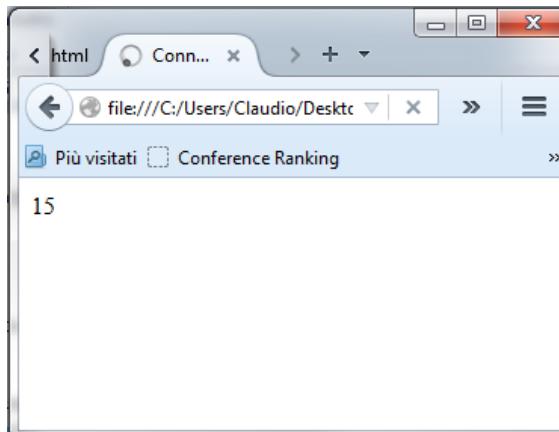


# Eventi (form)

- ALTERNATIVA: quando si preme il bottone, far scrivere il risultato di una funzione (form\_print\_temporeale.html)

```
<FORM name="myform">
 <INPUT type="button" name="bottone"
 value="Premi qui"
 onClick = "document.write(15)" >
```

```
</FORM>
```



# Eventi (onLoad)

- Evento scatenato quando un oggetto viene caricato
  - Ad esempio permette di ridirezionare l'utente ad un altro URL:

```
<SCRIPT language="JavaScript">
function jump(){
 window.location.href="http://www.unimi.it";
}
</SCRIPT>
</HEAD>
<BODY onLoad = "jump()">
```
- onLoad carica il gestore che viene innescato tramite la funzione jump()
  - Ridireziona all'URL puntata dalla proprietà href dell'oggetto location

# Gestione finestre (open)

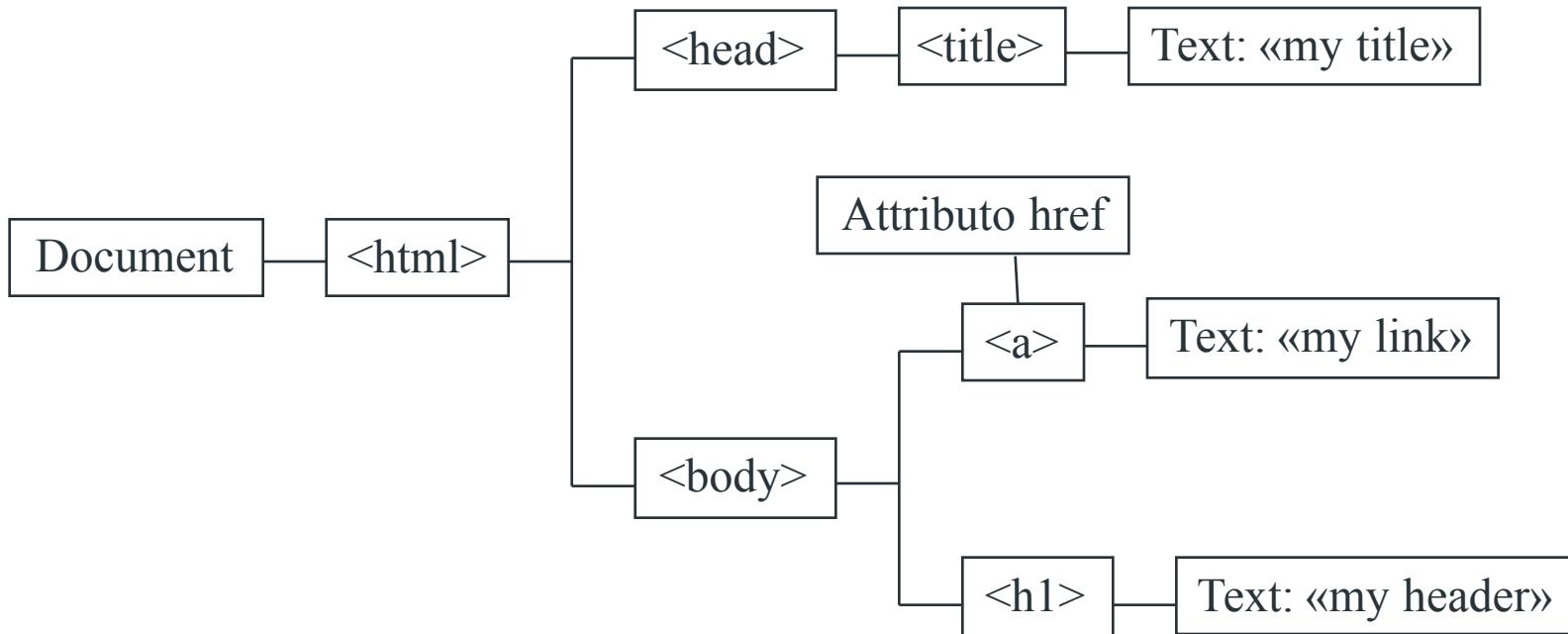
- Oggetto window permette di agire e gestire le finestre del browser
  - `window.open(URL, nome, [proprietà]);` apre una nuova finestra (o una nuova scheda)
    - URL = indirizzo della pagina da caricare
    - Nome = identificatore della finestra
    - Proprietà (opzionale): lista delle proprietà della nuova finestra (se omesso, la nuova finestra mantiene le proprietà della corrente)
    - Ritorna un riferimento alla finestra aperta (null se c'è errore)
- Esempio: apertura finestra nella stessa finestra/scheda
  - `<a href="#" onClick = "window.open('http://www.di.unimi.it', 'pippo'); return false;">nuova finestra!</a>`

# Funzioni come link

- Una funzione JavaScript costituisce un valido link utilizzabile nel tag HTML `<a href= ...> </a>`
- L'effetto del click su tale link è l'esecuzione delle funzione e l'apparizione del risultato in una nuova pagina HTML all'interno però della stessa finestra
- Esempio:
  - `<a href="JavaScript:Math.sum(43,58)"> Questo dovrebbe essere 101 </a>`

# Accesso agli oggetti di una pagina tramite nodi

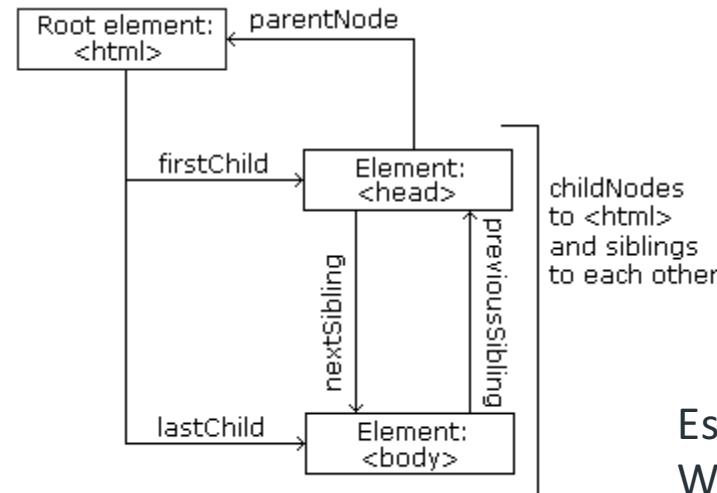
- Ogni oggetto è un nodo (HTML DOM)
  - Ogni elemento è un nodo
  - Ogni testo è un nodo testo
  - Ogni attributo un nodo attributo
  - I commenti sono dei nodi commenti



# Relazione tra nodi

- Definita attraverso diverse funzioni
  - Parent, child, sibling
  - Root è la radice del documento (<html>)
  - Ogni nodo ha un solo nodo padre a parte la root
  - Ogni nodo può avere 0 o più figli
  - Sibling sono nodi con lo stesso padre

```
<html>
 <head>
 <title>DOM Tutorial</title>
 </head>
 <body>
 <h1>DOM Lesson one</h1>
 <p>Hello world!</p>
 </body>
</html>
```



Esempio tratto da  
W3School

# Funzioni, modello a oggetti

# Introduzione

- Object-based language (ma non object-oriented)
  - Usa l'idea di incapsulare stato e operazioni all'interno di oggetti
  - Non applica i concetti di ereditarietà e sottotipi
- JavaScript a volte riferito come prototype-based language
  - Non esistono classi, ma gli oggetti ereditano codice e dati da altri oggetti template
  - Vengono clonati oggetti che servono come «prototipi»

# Definizione di funzioni

- Definite tramite keyword *function* e sempre racchiuso in un blocco
- Possono essere considerate sia procedure...
  - Non ha istruzione return
  - ```
function printSum(a,b) {  
    document.write(a+b)  
}
```
- ... sia funzioni in senso proprio (non esiste la keyword void)
 - ```
function sum(a,b) { return a+b }
```
- I parametri formali sono senza dichiarazione di tipo

# Chiamata a funzione

- Chiamate come in un linguaggio di programmazione tradizionale, fornendo la lista dei parametri attuali
  - `document.write(sum(10,5) + "<br/>")`
  - `printSum(19, 34.33)`
- Se i tipi non hanno senso per le operazioni fornite, l'interprete JavaScript ritorna un errore a runtime
  - Non viene mostrato il risultato

# Parametri di funzione

- Passaggio di parametro per valore
  - Nel caso di oggetti, si copiano riferimenti
- A differenza di C e Java, è lecito definire una funzione dentro un'altra funzione (es. Pascal)
- Se i parametri attuali sono più di quelli necessari
  - Nessun errore
  - Quelli extra vengono ignorati
- Se i parametri attuali sono meno di quelli necessari
  - Quelli mancanti sono inizializzati a undefined (una costante di sistema)

# Variabili: tipi di dichiarazione

- Dichiarazione delle variabili è
  - Implicita o esplicita per variabili globali
  - Necessariamente esplicita, per variabili locali
- Dichiarazione esplicita (keyword var)
  - `var pippo = 10 // dichiarazione esplicita`
  - `pippo = 10 // dichiarazione implicita`
- La dichiarazione implicita è sempre e solo per variabili globali
- La dichiarazione esplicita ha un effetto che dipende da dove si trova la dichiarazione

# Variabili: dichiarazione esplicita

- Fuori da funzioni, la parola chiave var è ininfluente
  - La variabile definita è globale
- All'interno di funzioni, la parola chiave var ha un significato preciso
  - Indica che la nuova variabile è locale, ossia ha come scope la funzione
- All'interno di funzioni, una dichiarazione senza la parola chiave var introduce una variabile globale

```
x=6 // globale
function test() {
 x = 18 // globale
}
test()
// qui x vale 18
```

```
var x=6 // globale
function test() {
 var x = 18 // locale
}
test()
// qui x vale 6
```

# Variabili: environment di riferimento

- Quando ci si riferisce a una variabile
  - Prima si cerca localmente
  - Se non è definita si accede a quella globale
- Esempio in ambiente globale
  - `var f = 4` // f è comunque globale
  - `g = f * 3` // g è comunque globale, e vale 12
- Esempio in ambiente locale (dentro a funzioni)
  - `var f = 5` // f è locale
  - `g = f * 3` // g è globale, e vale 15

# Variabili: environment di riferimento

```
pippo=10;
var fz=function()
{
 alert(pippo);
 var pippo=4;
}
fz();
```

- Cosa viene stampato?

# Variabili: environment di riferimento

```
pippo=10;
var fz=function()
{
 alert(pippo);
 var pippo=4;
}
fz();
```

- Cosa viene stampato? pippo=undefined (hoisting)
  - Valorizzato dopo dichiarato prima (hoisting ma senza valore)

# Variabili: environment di riferimento

- Il parser prima analizza tutto il codice e crea tutte le variabili e strutture lasciandole undefined
- Poi esegue una riga alla volta
  - Quando raggiungo la chiamata alert(pippo) cerca una variabile locale e la trova undefined
  - Stampa quindi undefined

# Variabili: environment di riferimento

- pippo=10;

```
var fz=function()
```

```
{
```

```
 alert(pippo);
```

```
}
```

```
fz();
```

- Cosa viene stampato?

# Variabili: environment di riferimento

- pippo=10;

```
var fz=function()
{
 alert(pippo);
}
fz();
```
- Cosa viene stampato? pippo=10
  - Non trovando la variabile locale, stampa la globale

# Funzioni e chiusure

- La natura interpretata di JavaScript e l'esistenza di un ambiente globale pongono una domanda
  - Quando una funzione usa un simbolo non definito al suo interno, quale definizione vale per esso?
    - La definizione che esso ha nell'ambiente in cui la funzione è definita, oppure
    - La definizione che esso ha nell'ambiente in cui la funzione è chiamata?

