

# Cloud e mobile

Silviu Filote

June 23, 2021

## Contents

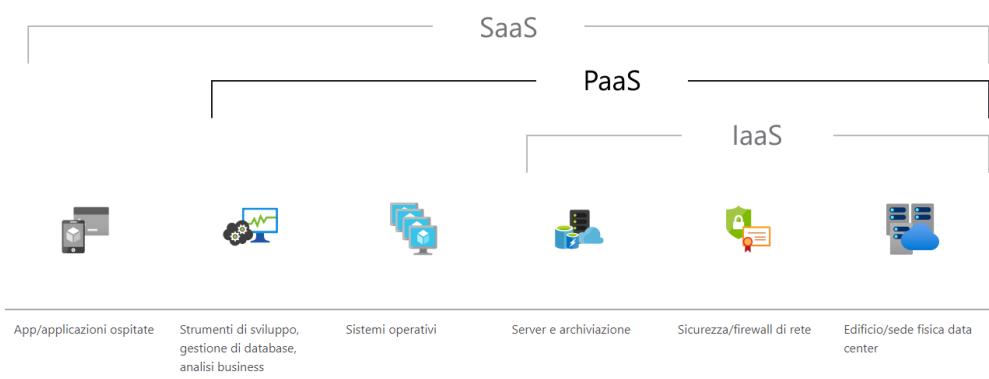
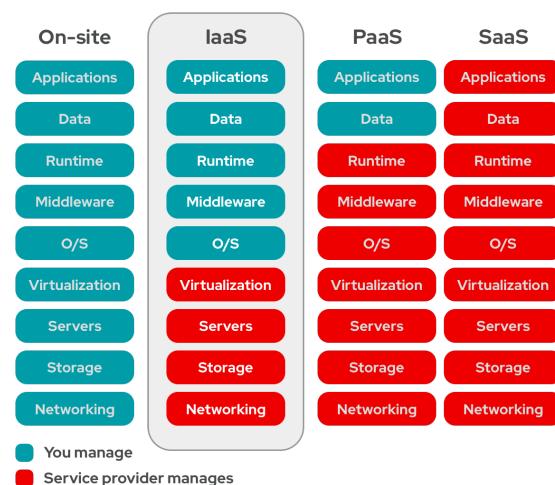
<b>1</b>	<b>Tecnologie Cloud e Mobile</b>	<b>2</b>
<b>2</b>	<b>JavaScript, Elementi Base</b>	<b>7</b>
<b>3</b>	<b>XML</b>	<b>13</b>
<b>4</b>	<b>JavaScript 2</b>	<b>20</b>
<b>5</b>	<b>AJAX e JSON</b>	<b>24</b>
<b>6</b>	<b>AJAX CON JQuery</b>	<b>33</b>
<b>7</b>	<b>XML Avanzato</b>	<b>36</b>
<b>8</b>	<b>Node.js</b>	<b>43</b>
<b>9</b>	<b>MongoDB e Node.js</b>	<b>46</b>
<b>10</b>	<b>Python</b>	<b>50</b>
<b>11</b>	<b>Pythone Moduli, Map-Reduce</b>	<b>59</b>
<b>12</b>	<b>BlockChain</b>	<b>67</b>
<b>13</b>	<b>Micro-Services e ReST</b>	<b>80</b>
<b>14</b>	<b>XML Schema e WSDL</b>	<b>90</b>

# 1 Tecnologie Cloud e Mobile

- I sistemi web si basano su **protocolli aperti** che consentono di realizzare un'infrastruttura di comunicazione **aperta** ⇒ chiunque può accedere, pubblicare un servizio web
- Ogni protocollo di rete agisce ad un determinato livello della pila **ISO-OSI**
- Per la realizzazione di questi sistemi web abbiamo bisogno di:
  - database / file system
  - DBMS
  - web server (che gestisca le chiamate HTTP)
  - macchine computazioni che ospitano queste risorse
- Problematiche dei server in casa → **Hosting**
- **Hosting:** fornisce risorse computazionali sulle sue macchine e i relativi costi/problematiche vengono gestite da terzi
- Per sfruttare al massimo le risorse hardware del server a disposizione vengono utilizzate le **macchine virtuali**
- **Macchine virtuali:** indica un software che, attraverso un processo di virtualizzazione, crea un ambiente virtuale che emula tipicamente il comportamento di una macchina fisica, grazie all'assegnazione di risorse hardware.
  - sulla stessa macchina possono essere emulate molteplici macchine virtuali
  - Soluzione ripartizione
    - \* soluzione tradizionale: **ripartizione fissa** delle risorse
    - \* soluzione mondo cloud: **ripartizione dinamica** delle risorse → **Pay per Use**
- Hosting → macchine fisiche → macchine virtuali
- Si passa ad un'infrastruttura di elaborazione che ridistribuisce le macchine virtuali sulla **nuvola server** → **Cloud computing**

- I layer del cloud computing:

- **Infrastruttura:** il sistema cloud fornisce l'infrastruttura di elaborazione, che poi il cliente gestisce, viene chiamata **IaaS: Infrastructure as a Service**
- **Piattaforma:** si intende la piattaforma sulla quale si sviluppano e si eseguono i programmi e sistemi informativi, comprende diversi strumenti resi disponibili (sviluppo, testing, deploying ...). E' conosciuta come **PaaS: Platform as a Service**
- **Software:** si intende il software applicativo usato dagli utenti, noto anche come **SaaS: Software as a Service**



- Storage as a Service: dropbox, google drive ..

- Le applicazioni mobili sono dette semplicemente **app**

- **Tipologie di app:**

- **Applicazioni Native (native applications):**

- \* Sono scritte nello specifico linguaggio di programmazione nativo della piattaforma mobile
    - \* La stessa App va completamente riscritta, per passare da una piattaforma ad un'altra

- **Applicazioni Ibride (Hybrid applications):**

- \* Come evitare di riscrivere le applicazioni per passare da iOS ad android e viceversa? usare un altro linguaggio di programmazione
    - \* Ci sono 2 modalità di esecuzione di queste app
      - **Interpretazione**
      - **Traduzione**
    - \* **Interpretazione:** il linguaggio viene interpretato all'interno di uno scheletro base
      - JavaScript + HTML
      - Web View: componente per visualizzare HTML
      - Interprete JavaScript
      - Bridge: libreria per consentire al codice di accedere al sistema operativo

- **Web App (embedded web applications):**

- \* L'app che si installa sul dispositivo mobile contiene solo una Web View
    - \* Le schermate sono pagine HTML + JavaScript che vengono recuperate dal server
    - \* l'app è di fatto un'applicazione web

- Le App devono comunicare con i server di appoggio

- server forniscono API

- **Progettazione integrata:** separare l'interfaccia dai servizi

- Tanti componenti diversi
  - Ognuno con un ruolo ben preciso

- Esempi:

- SaaS
  - \* La suite Google
  - \* Microsoft Office 365
  - \* Overleaf
- PaaS
  - \* Google Cloud Platform
  - \* Amazon Web Service
  - \* Microsoft Azure
- IaaS
  - \* HP
  - \* IBM

- Esempi:

- Applicazioni Native Android:
  - \* Linguaggio di programmazione: Java
  - \* Ambiente di sviluppo: Android Studio / eclipse
  - \* Target intermedio: Byte-code
  - \* Target finale: Dalvik virtual machine, con evoluzione del Byte-code a 16 bit
  - \* Distribuzione attraverso Google Play Store
  - \* Tutto il processo di pubblicazione è gestito via Web
  - \* **ha introdotto un nuovo linguaggio di programmazione *Kotlin*, più snello di Java con tipizzazione dinamica**
- Applicazioni Native iOS:
  - \* Linguaggio di programmazione: Objective C
  - \* Ambiente di sviluppo: XCODE su Mac
  - \* Distribuzione attraverso iTunes Store
  - \* Non si possono pubblicare App senza Mac
  - \* L'app viene preparata da XCODE, che la invia ad iTunes Store
  - \* **ha introdotto un nuovo linguaggio di programmazione *Swift*, molto più semplice e stabile di Objective C con tipizzazione dinamica**

- Esempi:

- **Apache Cordova**: applicazione ibrida interpretata
  - \* il framework che integra il bridge e l'interprete JavaScript
  - \* Vantaggi: riutilizzo stesse conoscenze di programmazione web
  - \* Svantaggio: esecuzione risulta essere molto lenta
- **Microsoft Xamarin**: applicazione ibrida tradotta
  - \* L'applicazione viene scritta in C#
  - \* Xamarin traduce il codice C#
    - In Java, per Android
    - Nel codice binario per iOS
  - \* Vantaggi: Le applicazioni sono molto veloci in esecuzione
  - \* Svantaggio: richiede di acquisire competenze specifiche C#
- **React Native**
  - \* Ideato e spinto da Facebook
  - \* Il codice REACT-native viene tradotto in Java (per Android) e i Objective C (per iOS)
  - \* Il linguaggio di programmazione è JavaScript
  - \* Ma non viene interpretato

## 2 JavaScript, Elementi Base

- **Client-Side Script:** programmi annegati nella pagina HTML che vengono interpretati all'interno del browser
- JavaScript fa una valutazione dinamica:
  - delle istruzioni
  - dei tipi delle variabili
- **Interprete:** è un programma in grado di eseguire altri programmi a partire direttamente dal relativo codice sorgente scritto in un linguaggio di alto livello, senza la previa compilazione dello stesso (codice oggetto), eseguendo cioè le istruzioni nel linguaggio usato traducendole di volta in volta in istruzioni in linguaggio macchina del processore.
- Essendo il linguaggio interpretato abbiamo difficoltà di debugging elevata
- All'interno si racchiude tutto il codice JavaScript

```
<script type="text/javascript">  
...  
</script>
```

- L'attributo **language** consente di specificare la versione, **ma non è necessario**

```
<script type="text/javascript" language="javaScript1.5">  
...  
</script>
```

- basta semplicemente specificare script

```
<script>  
...  
</script>
```

- **variabili:** Il tipo di dato è derivato dinamicamente

- var

- \* contestualizza la variabile localmente
  - \* var v = 1 numero intero
  - \* var n = "Pippo" Stringa

– const

- \* la variabile non è modificabile, è una costante
  - \* const d = 1

- let

- \* contestualizza la variabile nel blocco di codice che contiene la definizione (per esempio, tra graffe)

```
var d=1;  
{ let d=2; }  
// qui d vaut 1
```

## • Funzioni

- Non essendoci i tipi è necessaria una parola chiave per diversificare la funzione da una variabile → **function**

```
function nome ( parametri formali )
{
    codice
}
```

- Il tipo dei parametri formali e del valore restituito dalla funzione non sono specificabili
- Vengono valutati dinamicamente

<pre>FUNCTION SOMMA (x,y) {     VAR w = x + y;     RETURN w; }</pre>	<pre>VAR A = 2; VAR B = 3; VAR C = SOMMA(A,B) C → 11</pre>	<pre>VAR A = "a"; VAR B = "b"; VAR C = SOMMA(A,B) C → "ab"</pre>
--	--	--

- **funzioni senza nome**
  - \* la funzione = **oggetto**
  - \* per cui possono essere assegnate a variabili

<pre>function (x, y) {     var r = x + y;     return r; }</pre>	<pre>var fsomma = function (x, y) {     var r = x + y;     return r; }</pre>
---	--