# Funzioni e chiusure

- Si consideri il seguente programma JavaScript
  - var x = 20;  
function provaEnv(z) { return z + x; }  
alert(provaEnv(18)) // visualizza certamente 38  
function testEnv() {  
 var x = -1;  
 alert(provaEnv(18)); // COSA visualizza ???  
}
- Nella funzione testEnv si ridefinisce il simbolo x, poi si invoca la funzione provaEnv, che usa il simbolo x ... ma QUALE x?
- Nell'ambiente in cui provaEnv è definita, il simbolo x aveva un altro significato rispetto a quello che ha ora!

# Funzioni e chiusure

- ```
var x = 20;
function provaEnv(z) { return z + x; }
function testEnv() {
    var x = -1;
    return provaEnv(18); // COSA visualizza ???
}
```
- Se vale l'ambiente esistente all'atto dell'uso di `provaEnv`, si parla di chiusura dinamica; se prevale l'ambiente di definizione di `provaEnv`, si parla di chiusura lessicale
 - JavaScript adotta la chiusura lessicale → `testEnv` visualizza ancora 38 (non 17)

Classi – Oggetti - Proprietà

- Non esiste la keyword class ma
 - var Person = function () {};
 - E' di per se una classe con un costruttore (funzione) empty
- Oggetto istanza di una classe (new)
 - var person1 = new Person();
- Proprietà (attributo dell'oggetto)
 - var Person = function (firstName) {
 - this.firstName = firstName;
 - };

Funzioni come dati

- Le funzioni sono **first-class object**
- Variabili possono riferirsi a funzioni
 - La funzione non ha nome (anche se potrebbe)
 - `var f = function (z) { return z*z; }`
 - La funzione viene invocata tramite il nome della variabile
 - `var result = f(4);`
 - `g = f; //alias`
- Possibile passare funzioni come parametro ad altre funzioni
 - `function calc(f, x) {return f(x); }`
 - Se `f` cambia, `calc` calcola una funzione diversa

Funzioni come dati - Esempi

- `function calc(f, x) { return f(x) }`
 - `calc(Math.sin, .8)` ritorna `0.7173560908995228`
 - `calc(Math.log, .8)` ritorna `-0.2231435513142097`
- Altri esempi
 - `calc(x*x, .8)` ritorna un errore
 - `x*x` non è un oggetto funzione del programma
 - `calc(funz, .8)` va bene solo se la variabile `funz` fa riferimento a un costrutto `function`
 - `calc("Math.sin", .8)` ritorna errore
 - `"Math.sin"` è una stringa non una funzione
 - Il nome di una funzione non è la funzione

Funzioni come dati - Conseguenze

- Per utilizzare una funzione come dato occorre avere effettivamente un oggetto funzione
- Non si può sfruttare questa caratteristica per far eseguire una funzione di cui sia noto solo il nome (letto da tastiera)
 - `calc("Math.sin", .8)` ritorna errore
- o di cui sia noto solo il codice
 - `calc(x*x, .8)` ritorna errore
- Il problema è risolvibile
 - Oppure si costruisce esplicitamente un "oggetto funzione" (`Function`)

Oggetti

- Un oggetto JavaScript è una collezione di dati dotata di nome
 - Ogni dato interpretabile come una proprietà
 - Per accedere alle proprietà si usa la "dot notation"
 - nomeOggetto.nomeProprietà
 - Tutte le proprietà sono accessibili
- Un oggetto JavaScript è costruito tramite costruttore
 - Stabilisce la struttura dell'oggetto e quindi le sue proprietà
 - I costruttori sono invocati mediante l'operatore new
 - In JavaScript non esistono classi
 - Il nome del costruttore è a scelta dell'utente (indica implicitamente la "classe")

Oggetti: Definizione

- La struttura di oggetto JavaScript viene definita dal costruttore usato per crearlo
- È all'interno del costruttore che si specificano le proprietà (iniziali) dell'oggetto, elencandole con la dot notation e la keyword this
- Identificatore globale (function expression)
 - Point = function(i,j){
 this.x = i;
 this.y = j;
}
- Identificatore locale (function declaration)
 - function Point(i,j){
 this.x = i;
 this.y = j;
}
- La keyword this è necessaria, altrimenti ci si riferirebbe all'environment locale della funzione costruttore

Oggetti: Costruzione

- Per costruire oggetti si applica l'operatore new a una funzione costruttore
 - `p1 = new Point(3,4);`
 - `p2 = new Point(0,1);`
 - L'argomento di new non è il nome di una classe, è solo il nome di una funzione costruttore.
- A partire da JavaScript 1.2, si possono creare oggetti anche elencando direttamente le proprietà con i rispettivi valori
 - Sequenza di coppie nome:valore separate da virgole e racchiusa fra parentesi graffe.
 - `p3 = { x:10, y:7 }`
 - Il valore potrebbe essere una funzione

Oggetti: Accesso alle proprietà

- Proprietà di un oggetto sono pubbliche e accessibili
 - Esistono anche proprietà "di sistema" e come tali non visibili, né enumerabili con gli appositi costrutti
- Accesso attraverso dot notation
 - `p1.x = 10;` // da (3,4) diventa (10,4)

Costrutto WITH

- Utile per accedere a più proprietà metodi di uno stesso oggetto
- Evita di ripetere il nome dell'oggetto
 - `with (p1) x=22, y=2` equivale a `p1.x=22, p1.y=2`
 - `with (p1) {x=3; y=4}` equivale a `p1.x=3, p1.y=4`

Aggiunta e rimozione di proprietà

- Le proprietà specificate nel costruttore sono le proprietà iniziali
- È possibile aggiungere dinamicamente nuove proprietà semplicemente nominandole e usandole
 - `p1.z = -3; // da {x:10, y:4} diventa {x:10, y:4, z: -3}`
 - NB: non esiste il concetto di classe come "specifica della struttura (fissa) di una collezione di oggetti", come in Java o C++
- È possibile rimuovere dinamicamente proprietà, mediante l'operatore delete
 - `delete p1.x; // da {x:10, y:4, z: -3} diventa {y:4, z: -3}`

Metodi per (singoli) oggetti

- Definire metodi è semplicemente un caso particolare dell'aggiunta di proprietà
- Non esistendo il concetto di classe, un metodo viene definito per uno specifico oggetto (ad esempio, p1) non per tutti gli oggetti della stessa "classe"
- Metodo getX per p1:
 - `p1.getX = function(){ return this.x; }`
 - Al solito, this è necessario per evitare di riferirsi all'environment locale della funzione costruttore.
- E se si vuole definire lo stesso metodo per più oggetti?

Metodi per più oggetti

- Un modo per definire lo stesso metodo per più oggetti consiste nell' assegnare tale metodo ad altri oggetti
 - `p2.getX = p1.getX`
 - Definisce il metodo `getX` anche in `p2`
 - ATTENZIONE: non si sta chiamando il metodo `getX`, lo si sta referenziando → non c'è l'operatore di chiamata `()`
- Esempio d'uso
 - `document.write(p1.getX() + "
")`
 - L'operatore di chiamata `()` è necessario al momento dell'uso (invocazione) del metodo

Metodi per una "classe" di oggetti

- In assenza del concetto di classe, assicurare che oggetti "dello stesso tipo" abbiano lo stesso funzionamento richiede un'opportuna metodologia
- Un possibile approccio consiste nel definire tali metodi dentro al costruttore
 - Point = function(i,j) {
 this.x = i; this.y = j;
 this.getX = function(){ return this.x }
 this.getY = function(){ return this.y }
}

Metodi: Invocazione

- L'operatore di chiamata () è quello che effettivamente invoca il metodo
 - `document.write(p1.getX() + "
")` permette di invocare il metodo `p1.getX = function() { return this.x; }`
- ATTENZIONE: se si invoca un metodo inesistente si ha errore a run-time (metodo non supportato)
 - NB: se l'interprete JavaScript incontra un errore a run-time, non esegue le istruzioni successive e spesso non visualizza alcun messaggio d'errore!

Simulare proprietà "private"

- Le proprietà di un oggetto sono pubbliche
- Proprietà private possono essere simulate definendo variabili locali alla funzione costruttore

```
Rettangolo = function(){  
    var latoX, latoY;  
    this.setX = function(a) { latoX = a }  
    this.setY = function(a) { latoY = a }  
    this.getX = function() { return latoX }  
    this.getY = function() { return latoY }  
}
```

- La keyword `this` rende visibili a tutti le quattro proprietà `setX`, `setY`, `getX` e `getY`, mentre `latoX` e `latoY` sono visibili solo nell'environment locale della funzione costruttore e dunque "private"

Variabili e metodi "di classe"

- In assenza del concetto di classe, le variabili di classe si possono modellare ad esempio come proprietà dell'"oggetto costruttore" (che è una funzione - e quindi un oggetto - esso stesso). Ad esempio
 - `p1 = new Point(3,4); Point.color= "black";`
- Analogamente si possono modellare come proprietà dell'"oggetto costruttore" anche i metodi di classe
 - `Point.commonMethod= function(...) {...}`

Metodi: prototype

- **object prototype**

```
function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}
```

- Con **new** creo un oggetto di questo *object prototype*
 - Aggiungo metodi facilmente ai singoli oggetti
- Assegnare un nuovo metodo ad un *object prototype (classe)*

```
Person.prototype.sayHello = function() {  
    console.log("Hello, I'm " + this.firstName);};
```
- **In JavaScript i metodi sono funzioni associate ad un oggetto come proprietà.**
 - **Posso invocare il metodo fuori dal contesto**
 - var helloFunction = person1.sayHello;
 - **helloFunction();** //undefined o errore in strict mode, oggetto globale
 - helloFunction.call(person1); //ok esplicito call

Metodi Privileged

- Metodo che ha accesso a variabili private ma che espone esso stesso come pubblico
- Esempio completo

```
// Costruttore
function Ragazzo (nome) {
    // Privato
    var idolo = "Alan Turing";
    // Privileged
    this.getIdolo = function () {
        return idolo;
    };
    // Pubblico
    this.nome = nome;
}
```

```
// Pubblico
Ragazzo.prototype.getNome =
function () {
    return this.nome;
};
// Statico o di classe
Ragazzo.citta = "Crema";
```

Encapsulation - Abstraction

- E' limitata alla visibilità in catena gerarchica
 - Vedo i metodi del padre anche se non li ridefinisco
 - Nessuna definizione di version stringenti protected private etc.
- Abstraction: meccanismo che permette di modellare il problema come *specializzazione* o *composizione*
 - *Specializzazione*= ereditarietà
 - *Composizione* = istanze di classe come valori di attributi di oggetti

Ereditarietà (1)

- Supporto solo per ereditarietà singola
- Assegno una istanza di una classe padre alla classe figlia e di conseguenza specializzando

```
// costruttore
var Person = function(firstName) {
    this.firstName = firstName;
}
Person.prototype.sayHello = function(){
    console.log("Hello, I'm " + this.firstName);
}

// Studente
function Student(firstName, subject) {
    // Chiamo il costruttore del padre (call)
    // settando this in modo corretto
    Person.call(this, firstName);
    // Proprietà specifica dello studente
    this.subject = subject;
}
```

Ereditarietà (2)

- ```
Student.prototype = Object.create(Person.prototype);
// seta il costruttore per riferirsi a Student
Student.prototype.constructor = Student; //utile in casi
specifici)
// Rimpiazza un metodo
Student.prototype.sayHello = function(){
 console.log("Hello, I'm " + this.firstName + ". I'm
studying " + this.subject + ".");
}
• Uso
 • var student1 = new Student("Janet", "Applied Physics");
```

# Polimorfismo

- Classi diverse possono definire metodo con lo stesso nome
  - La classe definisce lo scopo
- Se le classi sono in relazione gerarchica allora ho sovrascrittura

# Javascript: paradigma funzionale

- Funzioni come first class objects
- A function è
  - Una istanza di un Object type
  - Può avere proprietà e ha un link al suo metodo costruttore
  - Posso salvare una funzione in una variabile
  - Posso passarla come parametro
  - Posso tornare una funzione come ritorno di un'altra

# Oggetti Function

- Permettono di definire funzioni
  - Ogni funzione JavaScript è un oggetto
  - Definizione implicita tramite il costrutto function
  - Definizione esplicita tramite il costruttore Function

# Oggetti Function: definizione implicita

- Definizione implicita tramite il costrutto function
  - Argomenti sono i parametri formali della funzione
  - Il corpo della funzione è racchiuso tra parentesi graffe
    - `funzione = function(x) { return f(x) }`
  - Costruito dentro il programma JavaScript
    - Valutato una sola volta
    - Efficiente ma non flessibile

# Oggetti Function: definizione esplicita

- Definizione esplicita tramite il *costruttore Function*
  - Argomenti sono tutte stringhe e rappresentano i parametri della funzione definita
  - Solo l'ultimo argomento ha un comportamento diverso e rappresenta il corpo della funzione
    - `funzione = new Function("x", "return f(x)")`
  - Costruito a partire da stringhe
    - Valutato ogni volta
    - Poco efficiente (non è parsato con il resto del codice), ma molto flessibile

# Scope: Function e function declaration

- Le funzioni create con il costruttore Function non creano una chiusura sul loro contesto di creazione
- Son sempre create nello scopo globale.
- Sono in grado di accedere solo alle loro variabili locali o globali non quelle dell'*outer scope* (*scope di chiamata*)

```
var x = 10;
function createFunction1() {
 var x = 20;
 return new Function("return x;") // x globale;
}
function createFunction2() {
 var x = 20;
 function f() {
 return x; // x locale
 return f;
 }
}
```

```
var f1 = createFunction1();
console.log(f1()); // 10
var f2 = createFunction2();
console.log(f2()); // 20
```

# Funzioni come dati: Oggetto function

- Riprendiamo la funzione `function calc(f, x) { return f(x) }`
  - `f` deve essere un oggetto funzione
  - `calc(Math.sin, .8)`      OK
  - `calc(x*x, .8)` NO
- Costrutture Function ci viene in aiuto
  - Data il codice definiamo un oggetto funzione che passiamo come parametro alla funzione calc
  - `calc(new Function("x", "return x*x"), .8)` OK

# Funzioni come dati: Esempio

- Funzione on demand
  - Inserire la funzione da calcolare
    - `var funzione = prompt("Scrivere f(x): ")`
  - Inserire il/i parametri da usare
    - `var x = prompt("Calcolare per x = ? ")`
  - Calcolare la funzione (invocazione riflessiva)
    - `var f = new Function("x", "return " + funz)`
  - Mostrare il risultato
    - `confirm("Risultato: " + f(x))`

# Funzioni come dati: Problema

- Valori immessi da linea di comando (prompt) sono stringhe
  - Ad esempio, una funzione che incrementa un numero ( $x+1$ ) viene considerata come un'operazione di concatenazione
  - $x+1$  con  $x=10$  ritorna 101
- Contromisure
  - Utente specifica il tipo del dato attraverso una conversione esplicita, ad esempio `parseInt(x)`
  - Programma implementa la conversione esplicita dopo il prompt:
    - `var x = parseInt(prompt("Calcolare per x = ? "))`
    - `typeof(x) = number`

# Oggetti Function: proprietà

- Proprietà statiche: (esistono anche mentre non esegue)
  - length - numero di parametri formali (attesi)
  - Name - nome
- Proprietà dinamiche: (mentre la funzione è in esecuzione)
  - arguments - array contenente i parametri attuali
  - arguments.length - numero dei parametri attuali
  - arguments.callee - la funzione stessa in esecuzione
  - caller - il chiamante (null se invocata da top level)
  - constructor - riferimento all'oggetto costruttore
  - prototype - riferimento all'oggetto prototipo

# Oggetti Function: metodi

- Metodi invocabili su una funzione (`toString`-`valueOf.html`)
  - `toString` – chiamata automaticamente quando la funzione deve essere rappresentata come testo
    - Ritorna una rappresentazione a stringa dell'oggetto funzione
  - `valueOf` - ritorna la funzione stessa come oggetto
  - `call` e `apply` – funzioni applicate all'oggetto passato come primo argomento, fornendo a tale oggetto i restanti parametri
    - Formato parametri differenzia `call` e `apply`
      - `funz.apply(ogg, arrayDiParametri )`  
equivale concettualmente a `ogg.funz(parametri)`
      - `funz.call(ogg, arg1, arg2, ... )`  
equivale concettualmente a `ogg.funz(arg1, arg2,..)`

# Call/apply: Esempio 1

- Definizione dell'oggetto funzione:
  - somma = function(x, y){ return x + y }
- Invocazione somma(5,7):
  - somma.apply(obj, [5,7] )
  - somma.call(obj, 5, 7 )
  - Se somma fosse una funzione senza parametri avrebbero lo stesso formato
  - Obj è irrilevante perché la funzione somma non fa riferimento a this nel suo corpo

# Call/apply: Esempio 2

- Funzione che fa riferimento a `this`:
  - `somma = function(y){ return y + this.x }`
- Oggetto destinatario del messaggio è rilevante
  - Determina l'environment di valutazione di `x`

1° caso

```
x = 10
somma.call(this,3)
```

Restituisce  $3 + 10 = 13$

2°caso

```
x = 10
function Obj(u) {
 this.x = u
}
obj = new Obj(7)
prova.call(obj,10)
```

Restituisce  $10 + 7 = 17$

# Oggetto globale

- PROBLEMA: come può JavaScript distinguere fra metodi di oggetti e funzioni "globali"?
- Non distingue: le funzioni "globali" non sono altro che metodi di un "oggetto globale" definito dal sistema.
- L'oggetto "globale" ha
  - Come metodi, le funzioni non attribuite a uno specifico oggetto nonché quelle globali (ad es., eval, parseInt)
  - Come dati, le variabili globali incluse quelle predefinite (ad es., undefined, NaN)
  - Oggetti predefiniti

# Funzioni globali predefinite

- Funzioni globali
  - eval valuta il programma JavaScript passato come stringa (riflessione, intercessione)
  - escape converte una stringa nel formato portabile: i caratteri non consentiti sono sostituiti da "sequenze di escape" (es. %20 per ' )
  - unescape riporta una stringa da formato portabile a formato originale
  - isFinite, isNaN, parseFloat, parseInt
  - ...

# [Costruttori di] Oggetti predefiniti

- Oggetti di uso generale
  - Array, Boolean, Function, Number, Object, String
  - Oggetto Math contiene la libreria matematica: costanti (E, PI, LN10, LN2, LOG10E, LOG2E, SQRT1\_2, SQRT2) e funzioni di ogni tipo
    - Non va istanziato ma usato come componente "statico"
  - Oggetto Date definisce i concetti per esprimere date e orari e lavorare su essi
    - Va istanziato nei modi opportuni
  - Oggetto RegExp fornisce il supporto per le espressioni regolari.
- Oggetti di uso grafico
  - Anchor, Applet, Area, Button, Checkbox, Document, Event, FileUpload, Form, Frame, Hidden, History, Image, Layer, Link, Location, Navigator, Option, Password, Radio, Reset, Screen, Select, Submit, Text, Textarea, Window

# Oggetto globale: chi è

- L' oggetto "globale" è UNICO e viene sempre creato dall'interprete prima di eseguire alcunché
- Però non esiste un identificatore "global": in ogni situazione c'è un dato oggetto usato come globale
- In un browser Web, l'oggetto globale solitamente coincide con l'oggetto window
  - Ma non è sempre così: a lato server, per esempio, sarà probabilmente l'oggetto response a svolgere quel ruolo!

# Funzioni anonime

- Le funzioni anonime sono quelle dichiarate dinamicamente a runtime
- Non prendono nome nel solito modo.
  - Dichiarazione
    - `function flyToTheMoon(){`
    - `alert("Zoom! Zoom! Zoom!");`
    - `}`
  - Operatore `function` (chiamato runtime)
    - `var flyToTheMoon = function(){`
    - `alert("Zoom! Zoom! Zoom");`
    - `}`

# Funzioni anonymous

- Il nome della funzione (non della variabile di tipo funzione) serve per
  - Chiamate riconoscitive
  - Debug visto che ne resta traccia nello stack

# Scoping e hoisting

- Lo scopo non viene delimitato da blocchi quali if while ecc.
  - Non esiste il **block-level scope**
- L'unico delimitatore di scopo è la funzione
  - Si parla di **function-level scope**
- Hoisting: la buona pratica di dichiarare tutte le variabili all'inizio del loro scopo
  - Per evitare errori dovuti a variabili valorizzate prima della dichiarazione
  - Questo movimento è come se venisse fatto in modo invisibile dall'interprete Javascript.
    - Hoisted solo la dichiarazione non eventuali valorizzazioni o assegnamenti

# Function binding

- Problema: Come mantenere il contesto all'interno di un'altra funzione
  - Uno potrebbe essere tentato di settare una variabile che riferisca il contest
    - Funziona ma non è elegante

# Esempio(1)

```
var myObj = {
 specialFunction: function () {},
 anotherSpecialFunction: function () { },
 getAsyncData: function (cb) {cb(); },
 render: function () {
 var that = this;
 this.getAsyncData(function () {
 that.specialFunction();
 that.anotherSpecialFunction();
 });
 }
};

myObj.render();
```

# Esempio(2)

- Se avessi lasciato `this` (`this.specialFunction()`) avrei ottenuto un errore
  - `Uncaught TypeError: Object [object global] has no method 'specialFunction'`
- `var that = this;` ci permette di mantenere il contest
- Posso risolvere in modo più formale con `Function.prototype.bind()`

```
render:function () {
 this.getAsyncData(function () {
 this.specialFunction();
 this.anotherSpecialFunction();
 }.bind(this));
}
```

# *Function.prototype.bind()*

- Crea una funzione che quando chiamata ha il suo `this` settato al valore passato.
  - Nell'esempio passando `this` permetto alla funzione di callback di eseguire con `this` impostato correttamente

```
Function.prototype.bind = function (scope) {
 var fn = this;
 return function () {
 return fn.apply(scope);
 };
}
```

# Esempio

```
var foo = {
 x: 3
}
```

```
var bar = function(){
 console.log(this.x);
}
```

```
bar(); // undefined
var boundFunc = bar.bind(foo);
boundFunc(); // 3
```

# Supporto al bind()

- Purtroppo Function.prototype.bind non è ben supportata dai browser meno recenti
- Essitono delle soluzioni alternative che includono la definizione di bind da includere.