- **Oggetti**

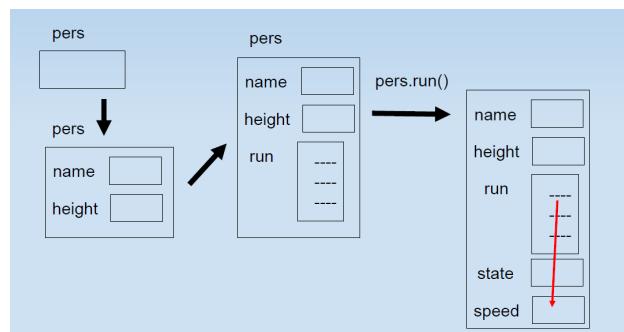
- può avere dei campi
- può avere dei metodi
- Esistono tre modalità per definire oggetti in JavaScript:

- \* Modalità 1: **Estensione di Object**

- Object è l'oggetto base, vuoto (o quasi)
- i campi si possono aggiungere dopo
- I campi sono come delle variabili contenute all'interno dell'oggetto
- in generale a un campo si può assegnare una funzione

```
pers = new Object()
pers.name = "Giuseppe"
pers.height = "1.80m"
pers.run = function()
  { this.state = "running"
    this.speed = "4m/s"
  }
```

- **this** prende il riferimento dell'oggetto corrente
- dopo la chiamata **pers.run()**; l'oggetto **pers** ha due campi in più



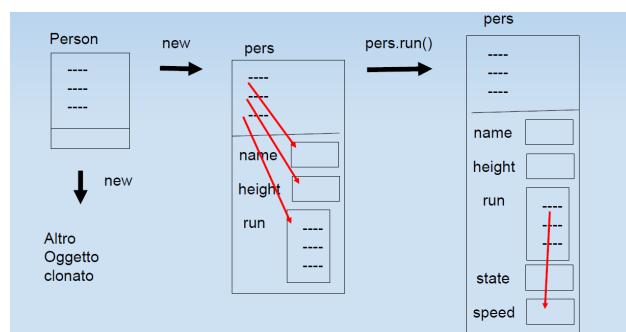
### \* Modalità 2: Quasi una classe → Clonazione

- Creare una Struttura Comune da replicare facilmente facilmente
- Nei linguaggi tipizzati questa struttura viene chiamata **classe**
- **Javascript procede per *Clonazione***
- viene costruito l'oggetto tramite il **costruttore**, che inizializza le proprietà e i metodi di un oggetto in maniera parametrica e poi si **Clona**

```
function Person(pname, pheight) {
    this.name = pname
    this.height = pheight
    this.run = function()
    { this.state = "running"
        this.speed = "4m/s"
    }
}
```

- **utilizzo di this** → un oggetto e una funzione sono la stessa cosa
- **this** accede allo spazio dati dell'oggetto implicitamente associato alla funzione
- una volta definito il costruttore / funzione si procede con la clonazione attraverso la parola **new**

```
var pers = new Pearson("Giuseppe", "1.80m");
pers.run();
```



- \* Modalità 3: **Literals**

- all'interno delle { } si definiscono al volo i campi e i metodi
- questa modalità è utile per predisporre degli oggetti di configurazione normalmente richiesti dai framework

```
pers = {
    name : "Giuseppe",
    height : "1.80m",
    run : function()
    { this.state = "running"
        this.speed = "4m/s" }
}
```

- JavaScript non supporta l'Information hiding
- Chiunque può vedere il contenuto di qualsiasi oggetto
- Il linguaggio fornisce due tipi di oggetti standard

- **Array**

- \* var a = new Array(10)
    - \* L'array è un oggetto
    - \* Il valore tra parentesi è il numero di elementi iniziali
    - \* **Gli elementi dell'array non sono tipizzati:**
      - a[0] = "Pippo";
      - a[1] = 2;

- **String**

- \* Le costanti stringa "testo" o 'testo' vengono automaticamente convertite → **String**
    - entrambi gli oggetti possiedono metodi già definiti

### 3 XML

- XML: eXtensible Mark-up Language ovvero Linguaggio a Marcatori Estendibile
- Non fornisce un insieme di marcatori predefinito (come HTML)
- Fornisce gli strumenti linguistici per definire l'insieme di marcatori e le regole di correttezza sintattica
- Vantaggi:
  - Non è legato a nessuna piattaforma hardware/software
  - Adatto quindi per l'interscambio di documenti tra applicazioni diverse
- Svantaggi:
  - Eccessiva “verbosità” del linguaggio
  - Rispetto a un formato piatto e senza marcatori, un documento XML richiede molti più Byte
  - La rappresentazione in memoria centrale richiede una struttura ad albero, la cui costruzione e navigazione non sono banali
- Elementi del Linguaggio:
  - ad ogni marcatore di apertura corrisponde un marcatore di chiusura con lo stesso nome

`<Nome> .... </Nome>`

- Una coppia di marcatori di apertura e di chiusura descrive un **Elemento**
- Un **Elemento** ha un contenuto e delle proprietà → **Attributi**
  - \* l'attributo definisce un proprietà del concetto descritto dall'elemento
  - \* il valore può essere racchiuso tra 'apici ' o "virgolette "

`<Nome attributo = "valore" > .... </Nome>`

- Esempi:
  - \* `< SECTION title="A Citation">`
  - \* `< CITE label='Knuth68' />`

```

<SECTION title="A Citation">
  Here we have an example of citation. We
  refer to
  Knuth's paper which introduced the
  concept of attribute
  grammar. This paper has the number <CITE
  label="Knuth68"/>
  in our bibliography.
</SECTION>

```

- **Elemento vuoto**, senza contenuto

- \* Può essere scritto nella forma compatta:

**<Nome/ >**

- \* Esempio:

**<CITE label="Knuth68"/>**

- Tutti i documenti XML sono iniziati dal marcatore di apertura

**<?xml version="1.0"?>**

- È possibile specificare l'encoding del testo

**<?xml version="1.0" encoding="ASCII" ?>**

- Un documento che rispetti tutti i vincoli sintattici appena descritti viene definito → **Ben Formato**
- I vincoli di contenuto possono essere specificati **nel DTD (Document Type Definition)**. E definisce:

- Elementi ammessi
- Struttura del contenuto degli elementi
- Attributi degli elementi
- \* le regole vengono scritte tramite → **Meta-Tags**

- **Meta-Tags**

**< !ELEMENT Nome StrutturaCont. >**

- **Nome**: il nome dell'elemento che si definisce
- **StrutturaCont**: specifica il numero di occorrenze dell'elemento e di che tipo è il contenuto
  - \* occorrenze:
    - ( ... )<sup>+</sup> → c'è almeno un elemento,  $\geq 1$
    - ( ... )<sup>\*</sup> → o vuoto o più di un elemento

- ( . . . )? → o vuoto oppure un solo elemento
- E1, E2, E3 → sequenza di elementi, figli
- (E1 | E2 | E3) → alternativa
- \* tipo contenuto:
  - (#PCDATA) → testuale
  - (#PCDATA | E1 | E2 | . . . )\* → misto
  - EMPTY → vuoto

< !ATTLIST Elemento Nome Tipo Obblig. >

- **Elemento:** per il quale si definiscono gli attributi
- **Nome:** dell’attributo
- **Tipo:** tipologia dell’attributo, ossia valore dell’attributo
  - \* CDATA → stringa generica
  - \* ID → identifica l’elemento, l’id è univoco
  - \* IDREF → riferimento all’ID di un altro elemento
- **Obblig:** opzione di obbligatorietà
  - \* #REQUIRED → attr. obbligatorio
  - \* #IMPLIED → attr. facoltativo
  - \* valore\_default → il valore quando non viene specificato

```

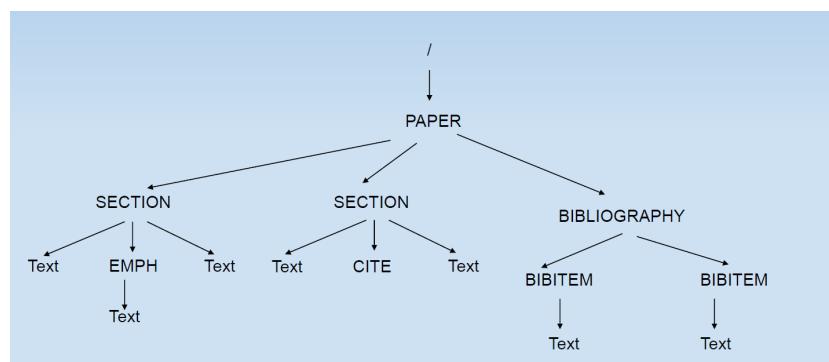
<!ELEMENT BIBITEM (#PCDATA)>
<!ATTLIST BIBITEM label ID #IMPLIED>
<!ELEMENT BIBLIOGRAPHY (BIBITEM)+>
<!ELEMENT CITE EMPTY>
<!ATTLIST CITE label IDREF #REQUIRED>
<!ELEMENT EMPH (#PCDATA)>
<!ELEMENT SECTION (#PCDATA | CITE | EMPH)*>
<!ATTLIST SECTION title CDATA #REQUIRED>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT PAPER (TITLE, (SECTION)*, (BIBLIOGRAPHY)?)>
<!ATTLIST PAPER name ID #REQUIRED>

```

- Nel documento XML si specifica il DTD, Dopo il marcatore di preambolo, il file **”paper.dtd”** contiene il DTD

<!DOCTYPE PAPER SYSTEM "paper.dtd">

- Il rispetto delle regole definite nel DTD porta ad un più alto livello di correttezza dei documenti. Un documento corretto da questo punto di vista viene detto → **Valido**
- Un **Processore XML** è un programma che ha il compito di processare un documento XML
- Un **Parser** è uno strumento software che effettua l'analisi sintattica di testi basati su un linguaggio artificiale
- XML è un linguaggio artificiale, quindi, per poter processare un documento XML occorre un **XML Parser**
- Tipologie di Parsing:
  - **SAX**
    - \* Modalità detta a **eventi**
    - \* al parser / **un'interfaccia** si deve fornire un oggetto **ContentHandler**
    - \* l'handler fornisce un metodo specifico per ogni elemento sintattico
    - \* es: quando il parser incontra un marcatore di apertura, chiama un metodo specifico, così come di chiusura
  - **DOM (Document Object Model)**
    - \* Il parser costruisce una rappresentazione del documento in memoria centrale → struttura ad **albero**

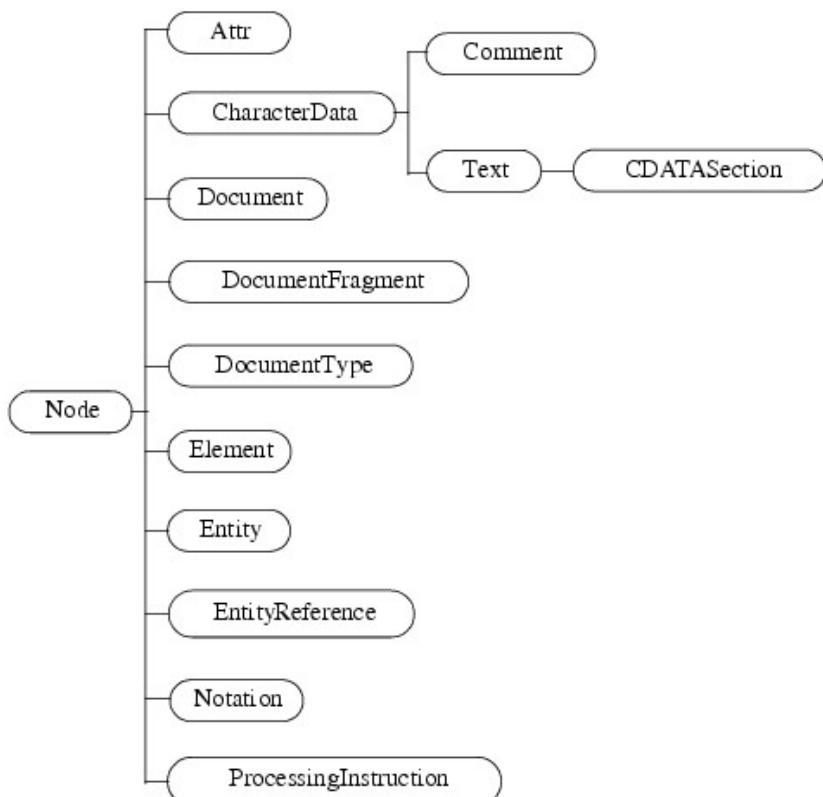


- \* I nodi dell'albero descrivono:
  - Gli elementi
  - Blocchi di testo

- Gli attributi degli elementi
- Un'altra classificazione di parser:
  - **Non validante:** il parser verifica solo che il documento sia *ben formato*
  - **Validante:** Il parser verifica la correttezza del documento rispetto al DTD, che sia *valido*
  - **Validante rispetto a XML Schema:** E' un'alternativa più moderna e flessibile al DTD. Il parser valida il documento rispetto alla specifica XML Schema
- Cenni su html e SGML
  - HTML è un linguaggio a marcatori con un insieme ben definito di marcatori
  - Il capostipite di XML e HTML è **SGML** che prevedeva l'utilizzo di marcatori per descrivere libri e articoli
  - SGML prevedeva dei costrutti che non sono mai stati implementati da nessuno, perché erano troppo complicati
  - Tim Berners-Lee aveva lavorato al progetto SGML, ne capì le potenzialità e definì **HTML**
  - Nacque poi **XML** Aperto e indipendente dalla piattaforma hardware/software e sarebbe dovuto diventare il formato dei dati sul web
    - \* Alcune scelte sintattiche di HTML non lo rendevano compatibile con XML
    - \* Esempio:
      - <br> → HTML
      - <br/ > → XML
  - HTML 5 è compatibile con la sintassi di XML
  - i browser sono tolleranti a scritture non propriamente corrette
- JavaScript e XML
  - Anche in JavaScript si possono elaborare documenti XML
  - Si passa sempre attraverso la rappresentazione DOM
  - **La gerarchia dei nodi** viene mantenuta in **Javascript** senza l'ereditarietà

- *Ogni pezzo (item) di un documento XML viene rappresentato da un oggetto Nodo (Node)*
- *Le relazioni di annidamento degli elementi nel documento XML, vengono rappresentate in DOM da una relazione PADRE - FIGLIO → Struttura ad Albero*
- Tutti i nodi dell'albero sono definiti sulla **classe Node**
- Node viene specializzata in sottoclassi che corrispondono ai diversi tipi di item XML
  - \* Element
  - \* Attribute
  - \* text
  - \* ....
- La classe Node possiede diversi campi, uno tra questi è **Record-Type** che indica il tipo di nodo mediante una **costante**
  1. ELEMENT\_NODE
  2. ATTRIBUTE\_NODE
  3. TEXT\_NODE
  4. CDATA\_SECTION\_NODE
  5. COMMENT\_NODE
  6. DOCUMENT\_NODE
  7. DOCUMENT\_TYPE\_NODE
  8. DOCUMENT\_FRAGMENT\_NODE
- **Element** estende Node, contiene nuovi campi per la gestione degli attributi, getElementById, ...
- **CharacterData** è una sottoclasse di Node, gestisce il contenuto testuale dei documenti XML (Qualsiasi pezzo di testo in mezzo a due marcatori)
- **Text** è una sotto-classe di CharacterData → **i nodi testuali sono di tipo Text**
  - \* l'unica cosa che cambia da CharacterData è che la sottoclasse aggiunge un metodo Text splitText
- **Document** contiene l'intero documento
  - \* Consente di creare Elementi e Testi
  - \* Consente di cercare gli Elementi

- **NodeList** classe che descrive liste di nodi, utile per rappresentare i nodi figli
- **NamedNodeMap**
  - \* Descrive gli insiemi di attributi di un elemento
  - \* Fornisce metodi per ottenere gli attributi in base a: nome attributo o posizione
  - \* I metodi restituiscono un Node



- Creare Documenti XML
  - Per inviare un documento XML occorre **serializzarlo**, cioè generare il testo corrispondente.
  - Gli interpreti JavaScript forniscono un oggetto XMLSerializer
  - il processo di **Parsing**: da testo a DOM → Si usa un oggetto DOMParser in Javascript

## 4 JavaScript 2

- Eventi

- Un evento si scatena quando qualcosa accade sugli elementi della pagina HTML
- Il browser gestisce solo alcuni eventi previsti dallo standard
- Ma è possibile aggiungere dei **gestori di evento**, che sono **funzioni JavaScript**

```
<p onClick="return fReazione();"> Clicca qui </p>
```

- L'attributo onClick specifica il gestore dell'evento, il suo valore è **return fReazione();** → unnamed function
- il gestore dell'evento restituisce:
  - \* False, l'evento viene fermato il browser non presegue
  - \* True, il browser prosegue con la gestione dell'evento
- Ogni elemento HTML ha eventi specifici
- L'elemento **BODY** prevede **onLoad**, che viene invocato quando tutta la pagina è stata caricata
  - \* Consente di svolgere attività di inizializzazione di vario tipo
- L'elemento **FORM** prevede l'evento **onSubmit**
  - \* L'attributo **action** form contiene l'indirizzo del programma sul server al quale mandare il contenuto della form
  - \* Questo evento viene generato quando l'utente preme il pulsante Submit
  - \* Se la gestione dell'evento non viene fermata, parte la chiamata HTTP

- Eccezioni

- una porzione di codice viene monitorata durante la sua esecuzione, se un errore occorre, un'eccezione viene generata
- L'eccezione può essere catturata, evitando che il programma si interrompa, gestendola
- L'eccezione in questo caso **e** contiene delle proprietà:
  - \* **name:** nome dell'eccezione
  - \* **message:** il messaggio dell'eccezione
- Se il blocco catch è vuoto, l'eccezione viene ignorata → Da non fare

```

try
{ ... /* Codice monitorato */ }
catch(e)
{ ... /* Gestione dell'errore */ }
finally(e) // opzionale
{ ... /* sempre eseguito */ }

```

- per conoscere il tipo di eccezione si utilizza **instanceof**

```
if (e instanceof TypeError){ .. }
```

- Oggetti predefiniti

- L'interprete JavaScript del Browser fornisce una serie di oggetti predefiniti

- \* **Window**

- Window è l'oggetto di più alto livello
  - Contiene al suo interno tutti gli altri oggetti
  - Ha molti campi e metodi: **document**, **alert(msg)**, **blur()**, **focus()** ..

- \* **document**

- L'oggetto document rappresenta la pagina HTML su cui si sta lavorando
  - **Il contenuto è rappresentato in formato DOM**
  - Il contenuto può essere modificato dal codice JavaScript

- HTML

- La pagina HTML è rappresentata tramite **DOM**
- L'oggetto predefinito **Document** contiene la rappresentazione **DOM della pagina**
- Quando il DOM viene modificato, il browser rigenera la visualizzazione (**rendering**) della pagina
- **In questo modo, il codice JavaScript controlla ciò che è visualizzato**
- La classe **Element** del DOM viene ulteriormente estesa dalla classe **HTMLElement**
- In HTML, tutti gli elementi possono avere un attributo di nome **id**

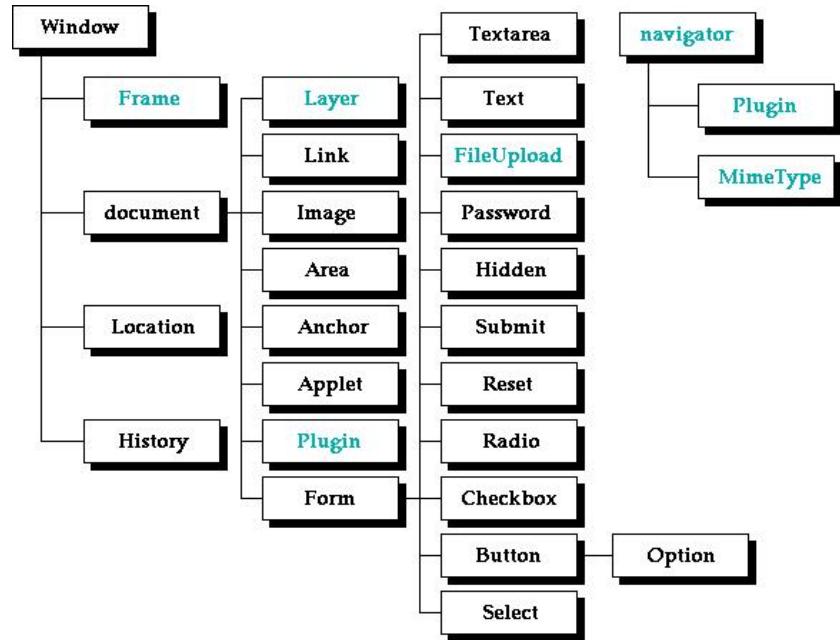


Figure 1: classi / oggetti JavaScript

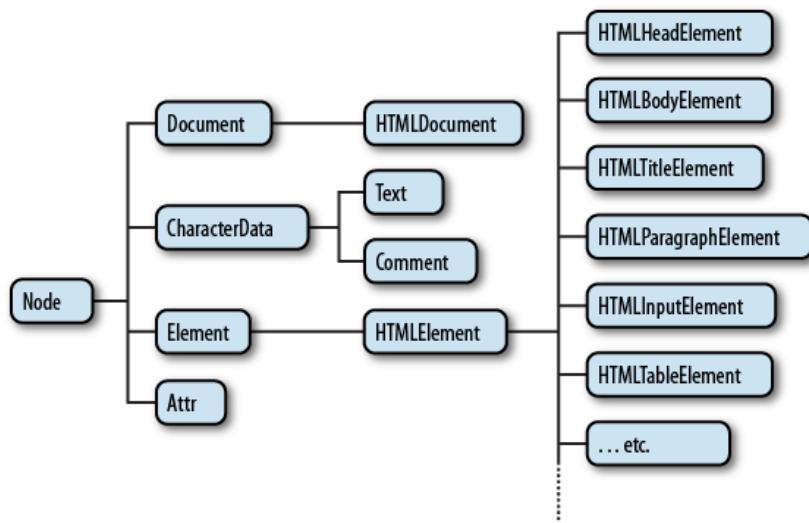


Figure 2: rappresentazione DOM

- Conversioni
  - **parseInt(s)**: conversione da stringa a numero intero
  - **parseFloat(s)**: conversione da stringa a numero in virgola mobile
- il valore **NaN** è una sorta di valore neutro
  - funzione booleana per controllare se un valore sia Nan o meno **isNaN(s)**
- La funzione **eval()** consente di eseguire (valutare) la stringa ricevuta come parametro attuale, per evitare il rischio di **JavaScript injection** non viene usata

## 5 AJAX e JSON

- **Siti Web Statici:** contenuto statico, i cosiddetti siti vetrina
- **Siti Web Dinamici:** contenuto generato dinamicamente da programmi che operano sul Web Server, la pagina viene interamente rigenerata ad ogni richiesta di elaborazione
- **tier:** annodatore, ossia la connessione tra due strati software
- Le architetture per realizzare siti web:
  - **1-tier Architecture**
    - \* adatta per siti statici
    - \* **il tier è il protocollo HTTP**
    - \* **funzionamento:**
      - Il browser invia una richiesta HTTP, richiedendo una specifica risorsa
      - Il server riceve la richiesta e se la risorsa richiesta è presente sul disco la invia al client, impacchettandola nella Risposta HTTP