# Closure function

- **Una chiusura (closure) è:** *una astrazione che combina una funzione con le variabili libere presenti nell'ambiente in cui è definita secondo le regole di scope del linguaggio*
- Ricorda l'environment dove è stata create
  - Utile per emulare metodi private
- le funzioni interne hanno accesso alle variabili dichiarate nei relative outer scope.
  - `function init() {  
 var name = "Mozilla"; // name is a local variable created by init  
 function displayName() { // displayName() is the inner function available only  
 within the body of init  
 alert(name); // use variable declared in the parent function  
 }  
 displayName();  
}  
init();`

# Esempio

```
function makeFunc() {
 var name = "Mozilla";
 function displayName() {
 alert(name);
 }
 return displayName;
}
var myFunc = makeFunc();
myFunc();
```

- Ritorna la stessa cosa ma:
  - La inner function displayName è ritornata dalla outer function prima che venga eseguita

# Closure

- MyFunc è diventata una closure
  - Oggetto che combina funzione e ambiente in cui è stata eseguita
    - L'ambiente è costituito da ogni cosa che è nello scopo al tempo della creazione della closure
- In Javascript l'uso di function dentro un'altra funzione crea una closure.
  - Un riferimento ad una funzione ha anche un riferimento segreto alla sua closure (scopo)
    - Resta uno stack frame in memoria anche dopo l'usicta dalla funzione outer

# Esempio

- var gLogNumber, gIncreaseNumber, gSetNumber;
- function setupSomeGlobals() {
  - // Local variable that ends up within closure
  - var num = 42;
  - // Store some references to functions as global variables
  - gLogNumber = function() { console.log(num); }
  - gIncreaseNumber = function() { num++; }
  - gSetNumber = function(x) { num = x; }
  - }
- setupSomeGlobals();
- gIncreaseNumber();
- gLogNumber(); // 43
- gSetNumber(5);
- gLogNumber(); // 5
- var oldLog = gLogNumber;
- setupSomeGlobals();
- gLogNumber(); // 42
- oldLog() // 5

# Closure

- In JavaScript quando si dichiara una funzione dentro un'altra, le funzioni interne sono recreate ogni volta che la funzione esterna viene chiamata

# Esempio

- function buildList(list) {
- var result = [];
- for(var i = 0; i < list.length; i++) {
- var item = 'item' + i;
- result.push( function() {console.log(item + '' + list[i])} );
- }
- return result;
- }
- 
- function testList() {
- var fnlist = buildList([1,2,3]);
- // Using j only to help prevent confusion -- could use i.
- for(var j = 0; j < fnlist.length; j++) {
- fnlist[j]();
- }
- }
- 
- testList() //logs "item2 undefined" 3 times

# Esempio

- Esiste solo una closure per le variabili locali di buildList
- Quando le funzioni anonime sono chiamate alla riga `fnlist[j]();` usano tutte l'unica closure
  - Dove i ha valore 3 perchè ha finito
  - item="item2" perchè si parte da item0
  - Le chiamate alla funzione anonima avvengono considerando l'unico scopo commune avente i=3 e item = item2

# Closure + hoisting

- function sayAlice() {
  - var say = function() { console.log(alice); }
  - // Local variable that ends up within closure
  - var alice = 'Hello Alice';
  - return say;
  - }
  - sayAlice()();// logs "Hello Alice"
- sayAlice()(); chiama direttamente la function reference

# Esempio

- Ogni call crea una chiusura separata per le variabili locali

```
• function newClosure(someNum, someRef) {
• var num = someNum;
• var anArray = [1,2,3];
• var ref = someRef;
• return function(x) {
• num += x;
• anArray.push(num);
• console.log('num: ' + num + '\nanArray' + anArray.toString() + '\nref.someVar ' + ref.someVar);
• }
• }
• obj = {someVar: 4};
• fn1 = newClosure(4, obj);
• fn2 = newClosure(5, obj);
• fn1(1); // num: 5; anArray: 1,2,3,5; ref.someVar: 4;
• fn2(1); // num: 6; anArray: 1,2,3,6; ref.someVar: 4;
• obj.someVar++;
• fn1(2); // num: 7; anArray: 1,2,3,5,7; ref.someVar: 5;
• fn2(2); // num: 8; anArray: 1,2,3,6,8; ref.someVar: 5;
```

# Effetti della closure: pattern of public, private, and privileged members

- Oggetti:
  - Javascript è fondamentalmente basato su Oggetti, array funzioni... tutti sono oggetti
    - Oggetto=collezione di nome-valore
      - Nome= stringa
      - Valore= stringa, numero oggetto funzione array ecc.
    - Implementati come hashtable
- Quando un valore è una funzione allora si parla di metodo
  - Quando invoco un metodo di un oggetto allora this riferisce l'oggetto
    - Il metodo può accedere alle variabili d'istanza attraverso il this
- Costruttori: funzioni che inizializzano l'oggetto

## 2. RESTful

Making APIs

# REpresentational State Transfer (2007)

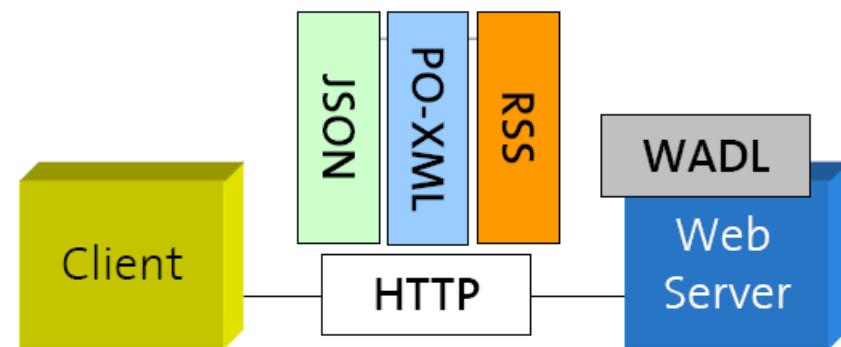
Idealizzato per primo da Roy Fielding:

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

E' un pattern architetturale basato sulle operazioni fornite dal protocollo HTTP e sulle risorse rappresentate come URI.

Eredita le caratteristiche dell'HTTP:

- Null style
- Client-server
- Stateless
- Cache
- Uniform interface
- Layered system
- Code-on-demand



# Elementi architetturali: ROA

- Resource-Oriented Architecture
  - REST è un'astrazione di elementi architetturali in un sistema hypermedia distribuito
  - REST ignora i dettagli implementativi dei componenti e la sintassi dei protocolli
  - REST si focalizza sul ruolo dei componenti, vincoli per interazione con altri componenti e l'interpretazione dei dati
- RESTful Web Service
  - È una configurazione di URI, HTTP, XML/JSON che si comporta come il web

# Resource

- Resource
  - Ogni informazione che può avere un nome (ad es., documento, immagine, persone, ...)
  - Ogni concetto che può essere target di riferimento ipertestuale
- Ogni resource è una membership function che al tempo t viene associata a un set di entità/valori
  - Valori sono la rappresentazione della risorse e/o gli identificativi
  - Risorse sono vuote, statiche, dinamiche
  - L'unica cosa sempre statica è la semantica della risorsa
  - Il mio libro preferito (risorsa dinamica)
  - Il libro XYZ (risorsa statica)

# Resource identifier

- Resource identifier
  - L'autorità di naming deve mantenere la validità semantica del mapping tra identificatore e risorsa
  - L'autore sceglie l'identificatore
  - URI/URN
- Due URI diverse possono puntare allo stessa risorsa, una stessa URI può ritornare informazioni su più risorse

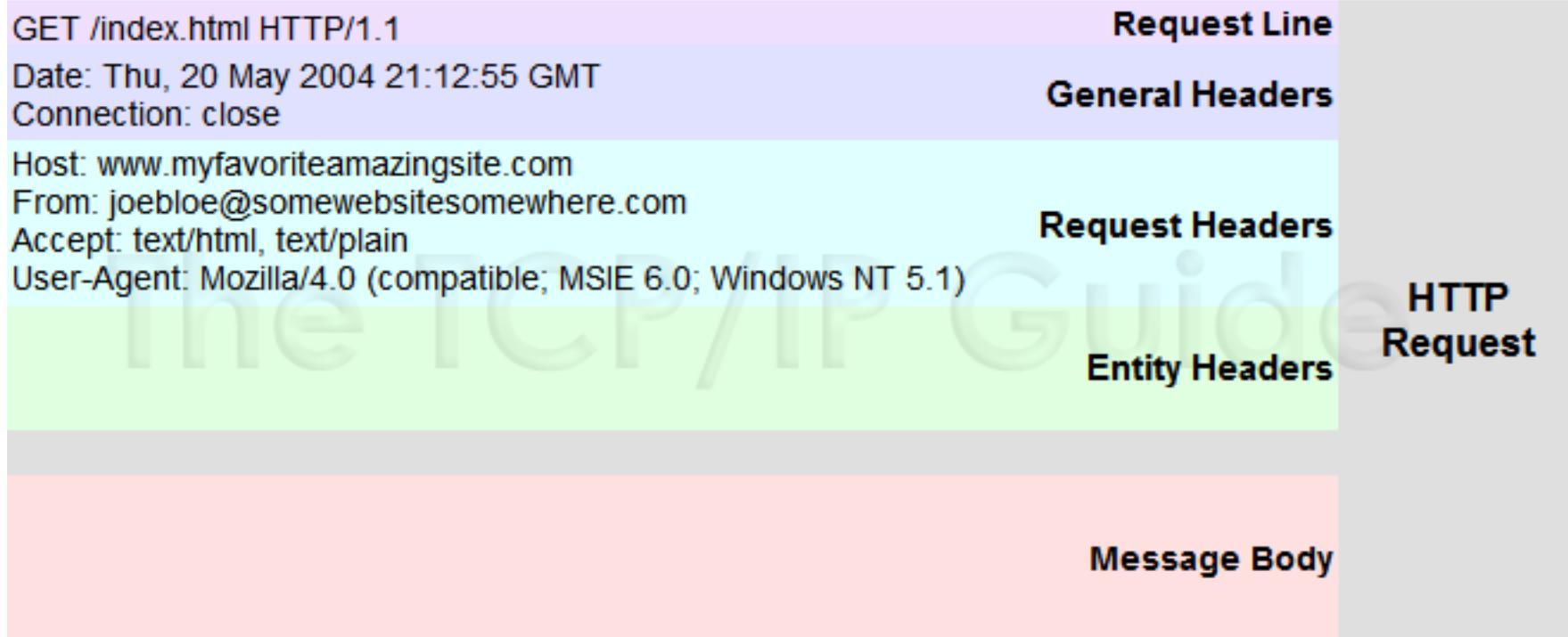
# Addressability

- Un'applicazione è indirizzabile se
  - Espone i suoi dati come risorse
  - Espone un URI per ogni informazione di interesse
  - Il file system è indirizzabile (ogni file ha un percorso univoco)
- Uno dei requisiti fondamentali per gli end user
- Addressability permette di supportare bookmarking, linking, emailing...

# Statelessness

- Ogni richiesta HTTP è eseguita in isolation
- Il server non usa informazioni da richieste precedenti, il client inserisce tutto ciò che è necessario a soddisfare la richiesta
- Consideriamo statelessness in funzione di addressability
  - Statelessness definisce i possibili stati di un server come risorse
- Nonostante il principio di statelessness, spesso il web è implementato stateful

# Http Request



Request Header    -> Importante per specificare molte informazioni (tipo di dato, autenticazione, etc)

Body                -> Il contenuto del messaggio/ per le request solo POST/PUT

# Http Response

HTTP/1.1 200 OK	<b>Status Line</b>	
Date: Thu, 20 May 2004 21:12:58 GMT	<b>General Headers</b>	
Connection: close		
Server: Apache/1.3.27	<b>Response Headers</b>	
Accept-Ranges: bytes		
Content-Type: text/html	<b>Entity Headers</b>	
Content-Length: 170		
Last-Modified: Tue, 18 May 2004 10:14:49 GMT		
<html>		<b>HTTP Response</b>
<head>		
<title>Welcome to the Amazing Site!</title>		
</head>		
<body>	<b>Message Body</b>	
<p>This site is under construction. Please come back later. Sorry!</p>		
</body>		
</html>		

# REpresentational State Transfer (Server)

- REST accede ai dati usando URI
  - <http://www.claudioardagna.com/resource/travels>
  - <http://www.claudioardagna.com/resource/travels/newyork>
- I suoi quattro principi mostrano il successo e la scalabilità del protocollo HTTP
  - Resource Identification attraverso URI
  - Uniform Interface per tutte le risorse
    - **GET (Query the state, idempotent, can be cached): ottiene una rappresentazione della risorsa**
    - **POST (Create a child resource): crea una nuova risorsa**
    - **PUT (Update, transfer a new state): crea o aggiorna una risorsa**
    - **DELETE (Delete a resource): cancella una risorsa**
  - “Self-Describing” Message attraverso metadati e rappresentazione di risorse multiple
  - Hyperlink per definire le transizioni di stato dell’applicazione e le relazioni tra risorse

# REpresentational State Transfer (Server)

<b>GET</b>	(Query the state, idempotent, can be cached): ottiene una rappresentazione della risorsa)
<b>POST</b>	(Create a child resource): crea una nuova risorsa
<b>PUT</b>	(Update, transfer a new state): crea o aggiorna una risorsa
<b>DELETE</b>	(Delete a resource): cancella una risorsa

# Consumare Servizi RESTful

Come posso consumare servizi REST?

- solitamente li consumiamo tramite il web browser ma non ce ne accorgiamo (web user)
- costruendo un client (web developer)
- con client rest come Postman (web developer che devono fare debug)



<https://www.getpostman.com/>

# 3. Loopback

Meta-data backend

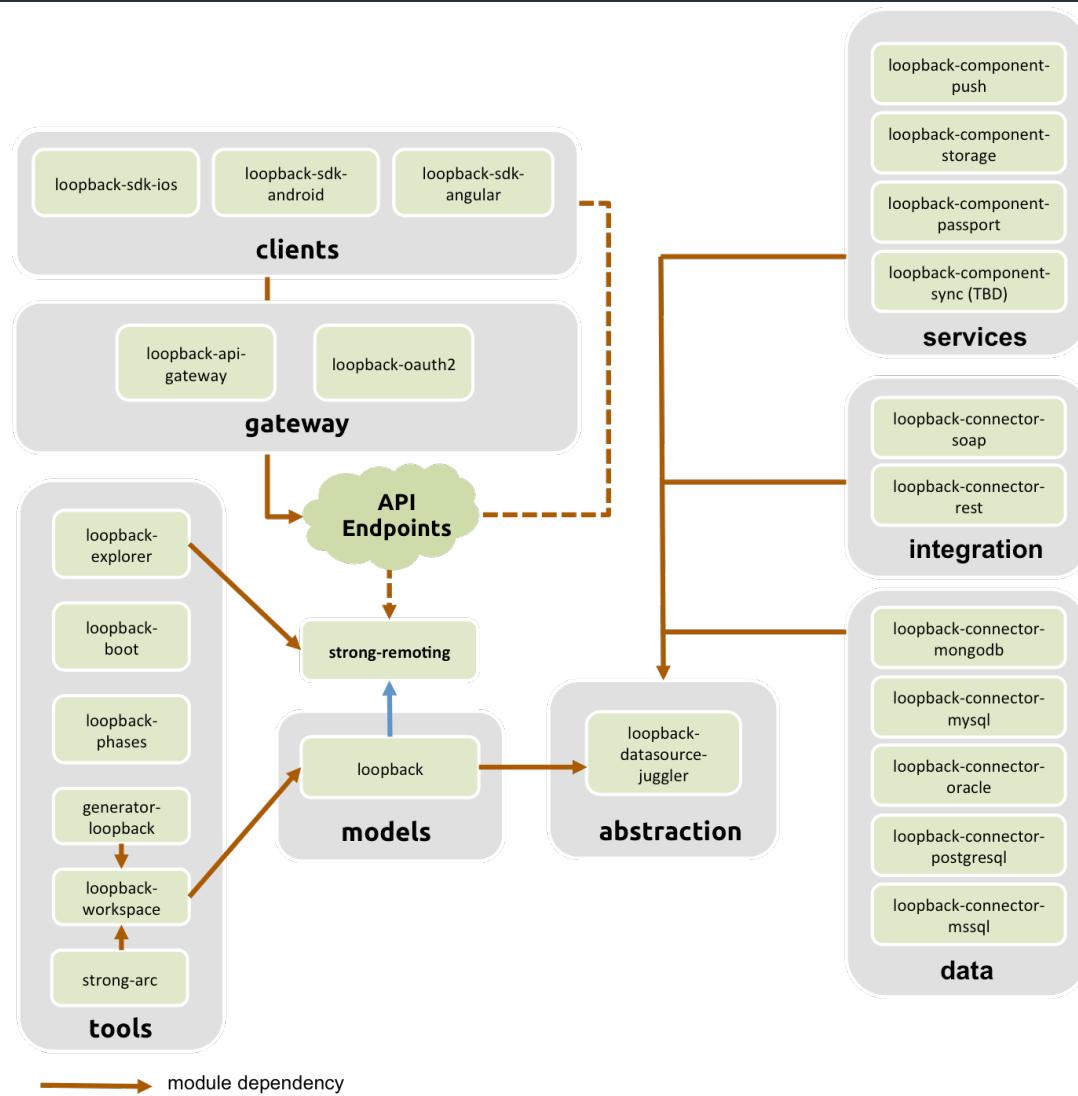


# Perchè Loopback.io?

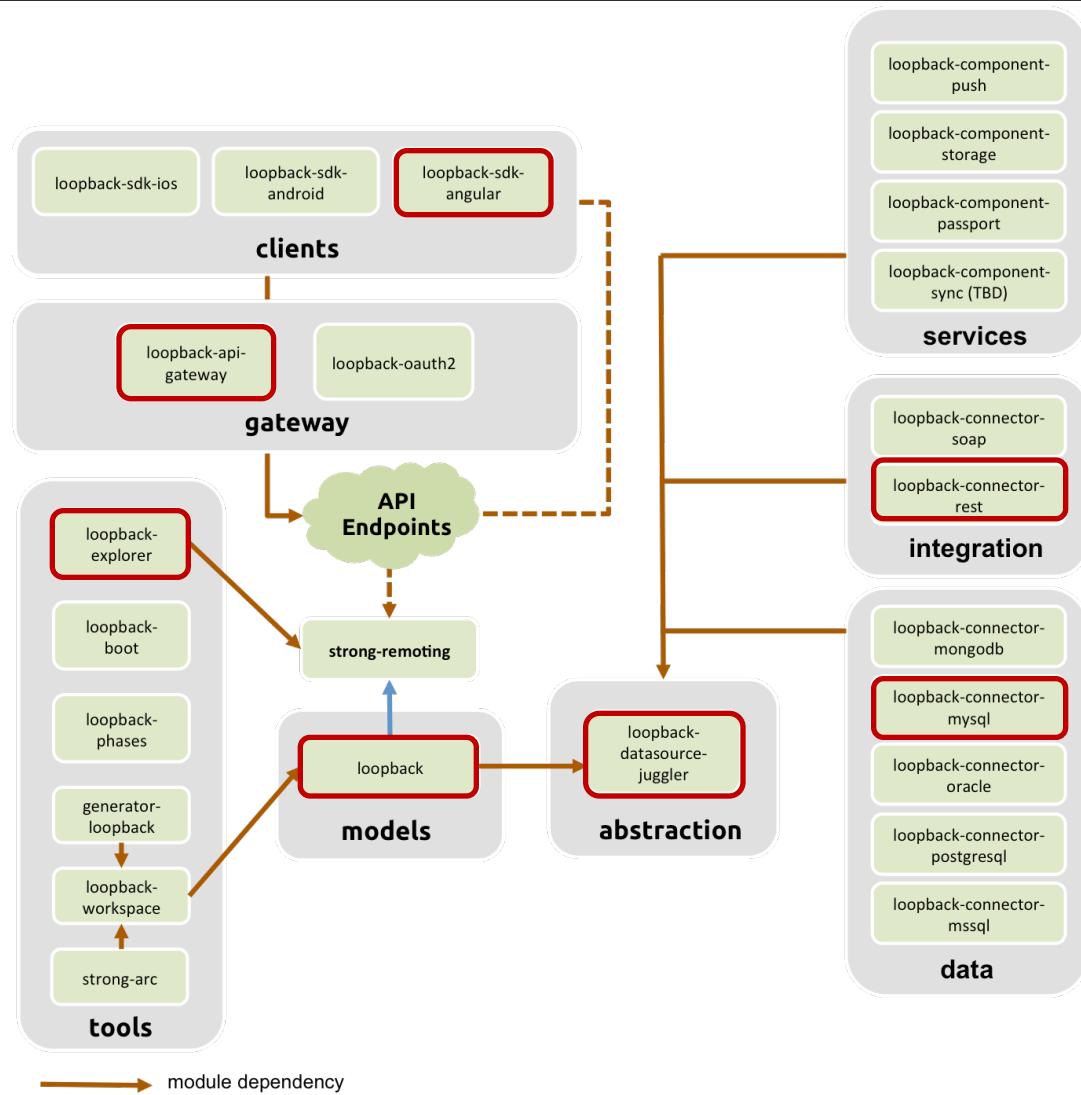
- Perchè è basato su nodejs e quindi su javascript (stesso linguaggio per frontend e backend)
- Per massimizzare la produttività, lasciando il coding da scimmie al framework

LoopBack is a highly-extensible, open-source Node.js framework that enables you to create dynamic end-to-end REST APIs **with little or no coding.**
- Ha una creazione guidata per i diversi componenti ed è tutto basato/configurato su meta-dati

# Loopback.io



# Loopback.io



# Componenti

Category	Description	Use to...	Modules
Models	Model and API server	Dynamically mock-up models and expose them as APIs without worrying about persisting.	loopback
Abstraction	Model data abstraction to physical persistence	Connect to multiple data sources or services and get back an abstracted model with CRUD capabilities independent of backing data source.	loopback-datasource-juggler
Initialization	Application initialization	Configure data sources, customize models, configure models and attach them to data sources; Configure application settings and run custom boot scripts.	loopback-boot
Sequencing	Middleware execution	Configure middleware to be executed at various points during application lifecycle.	loopback-phase
Data	RDBMS and noSQL physical data sources	Enable connections to RDBMS and noSQL data sources and get back an abstracted model.	<ul style="list-style-type: none"> <li>• loopback-connector-mongodb</li> <li>• loopback-connector-mysql</li> <li>• loopback-connector-postgresql</li> <li>• loopback-connector-mssql</li> <li>• loopback-connector-oracle</li> <li>• <a href="#">Many others...</a></li> </ul>
Integration	General system connectors	Connect to an existing system that expose APIs through common enterprise and web interfaces	<ul style="list-style-type: none"> <li>• loopback-connector-rest</li> <li>• loopback-connector-soap</li> </ul>
Components	Add-ons to core LoopBack	Integrate with pre-built services packaged into components.	<ul style="list-style-type: none"> <li>• loopback-component-push</li> <li>• loopback-component-storage</li> <li>• loopback-component-passport</li> </ul>
Clients	Client SDKs	Develop client app using native platform objects (iOS, Android, AngularJS) that interact with LoopBack APIs via REST.	<ul style="list-style-type: none"> <li>• loopback-sdk-ios</li> <li>• loopback-sdk-android</li> <li>• loopback-sdk-angular</li> </ul>

# Installazione e Requisiti

- il terminale...questo sconosciuto...

```
TERM=xterm /etc/init.d/telnetd start
$ telnet localhost 23
Connected to localhost.
Escape character is '^]'.
$ ls
bash examples
$ cd bash
$./examples/startup-files
$ cat startup-files
#!/bin/sh
This file is executed by /bin/sh(1) when logon shell
or /bin/sh(1) is run as a command.
$HOME/share/doc/bash/examples/startup-files (in the package bash-doc)
or examples

If HOSTNAME is set but you can change it here
export HOSTNAME=thebeast

Set up some environment variables
export IDESYSTEM=$HOME/system

term is the default TERM
export TERM=xterm
export TERMINFO=$IDESYSTEM/etc/terminfo

export TEMP=$HOME/tmp
export TMPDIR=$TEMP
export ODEX_FOLDER=$TEMP

export SHELL=$IDESYSTEM/bin/bash

export MC_DATADIR=$IDESYSTEM/etc/mc
export VIMRUNTIME=$IDESYSTEM/etc/vim

export EDITOR=vim

Check if we are running over telnet or ssh
This variable is set in the telnetd script
"TELNET_ON" = yes]

Change a few things
You may need to set a different TERM value
export TERM=xterm

Need to resize the screen
resize

A needs a few tweaks
export BOOTCLASSPATH=/system/classes/android.jar:$BOOTCLASSPATH

Default values
export TELNET_PORT=8080

Set the Lib Path
export LD_LIBRARY_PATH=$HOME/lib:$HOME/system/lib:$LD_LIBRARY_PATH

Set the PATH
export PATH=/HOME/bin:$HOME/bin/bbdir:$HOME/system/bin:$HOME/system/bin/bbdir:$PATH

Go to HOME directory
cd ~

91L, 2253C
:hellothere*
```

PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
9807	1	10087	S	1784	0.3	0	1.8	tmux
9863	9810	10087	R	2092	0.4	0	1.1	top
485	85	10000	S	310m	69.9	0	0.9	system server
16	2	0	S	0	0.0	0	0.3	[kondemand/0]
96	1	2000	S	3420	0.7	0	0.1	/sbin/adb
9098	9097	10087	S	600	0.1	0	0.1	telnetd -p 8080 -l /data/data/com.spartacusrex.spartacusrex.spartacusrex
43	2	0	S	0	0.0	0	0.1	[file-storage]
10	2	0	S	0	0.0	0	0.1	[sync_supers]
609	85	10044	S	243m	54.7	0	0.0	{e.process.gapps} com.google.process.gapps
6855	85	10087	S	160m	36.0	0	0.0	{ex.spartacuside} com.spartacusrex.spartacuside
7249	85	10061	S	159m	35.8	0	0.0	{locationService} com.google.android.apps.maps:Network
6866	85	10061	S	159m	35.8	0	0.0	{droid.apps.maps} com.google.android.apps.maps
558	85	1001	S	155m	35.0	0	0.0	{m.android.phone} com.android.phone
566	85	10006	S	155m	34.9	0	0.0	{app.twlauncher} com.sec.android.app.twlauncher
545	85	10008	S	151m	34.1	0	0.0	{android.systemui} com.android.systemui
6936	85	10042	S	151m	34.1	0	0.0	{alphonso.pulse} com.alphonso.pulse
6982	85	10088	S	145m	32.8	0	0.0	{ogle.android.gm} com.google.android.gm
672	85	10006	S	143m	32.3	0	0.0	{d.process.acore} android.process.acore
9090	85	10000	S	140m	31.6	0	0.0	{ndroid.settings} com.android.settings
2347	85	10003	S	140m	31.5	0	0.0	{android.vending} com.android.vending
7219	85	10008	S	139m	31.4	0	0.0	com.wssyncmlnd
559	85	10004	S	139m	31.3	0	0.0	{d.process.media} android.process.media
5381	85	10021	S	138m	31.1	0	0.0	{equicksearchbox} com.google.android.googlequicksearchbox
7197	85	10077	S	137m	31.0	0	0.0	{cooliris.media} com.cooliris.media
2598	85	10023	S	137m	30.9	0	0.0	{viders.calendar} com.android.providers.calendar
6841	85	10008	S	137m	30.8	0	0.0	{d.providers drm} com.sec.android.providers.drm
6920	85	10065	S	136m	30.8	0	0.0	{cwweatherwidget} com.sec.android.clockweatherwidget
6928	85	1001	S	136m	30.7	0	0.0	{osp.app.signin} com.osp.app.signin
1247	85	10011	S	136m	30.7	0	0.0	{samsungapps.una} com.sec.android.app.samsungapps.una
1067	85	10000	S	136m	30.6	0	0.0	{MtpApplication} com.android.MtpApplication
560	85	10002	S	136m	30.6	0	0.0	{mb.ap system} com.broadcom.bt.ap.system
5283	85	10026	S	135m	30.6	0	0.0	com.svox.pico
85	1	0	S	126m	28.5	0	0.0	zygote /bin/app process -Xzygote /system/bin --zygote
86	1	1013	S <	40396	8.8	0	0.0	/system/bin/mediaserver
92	1	1021	S	16552	3.6	0	0.0	/system/bin/gpsd -c /system/etc/gps.xml

bin	projects	system	tmux-client-9519.log
lib	sdcard	tmp	tmux-client-9523.log
terminal+@10.0.0.4-\$ 11			
drwx-----	1 10087 10087	0 Dec 1 11:12 bin	
drwx-----	1 10087 10087	0 Nov 24 15:05 lib	
drwx-----	1 10087 10087	0 Nov 24 15:05 projects	
lrwxrwxrwx	1 10087 10087	40 Nov 24 15:05 sdcard -> /mnt/sdcard	
drwxr-x---	1 10087 10087	0 Nov 24 15:05 system	
drwx-----	1 10087 10087	0 Dec 1 11:24 tmp	
-rwx-rw-rw-	1 10087 10087	26 Dec 1 13:34 tmux-client-9519.log	
		26 Dec 1 13:35 tmux-client-9523.log	
		15:12:16 Thu 12/0	

# Installazione e Requisiti

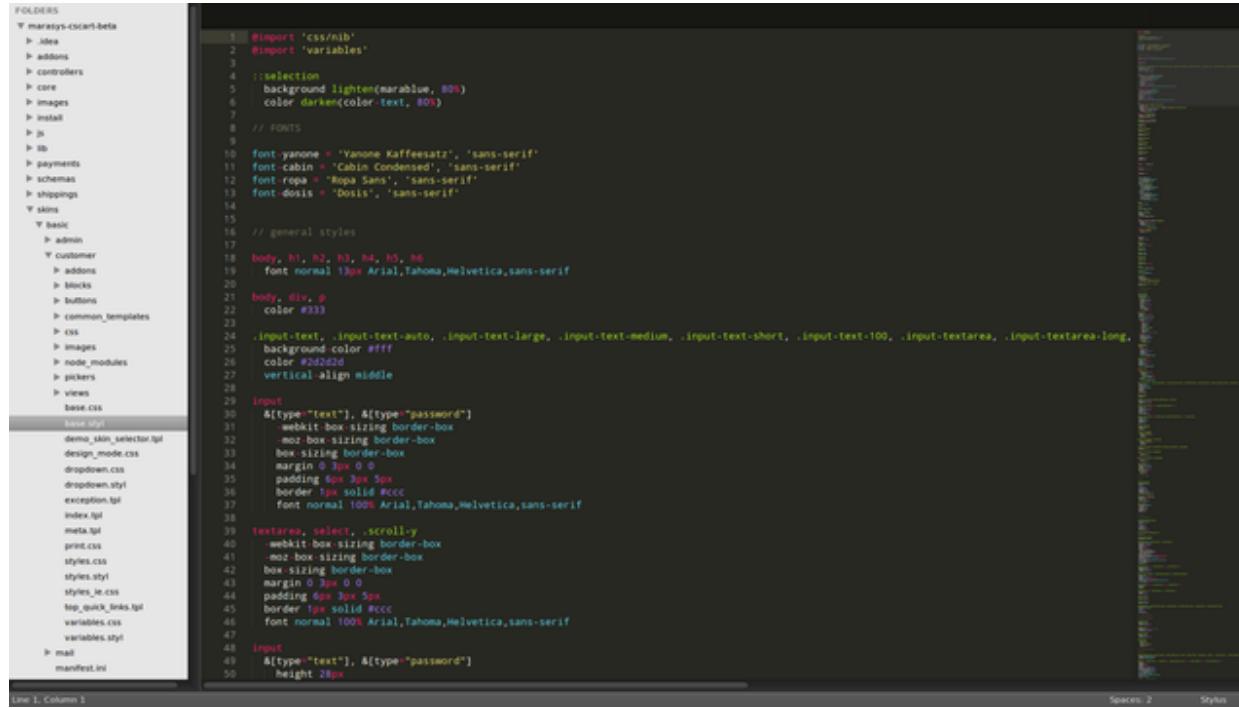
- **GIT**–is a version control system (VCS). Lo utilizzeremo per scaricare tutto il materiale che ho preparato per voi e che è su github



```
$ git clone
```

# Installazione e Requisiti

- **Sublime** – è un comodo editor di testo che utilizzeremo per scrivere il codice del nostro progetto.



The screenshot shows the Sublime Text code editor interface. On the left, there's a sidebar displaying a file tree for a project named "maratus-cscart-beta". The tree includes folders like "admin", "customer", "css", "images", and "views", along with various files such as "base.css", "index.tpl", "meta.tpl", "print.css", "variables.css", and "variables.styl". The main pane of the editor displays a portion of a CSS file with syntax highlighting. The code includes imports for "css/nib" and "variables", and defines styles for "body", "h1" through "h6", and various input types like "text", "password", "checkbox", and "radio". It also includes media queries for "min-width: 768px" and "min-width: 992px". The right side of the interface features a vertical sidebar with icons for file operations like "New File", "Save", and "Close". At the bottom, there are status bars showing "Line 1, Column 3" and "Spaces: 2 Styles".



# Installazione e Requisiti

- **NPM** - npm is the package manager for JavaScript. Find, share, and reuse packages of code from hundreds of thousands of developers and assemble them in powerful new ways.



# Installazione e Requisiti

- **NODEJS STABLE** – per evitare problemi re-installiamo nodejs nella sua versione stabile.

```
$ sudo npm cache clean -f
$ sudo npm install -g n
$ sudo n stable
```

# Installazione e Requisiti

- **LOOPBACK** – ora possiamo lanciare l'installazione di loopback con tutte le sue dipendenze

```
$ sudo npm install -g --unsafe-perm install strongloop
```



# TAKE A BREAK, USE YOUR BRAIN

- Ragioniamo su come modellare la nostra agenda  
TODOs

# TAKE A BREAK, USE YOUR BRAIN

- Ragioniamo su come modellare la nostra agenda  
TODOs

campo	tipo
name	string
description	string
dueto	date
place	string
done	boolean
important	boolean

# TODOS – loopback project

- Creiamo il nostro progetto loopback

\$ slc loopback



da adesso in poi lavoriamo sui nostri computer

# Looback - explorer

http://localhost:3000

The StrongLoop API Explorer interface displays a list of API endpoints for the 'loopback-getting-started' application. The interface includes a header with the logo, title, token status, and a 'Set Access Token' button. Below the header, the main content area shows two sections: 'CoffeeShop' and 'User'. Each section lists various HTTP methods (GET, POST, PUT, DELETE, HEAD) and their corresponding URLs and descriptions.

CoffeeShop		Operations
GET	/CoffeeShops	Find all instances of the model matched by filter from the data source.
PUT	/CoffeeShops	Update an existing model instance or insert a new one into the data source.
POST	/CoffeeShops	Create a new instance of the model and persist it into the data source.
GET	/CoffeeShops/{id}	Find a model instance by id from the data source.
HEAD	/CoffeeShops/{id}	Check whether a model instance exists in the data source.
PUT	/CoffeeShops/{id}	Update attributes for a model instance and persist it into the data source.
DELETE	/CoffeeShops/{id}	Delete a model instance by id from the data source.
GET	/CoffeeShops/{id}exists	Check whether a model instance exists in the data source.
GET	/CoffeeShops/change-stream	Create a change stream.
POST	/CoffeeShops/change-stream	Create a change stream.
GET	/CoffeeShops/count	Count instances of the model matched by where from the data source.
GET	/CoffeeShops/findOne	Find first instance of the model matched by filter from the data source.
POST	/CoffeeShops/update	Update instances of the model matched by where from the data source.

User		Operations
------	--	------------

[ BASE URL: /api , API VERSION: 1.0.0 ]

# TODOS – loopback project

- Selezioniamo il datasource che vogliamo utilizzare

```
$ slc loopback:datasource
```

```
Enter the data-source name: mysql
Select the connector for mysql: MySQL (supported by StrongLoop)
Connector-specific configuration:?
Connection String url to override other settings (eg: mysql://user:pass@host/db):
mysql://root:@localhost/todos
```

```
$sudo systemctl start mariadb
$ mysql -u root
create database todos;
exit;
```

# TODOS – loopback project

- Creiamo il modello

\$ slc loopback:model



da adesso in poi lavoriamo sui nostri computer

# MAKE IT WORK

```
$ sudo npm cache clean -f
$ sudo npm install -g n
$ sudo n stable
$ sudo npm install -g --unsafe-perm install strongloop
$ sudo systemctl start mariadb
$ mysql -u root
$ create database todos;
$ exit
da dentro cartella progetto
$ npm install
$ node .
```