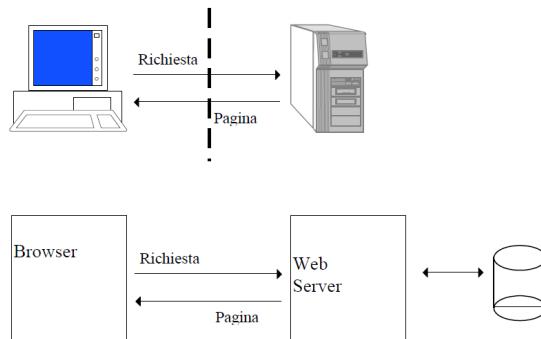


Figure 3: tier 1 architetture

- **2-tier Architecture**
  - \* Adatta per i siti dinamici
  - \* Dove i contenuti devono essere generati da programmi che lavorano sul server
  - \* **Il primo tier è il protocollo HTTP**
  - \* **Il secondo tier è il protocollo CGI** (Common Gateway Interface) tra il web server e l'applicativo CGI

\* **funzionamento:**

- Il browser invia una richiesta HTTP al server
- Il server riceve la richiesta HTTP, vede che è per un programma
- Attiva il programma (Applicativo CGI) e gli passa il contenuto della richiesta
- L'applicativo CGI genera la pagina HTML (o altro contenuto)
- Il server impacchetta l'output dell'applicativo CGI nella Risposta HTTP e viene inviata al browser sul client

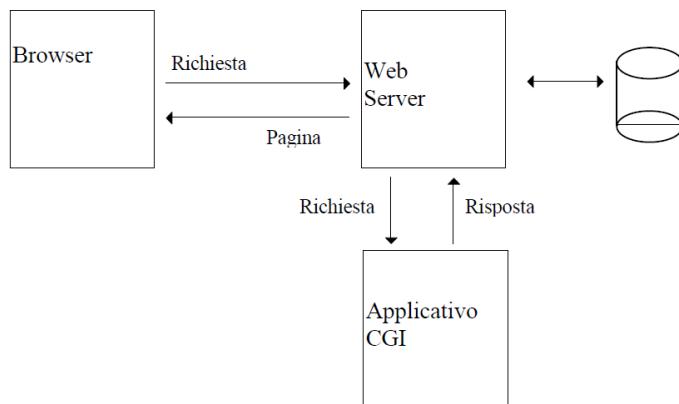


Figure 4: tier 2 architetture

– **3-tier Architecture**

- \* Interazione con il DBMS
- \* **Il primo tier è il protocollo HTTP**
- \* **Il secondo tier è il protocollo CGI** (Common Gateway Interface) tra il web server e l'applicativo CGI
- \* **Il terzo tier è la connessione con il DB**
  - il protocollo più generico si chiama **ODBC** (Open DataBase Connectivity)
  - Con Java, il protocollo di base si chiama **JDBC**
- \* **funzionamento:**
  - Il browser invia una richiesta HTTP al server
  - Il server riceve la richiesta HTTP, vede che è per un programma

- Attiva il programma (Applicativo CGI) e gli passa il contenuto della richiesta
- L'applicativo CGI apre la connessione con il DB, invia le query e riceve le tabelle
- Genera l'output e lo invia al web server
- Il web server impacchetta il contenuto nella Risposta HTTP e lo invia al browser

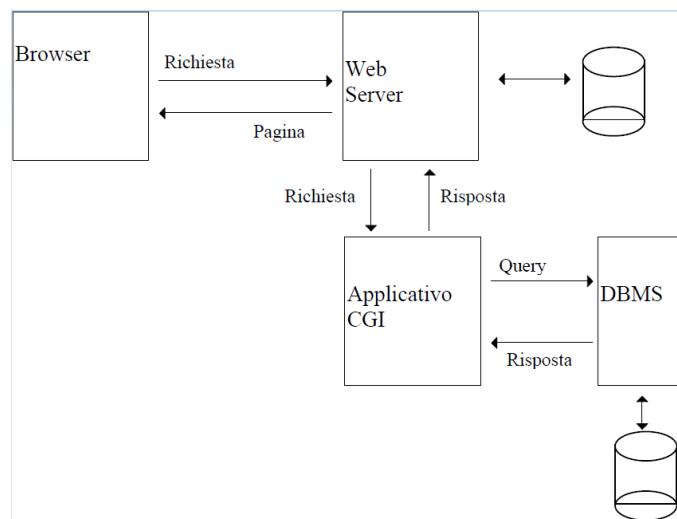


Figure 5: tier 3 architetture

- La Richiesta HTTP può essere fatta utilizzando 2 modalità
  - **Metodo GET**
    - \* **corpo della richiesta è vuoto**
    - \* Nella head, il campo URL contiene l'indirizzo della risorsa con l'aggiunta dei campi della form
    - \* i campi sono visibili dopo il carattere ?, **campo = valore** separati da & e sono del tipo
   
<http://...?email=a@unibg.it&nome=Pippo>
    - \* Es: come se sul fronte della busta mettiamo tutti i dati, ma la busta la lasciamo vuota
  - **Metodo POST**
    - \* i dati non sono visibili nel URL

- \* viene utilizzato quando ci sono molto campi, dati sensibili o tante informazioni
- \* **il corpo non è vuoto**

email=a@unibg.it&nome=Pippo

```
<form method="GET" action="...">
  onSubmit="return controlla();">
    <input type="text" name="email"/>
    <input type="text" name="nome"/>
    <input type="submit" value="Invia"/>
</form>
```

Figure 6: form

- Protocollo CGI

- è stato il primo protocollo per far comunicare il web server con le applicazioni lato server
- **Metodo GET:** i valori dei campi erano ricevuti come variabili d’ambiente
- **Metodo POST:** il contenuto del corpo della richiesta era ricevuto come standard input
- Negli anni sono stati sviluppati i **server-side script**
  - \* L’idea era quella di mischiare codice HTML con un server-side script
  - \* Esempi: PHP, JSP, VBA, C#
- CGI, di fatto, non viene più usato

- **AJAX (Asynchronous JavaScript And XML)**

- Le prime applicazioni web dinamiche erano un po’ scomode da usare
- Tutto era fatto dal server, quindi ogni azione fatta dall’utente richiedeva di aspettare l’arrivo della pagina rigenerata **richiesta bloccante (sincrona)**
- JavaScript associato alla pagina può comunicare direttamente con il server, in modo trasparente all’utente

- \* Per consentire al codice JavaScript di comunicare con il server è stato aggiunto un oggetto specifico → **XmlHttpRequest**
- \* È stato necessario estendere l'oggetto predefinito **Window** con specifiche funzionalità per gestire la **comunicazione asincrona**
- **Asynchronous:** Il client invia una richiesta al server, ma la risposta arriva in maniera asincrona, quindi non **bloccante**, infatti il client nel frattempo può fare altro
- **JavaScript:** Pensato per rendere a tutti gli effetti i codici JavaScript delle applicazioni client-server
- **XML**
  - \* I messaggi inviati dal server sono in XML e rappresentanti con la struttura **DOM**
  - \* usando il metodo POST, si possono mandare documenti XML nel corpo della richiesta HTTP
  - \* Javascript ha il compito di gestire questi messaggi XML, integrandoli nella pagina per essere visibili all'utente

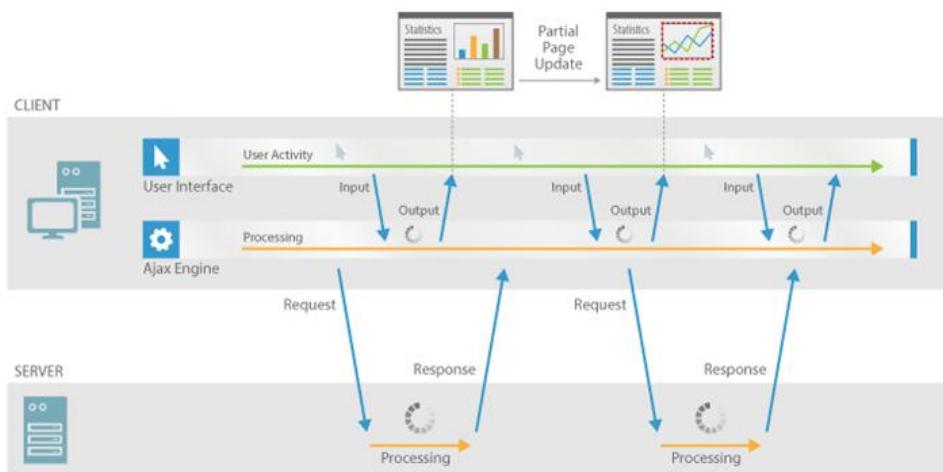


Figure 7: chiamata ajax

### • Tecnologia AJAX Base

- Gli interpreti JavaScript sono stati estesi per gestire la **comunicazione con il server**, fare il **parsing** del documento XML ricevuto dal server e **costruire la struttura dati DOM**

- \* L'oggetto built-in dell'interprete → **XmlHttpRequest**
- **Effettuare una richiesta:**
  1. Si clona l'oggetto con l'istruzione **new XMLHttpRequest()**
  2. Si impostano le proprietà e i metodi
  3. Si chiama il metodo **open**, per aprire la connessione
  4. Si chiama il metodo **send**, per inviare la richiesta.
- **Funzionamento:**
  - \* Lo script invia una **richiesta al server** tramite l'oggetto **XmlHttpRequest** e l'oggetto attende la risposta
  - \* L'oggetto XMLHttpRequest riceve la risposta HTTP e analizza il contenuto
    - Se questo è un documento XML, costruisce l'albero DOM  
→ **campo responseXML**
    - Altrimenti lo tratta come testo → **campo responseText**
  - \* Quando la risposta arriva chiama una funzione dello script detta **callback**, fornendo il messaggio ricevuto
  - \* Le funzioni di callback, sono funzioni passate come parametri di altre funzioni da eseguire al termine di una operazione asincrona;

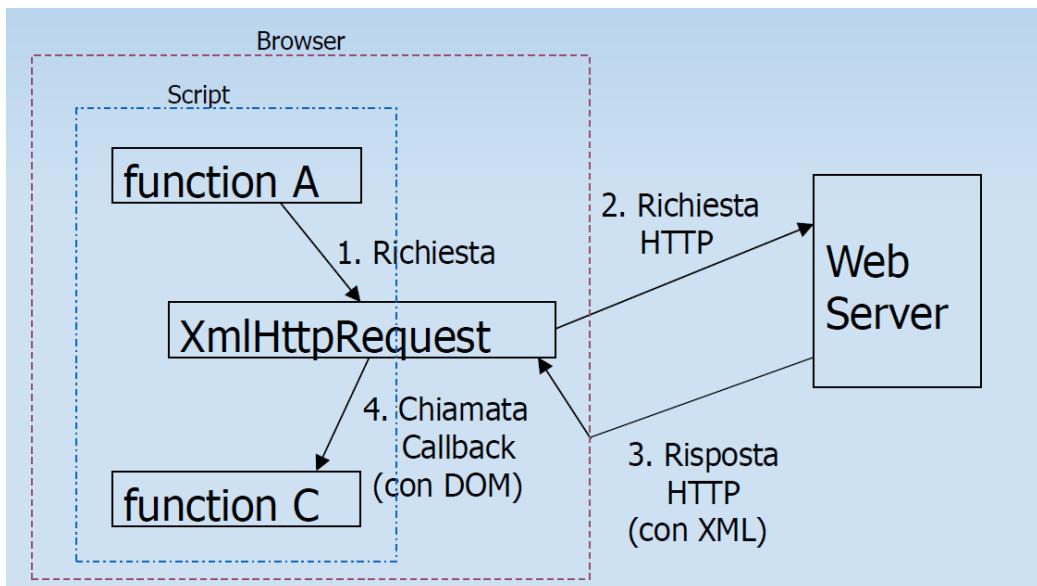


Figure 8: principio di funzionamento

- XMLHttpRequest - Tipo di comunicazione:
  - \* **Sincrona:** l'oggetto XMLHttpRequest invia la richiesta HTTP e aspetta la risposta (lo script è bloccato) → **Deprecata**
  - \* **Asincrona:** l'oggetto XMLHttpRequest passa su un **thread parallelo**; l'interprete JavaScript non si blocca e lo script continua a lavorare
- **campi XMLHttpRequest**
  - \* **readyState:** Stato della comunicazione, se il valore è **4**, indica che la risposta è arrivata
    - Quando il suo valore cambia l'oggetto XMLHttpRequest chiama il metodo **onreadystatechange**
  - \* **Valori readyState:**
    0. **Uninitialized:** Oggetto non inizializzato (metodo open non invocato)
    1. **Open:** Metodo open invocato, ma metodo send non invocato
    2. **Sent:** Metodo send invocato, ma i campi responseText e responseXML non hanno valore
    3. **Receiving:** Alcuni dati ricevuti, ma i campi responseText e responseXML non hanno valore
    4. **Loaded:** Dati ricevuti, i campi responseText e responseXML hanno valore ((se il documento ricevuto è un XML corretto))
  - \* **onreadystatechange:** funzione di callback interna
    - **Campo status:** il codice HTTP (200 = OK)
    - **Campo statusText:** il testo associato al codice HTTP
- Non Usate makeAjaxRequest perchè:
  - Non è robusta
  - Non fa le chiamate cross-domain
  - Non gestisce in automatico la correttezza dei contenuti ricevuti
  - usare **JQuery**
- **JSON (JavaScript Object Notation)**
  - Gestire un documento XML, per quanto già nella rappresentazione DOM, è complicato → **Sia lato client che lato server**

- I programmatori abbiano iniziato a usare un formato diverso → **JSON**
- Deriva dalla terza forma per creare oggetti in JavaScript, ossia i **Literals**
- Un documento JSON contiene solo dati
- Con poco codice si converte un documento JSON in un oggetto JavaScript

```

1  {
2    "string": "Hi",
3    "number": 2.5,
4    "boolean": true,
5    "null": null,
6    "object": { "name": "Kyle", "age": 24 },
7    "array": [ "Hello", 5, false, null, { "key": "value", "number": 6 }],
8    "arrayOfObjects": [
9      { "name": "Jerry", "age": 28 },
10     { "name": "Sally", "age": 26 }
11   ]
12 }
13

```

Figure 9: codice JSON

- Nomi dei campi tra virgolette (no apici singoli)
- Esistono
  - \* **Campi semplici:** numeri, stringhe (sia con doppi apici che apici singoli)
  - \* **Campi complessi:** oggetti annidati
  - \* **Campi array:** array di valori semplici e/o oggetti
- possiamo creare una stringa che contiene un documento JSON, **serializzando** l'oggetto
- da stringa a oggetto → **deserializzare**
- **eval** è problematica in termini di sicurezza. Conviene usare una libreria, chiamata **JSONparser**

**Per convertire la stringa in oggetto:**  
`varmyObject = JSON.parse( myJSONtext);`

**Per ottenere la stringa da un oggetto:**  
`varmyText = JSON.stringify( myJSONObject);`

```
var myJSONText =  
' {"bindings": [ {"ircEvent":  
    "PRIVMSG", "method": "newURI",  
    "regex": "http://.*"},  
    {"ircEvent": "PRIVMSG",  
    "method": "deleteURI", "regex":  
    "delete.*"} ] }';
```

Figure 10: varmyObject

## 6 AJAX CON JQUERY

- JQuery è un framework JavaScript, con lo scopo di fornire funzionalità di alto livello più sofisticate e complesse
  - nasce per interrogare il DOM, Al fine di trasformarlo con costrutti semplici e potenti
  - gestisce le chiamate AJAX in modo più robusto
- Il cuore del framework è l'**oggetto jquery** ed è rappresentato tramite → \$, contiene metodi e ha lo scopo di interrogare il DOM
- La chiamata AJAX viene direttamente supportata da JQuery
- La chiamata cross-domain è supportata direttamente
- Possono essere gestiti molti formati di risposta
- **L'oggetto jqXHR**
  - Incorpora ed estende l'oggetto **XmLHttpRequest**
  - Memorizza i dati ricevuti
  - Gestisce le funzioni di callback
  - Ci permette di accedere ai campi e metodi di **XmLHttpRequest**: status, statusText, ...
  - framework JQuery costruisca l'oggetto jqXHR per **estensione dell'oggetto base**, aggiungendo i suoi campi e i suoi metodi
- **L'oggetto \$ (jquery)** fornisce il metodo **ajax**
  - Il metodo ajax restituisce l'oggetto jqXHR
  - Il metodo ajax riceve un oggetto di configurazione, che contiene le informazioni per gestire la chiamata nel modo opportuno
    - \* Il modo migliore per preparalo è usare i **Literals, ossia prepararli al volo**
    - \* Oggetto di configurazione:
      - **url:** l'url(anche cross-doain) della richiesta AJAX
      - **type:** GET, POST
      - **cross-domain:** false (default), true
      - **dataType:** il tipo di dato da ricevere (xml, html, json, jsonp, text)

- **data:** il contenuto della richiesta da inviare (oggetto serializzabile o stringa)
- **Headers:** un oggetto con gli header della richiesta
- **mimeType:** stringa con il MIME type della richiesta
- **username:** nome utente per la connessione (opzionale)
- **password:** password per la connessione (opzionale)

```

1 | var menuId = $( "ul.nav" ).first().attr( "id" );
2 | var request = $.ajax({
3 |   url: "script.php",
4 |   method: "POST",
5 |   data: { id : menuId },
6 |   dataType: "html"
7 | });
8 |
9 | request.done(function( msg ) {
10 |   $( "#log" ).html( msg );
11 | });
12 |
13 | request.fail(function( jqXHR, textStatus ) {
14 |   alert( "Request failed: " + textStatus );
15 | });

```

Figure 11: **visitare:** <https://api.jquery.com/jQuery.ajax/>

- **Il Formato JSONP (JSON with Padding)**

- La risposta contiene la **chiamata ad una funzione**, che deve essere presente nel codice JavaScript
- Tipicamente, le chiamate in modalità GET specificano la funzione con cui fare il padding con il parametro jsonp → può essere pericoloso

*Process({ "result": 5 })*

- **Esistono altri metodi per fare chiamate AJAX in casi specifici**

- **\$.get:** chiamata con metodo GET
- **\$.getJSON:** chiamata GET che riceve uno JSON
- **\$.getScript:** chiamata GET che riceve uno script
- **\$.post:** chiamata con metodo POST

- Accesso al DOM

- Quando l'oggetto **\$** è usato come funzione, contiene dei **selettori** dei nodi del DOM, tipicamente si usano i selettori **CSS**

- Gli oggetti restituiti sono estensioni del DOM proprie di JQuery
- E forniscono metodi per poterli modificare
- Esempi:
  - \* `$('#mioblocco');`
  - \* `$('#mioblocco').css("border","2px solidgreen");`
  - \* `('.miaclasse');`
- La funzione `$` (o `jqyery`) restituisce **liste di nodi**
- Esempi:
  - \* `('#menu li').size();`
  - \* `('#menu li').length;`
  - \* `('.miaclasse')`
- *Per ottenere la rappresentazione DOM base di JavaScript, invece di quella estesa di JQuery*
  - \* Metodo: `get(indice)` fornisce l'elemento (in DOM puro) nella posizione indicata
  - \* `('#menu li').get();`
- Restituisce l'elemento (in DOM JQuery) nella posizione indicata
  - \* Metodo: `eq(indice)` → in DOM JQuery
  - \* `('#menu li').eq(0);`
- Dato un elemento DOM Jquery:
  - \* si possono manipolare i suoi attributi
  - \* accedere al suo contenuto testuale
  - \* accedere al suo contenuto HTML
  - \* si possono gestire le sue classi di stile
  - \* Processare ogni singolo elemento di una selezione

`("#menu li").each(function(i,el){...});`

*i: posizione*  
*el: è l'elemento (DOM puro)*

  - \* si possono accedere ai campi delle form **attraverso i selettori css**
- Si possono impostare eventi e gestori di evento

## 7 XML Avanzato

- In XML, un **namespace** definisce elementi specifici, per poter usare questi elementi, occorre indicare a quale namespace appartengono
- Ogni namespace usato nel documento ha un **prefisso**, che precede il nome dell'elemento
- Prefisso e nome sono separati da :
  - Es: soap:Envelope
  - **prefisso:nome elemento**
- Ma i prefissi vanno definiti, viene utilizzato l'attributo **xmlns**
  - Es: xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  - **xmlns:prefisso**
  - viene usato come valore **l'URI dello standard**, per identificare una risorsa, non per trovarla → **xmlns, dato un prefisso, fa riferimento alla risorsa**
- funzionamento:
  - Il processore analizza il documento
  - Vede l'URI associato a xmlns
  - Se lo conosce, è in grado di processare gli elementi appartenenti a quel namespace
  - Se non li conosce, li scarta/ignora
- in questa maniera si possono integrare nello stesso documento elementi appartenenti a namespace diversi
- **SOAP (Simple Object Application Protocol)**
  - È un protocollo del W3C usato per lo scambio di messaggi tra sistemi informativi
  - I messaggi sono documenti XML
  - È composto da:
    - \* **Envelope** e contiene al suo interno
      - **Header**, opzionale, contiene metadati come quelli che riguardano l'instradamento, la sicurezza, le transazioni

- **Body**, trasporta il contenuto informativo e talora viene detto carico utile (payload)
- il formato con cui vengono inviati tali messaggi è generico, non è relativo al protocollo SOAP
- funzionamento:
  - \* Il server SOAP riceve un messaggio SOAP
  - \* L'handler del protocollo SOAP riceve il contenuto XML del messaggio
  - \* Sa gestire i suoi elementi, non gli altri
  - \* Estrae il frammento nel body e lo passa al componente software del sistema informativo, che è in grado di processarlo

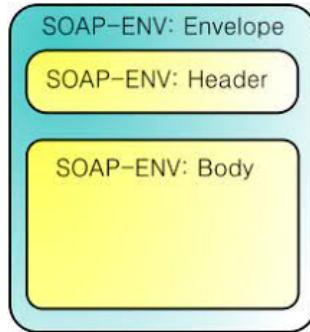


Figure 12: soap message

#### • XSLT e FO

- definire il concetto di Foglio di Stile per un documento XML
- il figlio di stile contiene le refeole per dare uno stile al documento
- Il linguaggio utilizzato → **XML**
- Nasce così **XSL (eXtensible Style-sheet Language)**
  - \* Un insieme di elementi che descrivono come formattare gli elementi di altri documenti XML
  - \* Ottenendo la versione HTML del documento originale
  - \* un documento XML specifica come processare un altro documento XML
  - \* Il W3C ha esteso questo approccio con la necessità di
    - Ristrutturare (trasformare) documenti XML in altri documenti XML