<https://github.com/anonymez>

```
$ git clone <your repository>
```

# RECAP

## RESTful Web Service

È una configurazione di URI, HTTP, XML/JSON che si comporta come il web

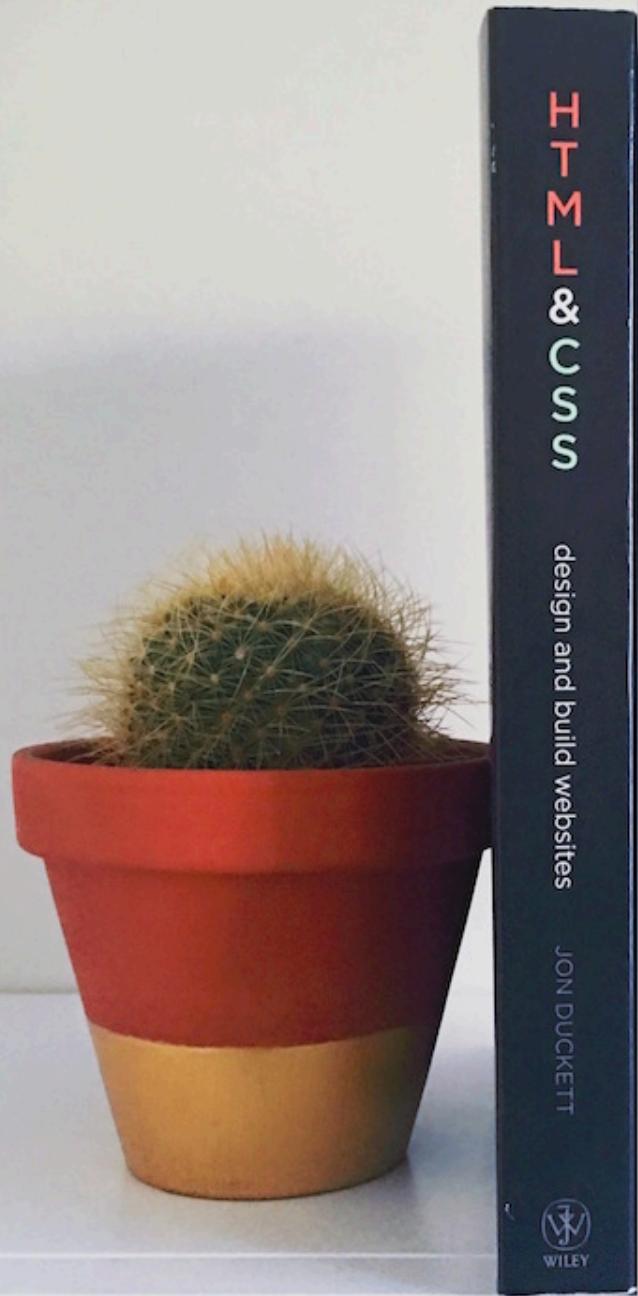
<b>GET</b>	(Query the state, idempotent, can be cached): ottiene una rappresentazione della risorsa)
<b>POST</b>	(Create a child resource): crea una nuova risorsa
<b>PUT</b>	(Update, transfer a new state): crea o aggiorna una risorsa
<b>DELETE</b>	(Delete a resource): cancella una risorsa

# RECAP

## todos

Show/Hide | List Operations | Expand Operations

PATCH	/todos	Patch an existing model instance or insert a new one into the data source.
GET	/todos	Find all instances of the model matched by filter from the data source.
PUT	/todos	Patch an existing model instance or insert a new one into the data source.
POST	/todos	Create a new instance of the model and persist it into the data source.
PATCH	/todos/{id}	Patch attributes for a model instance and persist it into the data source.
GET	/todos/{id}	Find a model instance by {{id}} from the data source.
HEAD	/todos/{id}	Check whether a model instance exists in the data source.
PUT	/todos/{id}	Patch attributes for a model instance and persist it into the data source.
DELETE	/todos/{id}	Delete a model instance by {{id}} from the data source.
GET	/todos/{id}/exists	Check whether a model instance exists in the data source.



# 5. HTML5 CSS3 Bootstrap3

MAKE PAGES

# Linguaggio di markup

- Linguaggio di markup modellato per rendere esplicita una particolare interpretazione di un testo
  - Per esempio, tutte quelle aggiunte al testo scritto che permettono di renderlo più fruibile
  - Oltre a rendere il testo più leggibile, il markup permette anche di specificare ulteriori usi del testo
- Con il markup per sistemi informatici, specifichiamo le modalità esatte di utilizzo del testo nel sistema stesso
- I linguaggi di markup sono i linguaggi più opportuni per strutturare e marcare i documenti in maniera indipendente dall'applicazione, favorendo la riusabilità, la flessibilità e la apertura ad applicazioni complesse

# HyperText Markup Language (HTML)

- HTML linguaggio per la creazione di pagine Web
  - Standard per la rappresentazione
  - Descrive la struttura della pagina
  - Non è un linguaggio di programmazione, è un linguaggio di markup
  - Descrive dati e regole su come mostrarli
  - Poche e semplici regole sintattiche

# Storia degli standard web

91-92	93-94	95-96	97-98	99-00	01-02	03-04	05-06	07-08	09-10	11-12	13-14
HTML 1	HTML 2	HTML 3	HTML 4	XHTML 1					HTML 5		
		CSS 1	CSS 2			Web 2.0			CSS3		
		JS	XML 1.0, DOM	DOM 2		XML 1.1	Ajax		DOM, APIs		

## HTML

1991	HTML 1 (HTML Tag)
1994	HTML 2 Draft
1995	HTML 3 Internet Draft
1997	HTML 3.2 W3C Rec
1997-98	HTML 4.0 W3C Proposed Rec
2008	HTML 5 W3C Working Draft
2012	HTML 5 W3C Candidate Rec
2014	HTML 5 W3C Rec

# HTML - Tag

- Tag sono marcatori che identificano porzioni di testo
- Permettono la personalizzazione della pagina, e di identificare e processare le porzioni di testo contrassegnate da una certa marcatura
- Nome tag = nome funzione
  - Tag contenuto tra parentesi triangolari
  - Tag di inizio, tag di fine
  - Contenuto del tag
  - <tag attributi>contenuto</tag>

# HTML5 Document

```
<!doctype html>
<html lang="it">
<head>
 <title>We5! La guida HTML5</title>
</head>
<body>
<h1>HELLO WORLD!</h1>
</body>
</html>
```

# CSS

- CSS (Cascading Style Sheet): fogli di stile a cascata
- Uno dei linguaggi fondamentali del W3C
- CSS parallelo a HTML
  - HTML specificato per definire il contenuto del sito...
  - ... non la sua formattazione
  - Quando con HTML 3.2 sono stati introdotti elementi tipo `<font>`, lo sviluppo di pagine web è diventato un incubo per gli sviluppatori
    - Formattazione e colori aggiunti ad ogni singola pagina

# CSS3 - stylesheet

Cascading Style Sheets (CSS) è un linguaggio per specificare come i documenti sono presentati all'utente.

Un *documento* è un insieme di informazioni strutturate utilizzando un *linguaggio a marcatori*.

```
h1
{
 color: red;
}

body
{
 background-color:white;
 margin-top:100px;
}
```

# BOOTSTRAP

Bootstrap is an HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.



Preprocessors  
Bootstrap ships with vanilla CSS, but its source code utilizes the two most popular CSS preprocessors, [Less](#) and [Sass](#). Quickly get started with precompiled CSS or build on the source.



One framework, every device. Bootstrap easily and efficiently scales your websites and applications with a single code base, from phones to tablets to desktops with CSS media queries.



Full of features  
With Bootstrap, you get extensive and beautiful documentation for common HTML elements, dozens of custom HTML and CSS components, and awesome jQuery plugins.

# LAYOUT

.col-md-1												
.col-md-8							.col-md-4					
.col-md-4				.col-md-4				.col-md-4				
.col-md-6						.col-md-6						

	Extra small devices Phones (<768px)	Small devices Tablets (≥768px)	Medium devices Desktops (≥992px)	Large devices Desktops (≥1200px)
<b>Grid behavior</b>	Horizontal at all times	Collapsed to start, horizontal above breakpoints		
<b>Container width</b>	None (auto)	750px	970px	1170px
<b>Class prefix</b>	.col-xs-	.col-sm-	.col-md-	.col-lg-
<b># of columns</b>	12			
<b>Column width</b>	Auto	~62px	~81px	~97px
<b>Gutter width</b>	30px (15px on each side of a column)			
<b>Nestable</b>	Yes			
<b>Offsets</b>	Yes			
<b>Column ordering</b>	Yes			

# Responsive Utility

	<b>Extra small devices</b> Phones (<768px)	<b>Small devices</b> Tablets ( $\geq 768\text{px}$ )	<b>Medium devices</b> Desktops ( $\geq 992\text{px}$ )	<b>Large devices</b> Desktops ( $\geq 1200\text{px}$ )
<code>.visible-xs-*</code>	Visible	Hidden	Hidden	Hidden
<code>.visible-sm-*</code>	Hidden	Visible	Hidden	Hidden
<code>.visible-md-*</code>	Hidden	Hidden	Visible	Hidden
<code>.visible-lg-*</code>	Hidden	Hidden	Hidden	Visible
<code>.hidden-xs</code>	Hidden	Visible	Visible	Visible
<code>.hidden-sm</code>	Visible	Hidden	Visible	Visible
<code>.hidden-md</code>	Visible	Visible	Hidden	Visible
<code>.hidden-lg</code>	Visible	Visible	Visible	Hidden

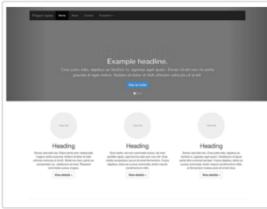
# Examples

<https://github.com/twbs/bootstrap/>



## Cover

A one-page template for building simple and beautiful home pages.



## Carousel

Customize the navbar and carousel, then add some new components.



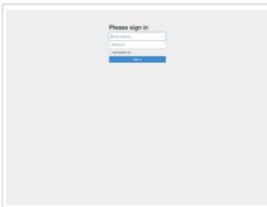
## Blog

Simple two-column blog layout with custom navigation, header, and type.



## Dashboard

Basic structure for an admin dashboard with fixed sidebar and navbar.



## Sign-in page

Custom form layout and design for a simple sign in form.



## Justified nav

Create a custom navbar with justified links. Heads up! [Not too Safari friendly](#).



## Sticky footer

Attach a footer to the bottom of the



## Sticky footer with navbar



<http://getbootstrap.com/getting-started/>

Let's see

<http://getbootstrap.com/examples/jumbotron/>

# Todo-list page

ADD TODOS

TODOS da fare

TODOS fatti



# 6. AngularJS

The client framework

# Cosa è ANGULARJS?

- Non è una libreria javascript.
- Non è un manipolatore del DOM come jQuery. (usa un subset di jquery: jqlite)
- Si concentra più sul lato HTML.
- MVC/MVVM design pattern



**AngularJS è un framework MVC javascript, sviluppato da Google, per sviluppare web application mantenibili e con un'architettura ben strutturata.**

# Filosofia

- “AngularJS è quello che HTML avrebbe dovuto essere se pensato per lo sviluppo di applicazioni web”.
- “AngularJS è stato creato con la filosofia che il codice dichiarativo è meglio di quello imperativo quando si sviluppa UI per web application”

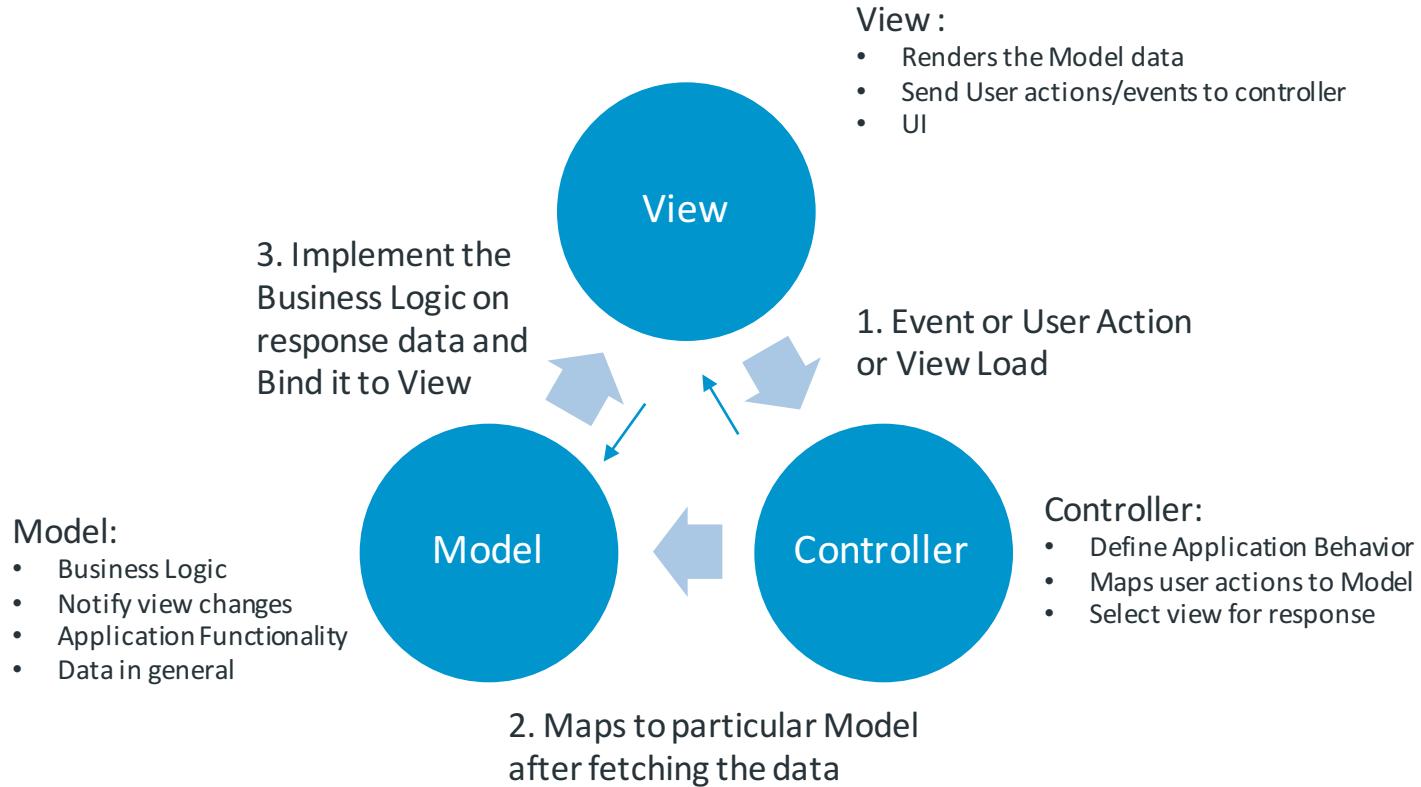
```
<!Doctype html>
<html ng-app>
 <head>
 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.0-rc.2/angular.min.js"></script>
 </head>
 <body>
 <div>
 <label>Name:</label>
 <input type="text" ng-model="yourName" placeholder="Enter a name here">
 <hr>
 <h1>Hello {{yourName}}!</h1>
 </div>
 </body>
</html>
```