- Generare documenti HTML come trasformazioni complesse di documenti XML
- Nasce **XSLT**

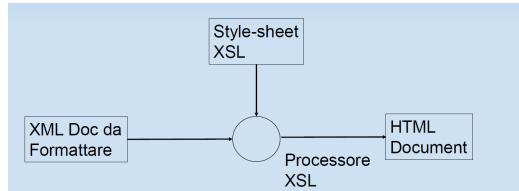


Figure 13: funzionamento

- **XSLT (eXtensible Style-sheet Language Transformation)**
  - \* non è più un semplice foglio di stile, ma un vero e proprio linguaggio dichiarativo per specificare trasformazioni complesse dei documenti XML, **usando la sintassi di XML**

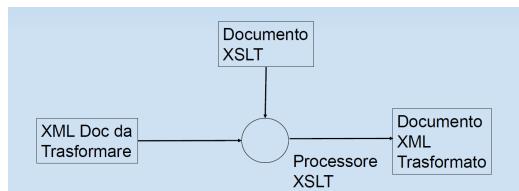


Figure 14: funzionamento

- Approccio **dichiarativo**: XSLT non è procedurale, quindi non è possibile cambiare il valore delle variabili o scrivere cicli
- Ma perché trasformare un documento XML in un altro documento XML? → **Per sfruttare altre tecnologie XML, esempio: FO**
- **FO (Formatting Objects)**
  - \* **Definisce ciò che compone un documento, come:**
    - Testi
    - Aree/box
    - Dimensioni
    - Linee
  - \* Non è legato ad uno specifico formato di output, infatti può generare: pdf, RNG, RTF, ....
  - \* Il processore FO è parametrico rispetto al formato target
  - \* Usando sempre come linguaggio XML

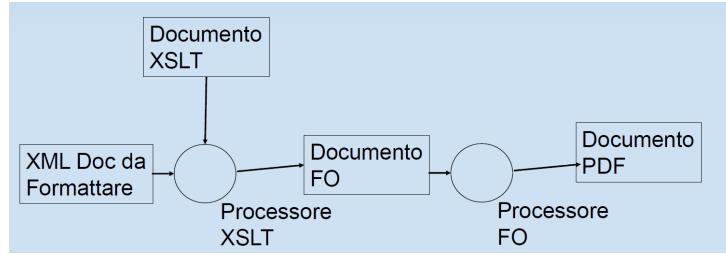


Figure 15: funzionamento

- XSLT in Dettaglio

- Il foglio di stile è contenuto nell'elemento

**xsl:stylesheet**

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...
</xsl:stylesheet>

```

Figure 16: esempio

- Un documento XSLT contiene delle regole che vengono applicate sugli elementi da formattare
    - \* le regole sono chiamate **template**

```

<xsl:template match="/">
...
</xsl:template>

```

Figure 17: esempio

- L'attributo **match** indica su quali elementi deve scattare la regola ((con il linguaggio **XPath**))
  - Grazie al namespace, il processore XSLT riesce a distinguere ciò che deve elaborare e ciò che deve lasciare inalterato.
  - I Template possono avere un nome

- \* l'attributo **name** dà un nome al template
 

```
<xsl:template name="Interno">
```
- \* Questo consente di invocare l'esecuzione del template quando serve
 

```
<xsl:call-template name="Interno"/>
```
- Quando un template viene processato, la visita dell'albero DOM si ferma, se non si dice di procedere ricorsivamente
  - \* L'elemento **<xsl:apply-templates>** dice al processore XSLT di visitare i discendenti, cercando di applicare i template definiti
- Il linguaggio **XPath** viene usato per specificare su quali elementi del documento sorgente applicare i template
  - \* Fa parte dei cosiddetti **XML Query Languages**
  - \* Sono linguaggi pensati per interrogare documenti XML e generare altri documenti XML
  - \* XPath chiamato così perchè con sente di selezionare dei path relativi/assoluti per selezionare gli elementi
  - \* Il linguaggio che è diventato il riferimento in questo settore si chiama **XQuery**
  - \* XPath prevede anche **funzioni** che forniscono valori relativi agli elementi selezionati

## • Il Linguaggio XPath

- **/** : Fa riferimento all'elemento radice
- **/A** : Fa riferimento all'elemento radice A, se l'elemento radice è diverso, il match non avviene
- **//A** : Fa riferimento ad un qualunque elementA, in qualsiasi posizione nel documento
- **/A/B/C** : Specifica l'annidamento, la radice A contiene B che a sua volta contiene C; il match avviene su C
- **/A/B//C** : Il match avviene su un qualunque C, purchèal di sotto di B contenuto nella radice A
- **/A/B/\*** : Cerca qualsiasi elemento figlio di B contenuto sotto la radice A
- **/A/B/@at="Espressione"** : preleva il valore dell'attributo at di B

- `//B[C='v']` : Cerca tutti gli elementi B il cui figlio C contiene il testo 'v'
- `//B[@at='v']` : Cerca tutti gli elementi B il cui attributo atvale 'v'

- **funzioni XPath**

- ***position()*** : Fornisce la posizione di un elemento nella lista degli elementi selezionati
- `//A[position() = 5]` : Seleziona l'elemento A in posizione 5 tra tutti gli A selezionati
- ***last()*** : Fornisce il numero di elementi nella lista degli elementi selezionati
- `//A[position() = last()]` : Seleziona l'elemento A in ultima posizione tra tutti gli A selezionati
- ***count(selezione)*** : Conta il numero di elementi selezionati
- ***count(//A[position() = last()])*** : Conta quanti elementi A sono in ultima posizione nella lista
- ***current()*** : Fa riferimento all'elemento corrente
- ***name()*** : Il nome completo (Q-Name) del primo elemento tra quelli selezionati (o quello corrente)
- `//myns:A/mame()` : Restituisce il nome completo dell'elemento selezionato, cioè 'myns:A'
- ***local-name()*** : Il nome senza prefisso (Local Name) del primo elemento tra quelli selezionati (o quello corrente)
- `//myns:A/local-mame()` : Restituisce il nome locale dell'elemento, cioè 'A'
- ***namespace-uri()*** : Restituisce l'URI del namespace dell'elemento selezionato
- ***text()*** : Restituisce il nodo DOM corrispondente al contenuto testuale dell'elemento selezionato

- **Estrarre Valori dal DOM**

- estrae il valore dell'attributo Nome

```
<xsl:value-of select="@Nome"/>
```

## • Variabili

- Sono variabili dichiarative, cioè non possiamo cambiarne il valore, una volta definite
- Questo elemento definisce la variabile il cui nome è indicato dall'attributo **name**

```
<xsl:variable name="conteggio">
```

- Il valore della variabile è il contenuto dell'elemento **xsl:variable**
- Il valore può essere ottenuto dall'elemento **xsl:value-of**
- L'attributo **select** prende il valore del nodo corrente, che è il testo
- Per far riferimento alle variabili, si usa la notazione **\$nome**

```
<xsl:value-of select="$conteggio"/>
```

## • Condizioni

- XSLT fornisce elementi condizionali
- Primo elemento: **xsl:if**
  - \* L'attributo **test** contiene la condizione da verificare (su variabili, funzioni, attributi)

```
<xsl:if test="position()=1">
```

- Se la condizione è vera, il contenuto viene messo in output, altrimenti no.
- L'elemento **xsl:choose** consente di gestire le alternative
- Un blocco **xsl:when** gestisce un'alternativa
  - \* Vi possono essere molti blocchi XSL:when
  - \* Al termine, se nessun blocco xsl:when è stato attivato, il blocco **xsl:otherwise** viene attivato

## • Processare Liste di Elementi

- L'elemento **xsl:for-each** consente di processare una lista di elementi
- Processando un elemento per volta
- Non è un ciclo

## 8 Node.js

- Due linguaggi di programmazione diversi tra il lato client e il lato server richiedono competenze diverse
- Far spostare i programmati JavaScript sullo sviluppo del lato server → Obiettivo di Node.js e usare quindi un solo linguaggio
- Usare la tecnologia JavaScript sul lato server, per manipolare i dati JSON con il linguaggio all'interno del quale sono nati
- **npm** consente di installare pacchetti aggiuntivi a Node.js
- I programmi Node.js sono **fortemente basati sulla programmazione a eventi**
  - In JavaScript, sono le funzioni di callback
- Node.js gestisce diverse code di eventi, a diversa priorità
- Il cosiddetto **EventLoop** ha il compito di
  - Prendere un evento dalla coda non vuota a più alta priorità
  - Chiamare la funzione di callback associata
  - Ripetendo questo processo all'infinito
- Richieste da una Porta TCP
  - Una richiesta da una porta TCP viene trasformata in evento
  - L'evento viene accodato ad una coda di eventi
  - Quando è il turno, l'evento viene processato e la funzione di callback corrispondente viene chiamata
  - In questo modo, il sistema può gestire richieste multiple su porte diverse
- Un programma per Node.js non lavora nel browser
  - L'oggetto Window non esiste
  - Il nuovo oggetto di contesto si chiama **global**
- **Funzionamento**
  - Un programma JavaScript per Node.js ha il compito di **attivare un server**

- \* Cioè mettersi in ascolto su una porta TCP e rispondere alle richieste entranti
- \* Inoltre, può gestire in modo nativo il protocollo HTTP (e HTTPS)
- le richieste che arrivano sono eventi e vengono gestite tramite **EventLoop**
- Perciò, global fornisce oggetti utili a creare il server e a gestire il processo di comunicazione
  - \* Global quindi possiede un insieme di metodi e oggetti

### • Esecuzione

- Dato un programma, per esempio server.js
- Con **node server.js** si avvia il programma
- Essendo un server, il processo è attivo e aspetta di servire qualche richiesta
- Per interrompere: Ctrl + C

### • Scrittura del programma

- Nella parte iniziale, occorre specificare i moduli
  - \* **http:** modulo predefinito per creare server http
  - \* **url:** modulo predefinito per manipolare gli URL
- Vengono definite due costanti
  - \* Indirizzo IP dell'Host
  - \* Porta TCP
- Viene creato il server, con **http.createServer**
  - \* Il parametro è la funzione di callback da chiamare quando arriva una richiesta
  - \* Il server viene assegnato alla variabile/costante server

```
const server = http.createServer(function (req, res)
  { ..... });
```

  - \* **Parametri:**
    - **req:** oggetto che descrive la richiesta HTTP Classe: http.ClientRequest
    - **res:** oggetto che gestisce la risposta Classe: http.ServerResponse
  - Occorre associare il server all'host e alla porta

- \* Il metodo mette in ascolto il server sull'host e sulla porta specificata
- \* La funzione di callback viene chiamata quando il server è effettivamente in ascolto

```
server.listen(port, hostname, function() ...);
```

- **Invocazione:**

- Usiamo il seguente URL: `http://127.0.0.1:8080/?a=b&c=d`
  - \* Che invia due parametri **a** e **c**

- Debugging

- Si può fare debugging del codice JavaScript, attivando Node.js in modalità di debug
- Viene aperta una porta TCP (9229) specifica che dei tool esterni possono contattare

```
node - -inspectserver.js
```

- Se la porta 9229 risulta occupata
- Si può cambiare la porta

```
node - -inspect-port=9228 - -inspectserver.js
```

- **paradigma del Model-View-Controller (MVC)**

- Consentono di gestire in modo ingegneristico lo sviluppo e la manutenzione delle applicazioni web
- Suddivide l'applicazione in tre parti:
  - \* **Model:** il gestore del modello dei dati, normalmente il DB
  - \* **View:** l'interfaccia utente
  - \* **Controller:** il codice che controlla le trasformazioni dei dati, in base alle richieste ricevute dal view
- Esempio, non basato su AJAX:
  - \* Model: il DB
  - \* View: il codice HTML che viene inviato al server, il cui contenuto viene generato a partire dai dati ottenuti dal DB
  - \* Controller: il codice lato server che esegue le query sul DB e comanda la rigenerazione della pagina (gestisce la **business logic** dell'applicazione)
  - \* Un framework molto usato: Spring

## 9 MongoDB e Node.js

- I DBMS relazionali basati su SQL con l'avvento dei Big Data e dei formati semi-strutturati, come XML e JSON, li hanno resi inadatti al nuovo mondo dei Big Data → **a causa della loro rigidità**
- NoSQL:
  - Not only SQL indica che non esiste solo SQL, c'è anche altro
- I cosiddetti JSON Databases, o JSON Stores, sono una particolare categoria di NoSQL DB → **MongoDB è il più famoso e usato**
- Modello dei Dati
  - **Database:** è insieme di collezioni
  - **Collezione:** un insieme di documenti JSON eterogenei (nome univoco nel database)
  - **Istanza della collezione:** un insieme di documenti JSON senza alcun vincolo di struttura
- Oggetti e ID
  - **Una volta memorizzato, ogni oggetto JSON ha un identificatore univoco** → *campo \_id*
  - Questo valore può essere già presente quando l'oggetto viene caricato
  - Se non è presente, viene aggiunto e valorizzato automaticamente dal sistema
- MongoDB
  - MongoDB è un DBMS
  - Raggiungibile su una porta specifica di un server specifico
  - In Linux esiste un'interfaccia prompt chiamata **mongo**, in windows non esiste
  - Linguaggio di Query
    - \* Adotta un **approccio a oggetti**, con la **dot notation**
    - \* Oggetto di base: **db** → Corrisponde all'intero database, che ovviamente contiene le collezioni
    - \* **Approccio alla JSON:**

- ogni collezione è un campo dell'oggetto db, con il nome differente da quello delle altre collezioni **db.MyCollection**

- \* **In alternativa:**

- **db.getCollection('MyCollection')**

- \* L'oggetto collezione fornisce alcuni metodi per interrogare la collezione o modificarla (operazioni DML)
- \* **db** è un oggetto JavaScript, infatti, possiamo definire funzioni JavaScript da usare nelle query

- Metodi:

- **db.collezione.insertOne(name: "Pippo")**
- **db.collezione.insertMany(array di oggetti)**
- **db.collezione.find(argomento)** → interrogare la collezione
- **db.inventory.find()** → Se non si specifica nessun oggetto, tutto il contenuto della collezione è recuperato
- **db.inventory.find(status: "D")** → Equivale ad una condizione di uguaglianza, su un campo che deve essere presente
- {**<field>: { \$eq: <value> }**} → valore uguale a quello specificato
- {**<field>: { \$ne: <value> }**} → valore diverso
- {**<field>: { \$gt: <value> }**} → valore maggiore
- {**<field>: { \$gte: <value> }**} → valore maggiore o uguale
- {**<field>: { \$lt: <value> }**} → valore minore
- {**<field>: { \$lte: <value> }**} → valore minore o uguale
- **db.inventory.find({qty:{\$in: [ 5, 15 ]}})** → Il campo deve avere un valore contenuto nell'array di valori specificato.
- \$nin** vero se il valore non è nell'array specificato

- **Sintassi**

- dove l'operatore è un campo il cui nome inizia con \$

{ **<field1>: { <operator1>: <value1> }, ... }**

**\$and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ]**

- Operatore Logici
  - **L'AND** è implicito: due o più operatori o due o più campi nello stesso oggetto
    - \* `db.inventory.find( { qty: { $gte: 50, $lte: 150 } } )`
  - C'è comunque un operatore \$and
    - \* `db.inventory.find({ $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] })`
  - **OR** → `$or`
  - **NOR** → `$nor`
  - **NOT** → `$not`
- Per approfondire → <https://docs.mongodb.com/manual/>
- Esempio: Node.js e MongoDB
  - Per far connettere Node.js ad MongoDB, occorre installare il driver:
 

```
npm install mongodb - -save
```
  - per importare il modulo `mongodb`

```
const mongo = require('mongodb')
```
  - **async** specifica che la funzione è asincrona, quando viene chiamata, non viene eseguita immediatamente. Viene inserita nelle code degli eventi ed eseguita in modo asincrono dall'eventloop
  - Questo è necessario perché contattare il DBMS richiede tempo e viene fatto nello stesso modo, con la gestione a eventi
  - Perciò, i metodi per accedere al DBMS vanno chiamati con l'opzione **await** prima della chiamata
    - \* Sospende l'esecuzione di una funzione in attesa che la **Promise** associata ad un'attività asincrona venga risolta o rigettata. → viene fermata l'esecuzione e ripresa la funzione solo quando abbiamo un risultato
    - \* Le **Promise**, cioè oggetti il cui stato rappresenta lo stato di esecuzione di una attività asincrona.
    - \* Solo le funzioni **async** possono fare chiamate **await**
    - \* Ci permette di accedere al db:

```
const client = new mongo.MongoClient(db_url,  
                                     client_config)
```

```
db = await client.db(db_name);
```

- Otteniamo la collezione

```
var collection = await db.collection(collection_name);
```

- A questo punto, si possono usare i function method di MongoDB per fare le querye gli update
- Finito il lavoro, chiudere la connessione con **client.close()**;

## 10 Python

- È un linguaggio relativamente semplice, con **tipizzazione dinamica, interpretato e a oggetti**

- **Caratteristiche**

- Linguaggio interpretato
- Type checking dinamico
- Sintassi snella
- Le variabili non vanno dichiarate
- Le variabili sono puntatori/reference
- Programmazione a oggetti
- I tipi built-in sono classi, quindi i valori sono oggetti

- **Tipi delle variabili**

- Numerici: int, float, complex
- Liste: string, list, tuple
- Dizionari: mappe chiave valore

- Numeri

- `/` → quoziente
- `%` → resto
- `**` → potenza

- Classi/Costruttori/Casting

- `int(...)` → `a = int("12")`
- `float(...)` → `b = float(a)`
  - \* Il costruttore degli interi converte la stringa in intero
  - \* Il costruttore dei float converte l'intero in float

- **Numeri complessi**

`a = complex(1, 2)`

- `print(a)` →  $(1 + 2j)$
- `print(a.real)` → 1.0
- `print(a.imag)` → 2.0

- Oggetti

- **Immutable**

- \* Gli oggetti (perché le variabili puntano agli oggetti), una volta creati, non possono essere cambiati
    - \* Numeri, stringhe, tuple

- **Mutable**

- \* Gli oggetti possono cambiare il loro stato: esistono metodi di modifica
    - \* Liste, dizionari

- **Stringhe**

- Non hanno un costruttore
  - Semplicemente si assegna una stringa

**a = "Ciao"**

- Convertire un numero in stringa → **format**
  - \* Il metodo formatsostituisce le graffe con il valore dell'argomento

**b = "{}".format(12)**

- **Liste**

- Non hanno un costruttore
  - Sostituiscono i vettori/array
  - **x = [ 1, "b"]**
    - \* **x.append(3)** → aggiunge 3 in fondo al vettore
    - \* **print(x[2])** → mi visualizza 3
    - \* **print(x)** → la lista viene stampata con le quadre [1, 'b', 3]

- **Tuple**

- Ha il costruttore **tuple(...)**
  - Sono dette liste immutable
  - Creazione Tuple
    - \* **t = (1, "b")**
    - \* **t1 = tuple("a")**
    - \* **t2 = t + t1**
    - \* **print(t2) → (1, 'b', 'a')**
  - se si assegna **t1 = ("a")** → **t1** è una stringa

- **Stringhe, Liste, Tuple**

- Hanno in comune alcuni costrutti e funzioni
  - \* **len(x)** → lunghezza
  - \* **x[2]** → terzo elemento
  - \* **x[1:3]** → elementi da 1 a 2 (3 escluso)
  - \* **x[-1]** → ultimo elemento

- Esistono anche le collezioni di tipo **set** cioè senza ripetizioni

- **set** → insieme Mutable
- **FrozenSet** → insieme non Mutable

- **Dizionari**

- Sono contenitori chiave-valore
- Hanno il costruttore **dict()**, ma si può usare anche **{}**
- **Le costanti dei dizionari sono proprio dei documenti JSON**

```

• d1 = {}
d1["name"] = "Pippo"
d1["age"] = 25
d1["cars"] = [{"Model": "A"}, {"Model": "C"}]
d2 = {"name": "Pluto", "age": 30}

```

Name	Type	Size	Value
d1	dict	3	{"name": "Pippo", "age": 25, "cars": [{"Model": "A"}, {"Model": "C"}]}
d2	dict	2	{"name": "Pluto", "age": 30}

Figure 18: esempio

- **Puntatori/Reference**

- Liste e dizionari sono oggetti Mutable
- Esempio:
  - \* **l1 = [1, 2]**
  - \* **l2 = l1** → gli abbiamo passato il riferimento all'oggetto
  - \* **l1.append(3)**

Name	Type	Size	Value
l1	list	3	[1, 2, 3]
l2	list	3	[1, 2, 3]

Figure 19: esempio

- L’operatore + concatena le liste, quindi ne crea una nuova
  - \* l1 = [1, 2]
  - \* l2 = l1
  - \* l1 = l1 + [3] → viene creata un nuovo oggetto l2 ha il riferimento vecchio

Name	Type	Size	Value
l1	list	3	[1, 2, 3]
l2	list	2	[1, 2]

Figure 20: esempio

- Istruzioni generali
  - non si usano le parentesi ma il tab
  - La condizione è seguita da :
  - if, elif, else

```
a=2
if a==1:
    print("A")
elif a==2:
    print("Vale") L'azione
    print(a) di elif
else:
    print("Nessuno")
```

- for
    - \* non è un ciclo a contatore, ma a iteratore
- for variabile in lista:**
- ....

- **range**

- \* ritorna una lista con tutti i valori in un intervallo
- \* **range(from, to, step)**
  - from: valore iniziale (incluso)
  - to: valore finale (escluso)
  - step: incremento (opzionale, anche negativo)

```
for v in range(4, 1, -1):
    print(v)
    4
    3
    2
```

- **break:** esce dal ciclo
- **continue:** passa all'iterazione successiva
- **Definizione di Funzioni**

- \* possono essere definite funzioni annidate
- \* la funzione dentro è visibile solo dalla funzione fuori

```
• def nomefunzione( parametri ):
    codice (con tab)
• Esempio
def f(x, y):
    m = (x + y) /2
    return m

print( f(3, 6))
```

- **Visibilità delle variabili**

- \* una variabile nella funzione può coprire una variabile esterna

```
• Esempio:
pi = 'outer pi variable'           inner pi variable
def print_pi():
    pi = 'inner pi variable'
    print(pi)
print_pi()
print(pi)
```

```

• pi = 'outer pi variable'      outer pi variable
  def f():
    global internal_pi          inner pi variable
    internal_pi = 'inner pi variable'
  f()
  print(pi)
  print(internal_pi)

```

- \* Una variabile locale può essere dichiarata visibile a livello globale → si dichiara la variabile **global**

#### – Variabili Esterne Mutable

- \* Le variabili esterne non sono modificabili da una funzione
- \* Ma se il valore è un oggetto mutable, lo stato dell'oggetto può essere cambiato → **cambia il reference all'oggetto**