# Perchè AngularJS?

- Permette di organizzare una pagina con una buona struttura logica.
- Migliora HTML permettendo l'uso directives, custom tags, attributes, expressions, templates within HTML.
- Usa un linguaggio dichiarativo
- Incoraggia l'uso dei MVC/MVVM design pattern
- Ottime per single page app(SPA)

## **BINDING AUTOMATICO**

# MVC : Model View Controller



# AngularJS attribute: **ng-app**, **ng-controller**, **ng-scope**

- **ng-app** è un attribute che indica lo scope nella pagina html dell'applicazione angularjs
  - Solitamente viene aggiunto al tag HTML `<html ng-app>`
- **ng-controller** permette di fare il binding fra la view e il controller
  - Solitamente viene aggiunto a body o al tag di cui è il controller  
`<div ng-controller="ctrl">`
- **ng-scope** è il modello. Tutte le variabili contenute nello scope del controller possono essere mostrate all'interno della sua view relativa (rivedere esempio)

# Module

- Modules specifica come una applicazione dovrebbe essere fatta partire (è l'init dell'applicazione angular)
- Possono esserci più moduli nella nostra app.

```
// declare a module
var myAppModule = angular.module('myApp', [--here goes the dependent Modules--])
```

- Dependent modules sono moduli dipendenti che ci aiutano nello sviluppo con librerie specializzate, vedremo il modulo di loopback nelle prossime slide

# Module

- Analizziamo per blocchi il codice `/client/angulartest2.html` nel repository in [github.com/anonymez/advancedwebprogramming](https://github.com/anonymez/advancedwebprogramming)

# Lunch Break



Until 2:00 PM

# Service and Factory

I servizi sono i componenti Angular che offrono funzionalità indipendenti dall'interfaccia utente. Essi, in genere, consentono di **implementare la logica dell'applicazione**, cioè le funzionalità che si occupano di elaborare e/o recuperare i dati da visualizzare sulle view tramite i controller.

```
angular.module("myApp")
 .factory("myfactory", function() {
 //...
});
```

Il Servizio definito tramite il metodo `service()`, Angular ci fornisce un'istanza della funzione associata al servizio. Quando utilizziamo un servizio definito tramite il metodo `factory()`, Angular ci fornisce il valore restituito dall'esecuzione della funzione associata.

# Service and Factory

```
angular.module("myApp")
 .service("somma1", function() {
 this.somma = function(a,b) { return a + b;};
 });
```

```
angular.module("myApp")
 .factory("somma2", function() {
 return function(a, b) { return a + b;};
 });
```

```
angular.module("myApp")
 .controller("myController",
 function($scope, somma1, somma2) {
 $scope.x = somma1.somma(1,2);
 $scope.y = somma2(1,2)
 });
```

# TODO App e AngularJS

- Vediamo alcuni costrutti di angular applicati alla nostra applicazione.
  - Creiamo la nostra app
  - Creiamo il nostro controller
  - Aggiungiamo nello scope l'oggetto/lista todos
  - Andiamo a modificare l'html form utilizzando ng-model
  - Utilizziamo ng-class
  - Utilizziamo ng-repeat (magia)



## 7. Loopback Frontend

Make frontend integration  
easy

# AngularJS e Loopback

- The LoopBack AngularJS SDK has three major components:
  - Auto-generated AngularJS services, compatible with [ngResource.\\$resource](#), that provide client-side representation of the models and remote methods in the LoopBack server application.
  - The lb-ng command-line tool that generates Angular \$resource services for your LoopBack application.
  - A Grunt plugin ([grunt-loopback-sdk-angular](#)), if you want to use Grunt instead of lb-ng.
- The client is dynamic, in other words it automatically includes all the LoopBack models and methods you've defined. You don't have to manually write any static code.
- The SDK fits seamlessly into the workflow of a front-end developer:
  - The generated Angular objects and methods have [ngdoc](#) comments. Use an ngdoc viewer like [Docular](#) to view documentation of the client available to your AngularJS client.
  - If you wish, you can use the provided Grunt task to generate the client services script, which make it easy to include this file in an existing Grunt-based workflow (for example for bundling or "minification").

# AngularJs e Loopback

```
$ mkdir js
$ lb-ng ../server/server.js js/lb-services.js
```

```
var app = angular.module('mapp', ['lbServices']);
app.controller('myCtrl',
['$scope', 'Todo', 'User',
function($scope, Todo, User) {
}]);
```

# chiamate asincrone e promise

```
function getTodos() {
 Todomysql
 .find()
 .$promise
 .then(function(results) {
 $scope.todos = results;
 });
}
getTodos();
```

# User login

```
User.login({email: "me@domain.com", password: "secret", rememberMe: true});
```

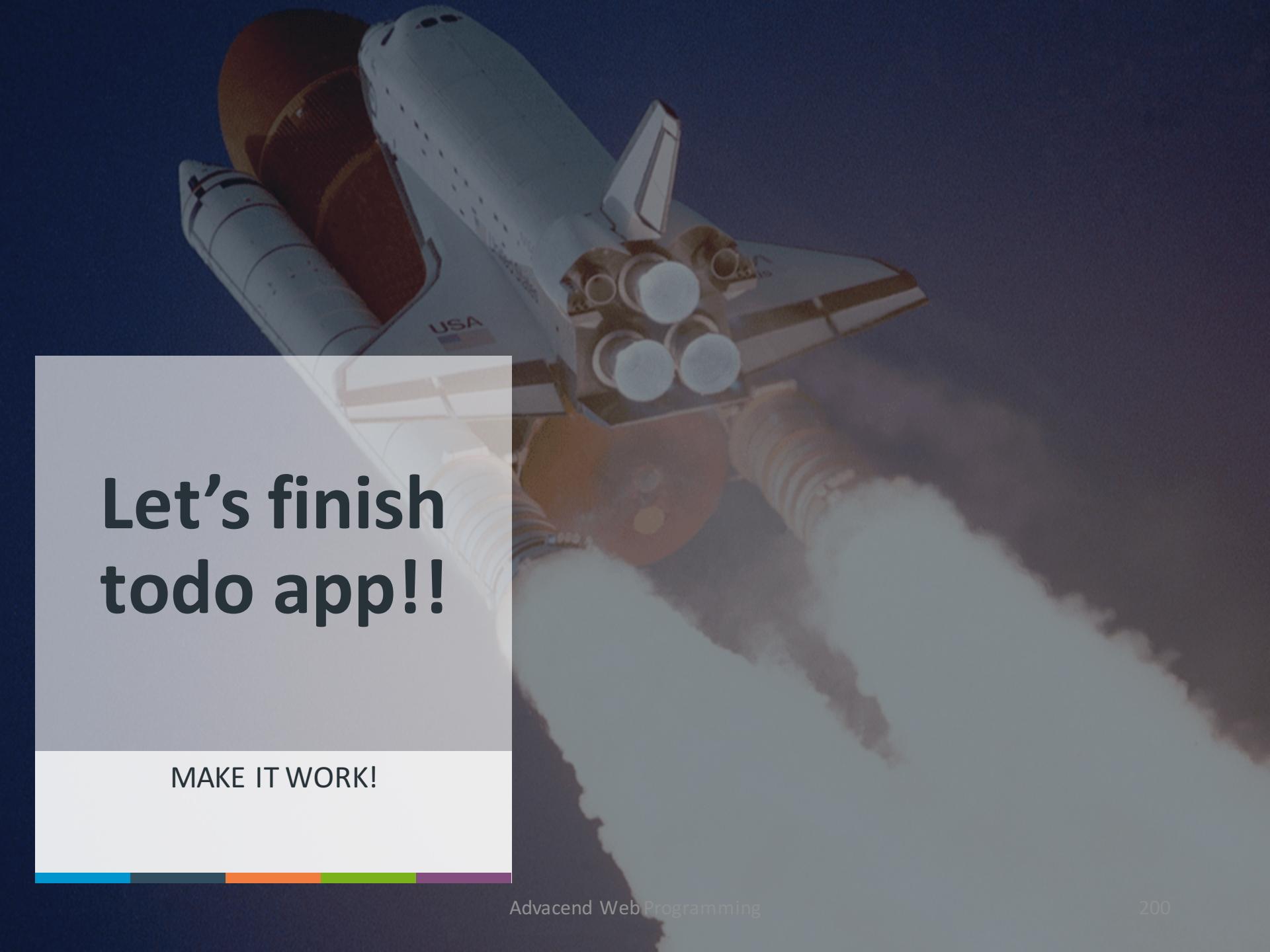
# Http request

```
// Simple GET request example:
$http({ method: 'GET', url: '/someUrl' }).
then(function successCallback(response) {
 // this callback will be called asynchronously
 // when the response is available
}, function errorCallback(response) {
 // called asynchronously if an error occurs
 // or server returns response with an error status.
});
```

# factory and promises

```
01 angular.module('atTheMoviesApp', [])
02 .controller('GetMoviesCtrl',
03 function($log, $scope, movieService) {
04 $scope.getMovieListing = function(movie) {
05 var promise =
06 movieService.getMovie('avengers');
07 promise.then(
08 function(payload) {
09 $scope.listingData = payload.data;
10 },
11 function(errorPayload) {
12 $log.error('failure loading movie', errorPayload);
13 });
14 };
15 })
16 .factory('movieService', function($http) {
17 return {
18 getMovie: function(id) {
19 return $http.get('/api/v1/movies/' + id);
20 }
21 }
22 });

```



**Let's finish  
todo app!!**

MAKE IT WORK!

**T**HANK **Y**OU!

Spero il corso vi sia piaciuto :)

[filippo.gaudenzi@unimi.it](mailto:filippo.gaudenzi@unimi.it)

## Documentations:

- Loopback
  - <http://loopback.io/doc/en/lb2/>
- Bootstrap
  - <http://getbootstrap.com/getting-started/>
- AngularJS
  - <https://angularjs.org>
  - <https://www.youtube.com/watch?v=AZM7D2NyOE>
  - <https://www.youtube.com/watch?v=KGLTsulDS0g>
  - <https://www.youtube.com/watch?v=yYhMfLrOAJE>
- Javascript



marco.anisetti@unimi.it  
filippo.gaudenzi@unimi.it



@Filippogau



# Coffee Break

15 minutes





# Lunch Break



Until 2:00 PM

# Lunch Break



Until 2:00 PM