```

• elenco = []
  def f( valore ):
    elenco.append(valore)
  f(1)

```

Name	Type	Size
elenco	list	[1]

#### – Parametri Formali

- \* I parametri in coda possono avere un valore di default, quindi si possono non specificare

```

• Esempio:
• def f(a, b, c=3):
    return a + b + c
  print(f(1, 2))

```

- \* Se i parametri con valore di default sono multipli, si può scegliere a quali passare il valore

```

• Esempio:
def f(a, b, c=3, d=0, e=0):
    return a + b + c + d + e
print(f(1, 2, d=1, e=1))

```

- \* Il parametro **\*varargs** indica una lista anche vuota di parametri. **\*varargs** è una tupla

```

• def f(a, b, *varargs):
    s = 0
    for v in varargs:
        s +=v
    return a + b + s
  print(f(1, 2, 1,1))
  print(f(1, 2, 1,1, 3))

```

- **Classi**

- Classe archivio con 3 metodi: add\_nome, show, clean

```
class Archivio:
    def __init__(self):
        self.elenco = []
        return
    def add_nome(self, nome, eta):
        self.elenco.append({"nome": nome, "eta": eta})
        return
    def show(self):
        for e in self.elenco:
            print(e)
        return
    def clean(self):
        self.elenco = []
        return
```

- La parola chiave → **class** introduce la definizione della classe
- Il contenuto della classe è indentato dopo il :
- metodi sono definiti quasi come normali funzioni
- *obbligatorio il parametro **self** nei metodi, che fa riferimento all'oggetto*
- **Per creare il costruttore**

**def \_\_init\_\_(self, eventuali parametri):**

- Si può aggiungere i campi all'oggetto, con **self.campo**
- Ogni metodo può aggiungere campi all'oggetto
- ***È ammesso un solo costruttore***

```
nomi= Archivio()
nomi.add_nome("Pippo", 25)
nomi.add_nome("Pluto", 30)
nomi.show()
• Per creare l'oggetto, basta
chiamare il costruttore
{'nome': 'Pippo', 'eta': 25}
{'nome': 'Pluto', 'eta': 30}
```

- È possibile definire delle sotto-classi → **ereditarietà**
  - \* **class sottoclasse( superclasse):**
  - \* La sotto-classe eredita automaticamente campi e metodi

- \* La sotto-classe può fare l'**overriding** dei metodi ereditati
- **super()** fa riferimento alla super-classe
- Consente di chiamare il costruttore della super-classe  
 $\text{super()}\_\_\text{init}\_\_(\text{parametri})$
- **Esempio:**
  - \* Il costruttore riceve due parametri: **id**, **tipo**
  - \* Vi sono due metodi: **che\_tipo**, **get\_area**

```
class Figura:
    def __init__(self, id, tipo):
        self.tipo = tipo
        self.id = id
        return
    def che_tipo(self):
        return self.tipo
    def get_area(self):
        return 0
```

- \* Sotto-classe Rettangolo:
- \* Il costruttore chiama il costruttore della super-classe e inizializza il campo area
- \* Overriding di get\_area

```
class Rettangolo(Figura):
    def __init__(self, id, base, altezza):
        super().__init__(id, "Rettangolo")
        self.area = base * altezza
        return
    def get_area(self):
        return self.area
```

- **Eccezioni**
  - **try, except, else** (opzionale, codice da eseguire se non ci sono eccezioni)

```

op1 = int(input("Dividendo:"))
op2 = int(input("Divisore"))
try:
    n = op1 / op2
except ZeroDivisionError as err:
    print('Invalid operation ({})!'
          .format(err))
except ArithmeticError:
    print('Invalid operation!')
else:
    print("Risultato: {}", n)

```

- L'istruzione `raise` consente di generare le eccezioni

```
raise ZeroDivisionError('Impossibile dividere per 0')
```

- Si può anche propagare l'eccezione catturata

```

except ZeroDivisionError as err:
    raise err

```

- Si possono definire eccezioni, come sotto-classi della classe `Exception`

- Esempio

```

class MyException(Exception):
    pass

raise MyException("CIAO")

```

- L'istruzione «`pass`» consente di lasciare vuoti i blocchi annidati

MyException: CIAO

## • Garbage Collector

- Non esiste una de-allocazione esplicita

## 11 Python Moduli, Map-Reduce

- In Python si possono passare funzioni come parametri

```
def stampa(sum):
    print("Sum = {}".format(sum))

def main(a, b, callback = None):
    print("adding {} + {}".format(a, b))
    if callback:
        callback(a+b)

main(1, 2, stampa)
```

- **I Moduli**

- I moduli sono delle parti di codice contenute in un contenitore
- In Python, il modulo ha un nome e incapsula al suo interno variabili e funzioni
- L'interprete Python contiene molti moduli built-in
  - \* Questi sono già disponibili, non devono essere pre-installati
  - \* Per usare un modulo, si deve usare l'istruzione **import** per dire all'interprete di importarlo nel codice
- Il nome del modulo fa da **prefisso**
  - \* **nome\_modulo.nome\_funzione**
  - \* In questo modo, non si corre il rischio di conflitti tra moduli diversi
- Il modulo **gc** consente di gestire il garbage Collector
  - \* **gc.collect()**: fa partire la raccolta
  - \* **gc.enable()**: attiva il garbage collector (default)
  - \* **gc.disable()**: disabilita il garbage collector
  - \* **gc.isenabled()**: true se è attivo
- Un modulo built-in di Python consente di lavorare sul formato JSON
  - \* serializzare/deserializzare un dizionario come documento JSON e viceversa
  - \* **import json**

## – I File Handler

- \* Sono degli oggetti built-in che servono per gestire i file
- \* Per aprire il file in lettura:
  - `filein = open(stringa_nome_file)`
- \* Per aprire il file in scrittura:
  - `fileout = open(stringa_nome_file, "w")`
- \* `filein` e `fileout` sono file handler
- \* Per chiudere un file:
  - `filehandler.close()`
  - Ricordatevi di chiudere i file, perché fino a che il processo dell'interprete non è terminato, viene **mantenuto un lock sui file**
- \* Per leggere un file:
  - `string = filehandler.read()`
- \* Per leggere una riga per volta:
  - `string = filehandler.readline()`
- \* Si può iterare con `for` su file handler:
  - `for line in filehandler:`
- \* Per scrivere su un file:
  - `filehandler.write(testo)`
- \* Per leggere tutte le righe di un file:
  - `lista = filehandler.readlines()`
- \* con l'istruzione `with` non ci dobbiamo preoccupare di svolgere operazioni di chiusura perchè vengono gestite automaticamente tramite questa istruzione

## • Chiamate HTTP

- Occorre usare il modulo built-in `requests` → `import requests`
- Effettuare la chiamata con il metodo GET:
  - \* `res = requests.get(url, params)`
    - url: è l'indirizzo al quale inviare la richiesta
    - params: dizionario di parametri da accodare dopo il ?
- Effettuare la chiamata con il metodo POST:
  - \* `res = requests.post(url, data=None, json=None))`
    - url: è l'indirizzo al quale inviare la richiesta

- data: i dati da mandare (opzionale)
- json: dati json da mandare (opzionale)
- *Entrambi i metodi restituiscono un oggetto che consente di gestire la risposta*

- Creare un modulo

- *Un modulo è un normalissimo programma, che contiene solo funzioni, salvato in un file con estensione .py*
- L'istruzione **import** semplicemente importa il file (senza specificare l'estensione)

## Esempio

- Prepariamo un file «tcm.py»
- Contiene una sola funzione:  
`def tcm():
 return "Modulo TCM"`

## Esempio

- Creiamo un altro programma
- Importiamo e usiamo:  
`import tcm
print (tcm.tcm())`

**Modulo TCM**

- Map-Reduce

- E' un paradigma di programmazione, ossia un modo di programmare
- Pensato per:
  - \* Rendere paralleli gli algoritmi
  - \* Senza occuparsi del parallelismo

- Scrivere algoritmi paralleli non è facile, perché bisogna coordinare l'attività e lo scambio di dati tra i processori stessi
- Infatti la maggior parte del tempo dello sviluppo è dedicata allo scambio dei dati e al coordinamento dei processi
- Sono poche le persone in grado di scrivere algoritmi paralleli → **sviluppo costoso**
  - \* **nasce l'idea di alla base di Map-Reduce, ossia separare:**
    - l'algoritmo da eseguire in parallelo
    - dalle attività di scambio di dati e coordinamento → lasciando quest'ultimo compito ad un framework software dedicato
- Vantaggi per il programmatore
  - \* Lo sforzo di programmazione è rivolto agli aspetti algoritmici
  - \* Un algoritmo è composto da due soli tipi di attività primitive, cioè **Map** e **Reduce**, in genere di implementazione relativamente facile
  - \* Lo sviluppo diventa, quindi, molto veloce ed economico
- **Il Framework**
  - \* Distribuisce i task sui processori effettivamente disponibili
  - \* Distribuisce i dati da analizzare ai vari task
  - \* Raccoglie i dati prodotti dai vari task
  - \* Bilancia il carico di lavoro
- I Framework Map-Reduce lavorano su **cluster** di nodi di una rete di computer
  - \* i vari nodi non condividono la memoria centrale
  - \* Sono diventati famosi nel cloud computing
- Big Data e della Data Science ha bisogno della tecnologia Map-Reduce
  - \* il volume dei dati da trattare è troppo grande e non è possibile pensare di elaborarli con un approccio tradizionale
  - \* Serve la potenza di calcolo di molte macchine che lavorano in modo parallelo
- Più nodi abbiamo a disposizione meno tempo durerà l'elaborazione
  - \* L'approccio del cloud computing viene in aiuto:
    - Si attivano i nodi che servono

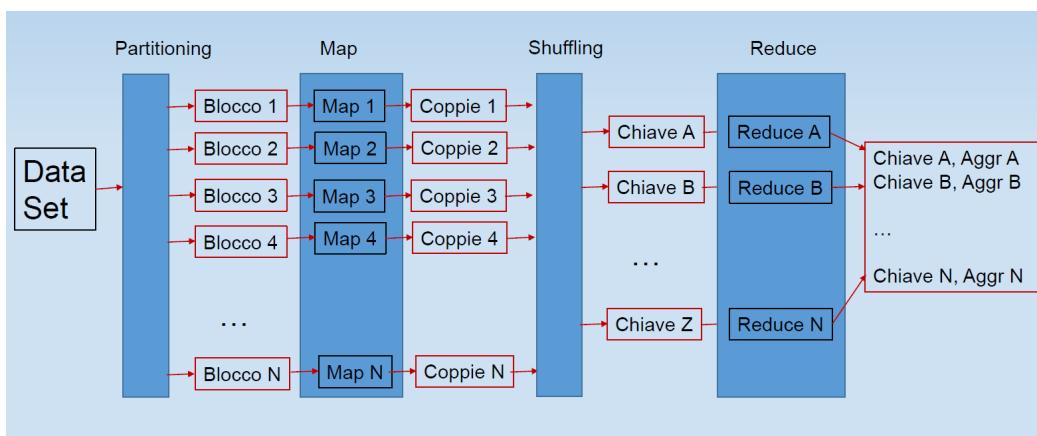
- Per il solo tempo in cui servono

- Il Paradigma Map-Reduce

- Primitive di base (da scrivere):
  - \* **Map:** un frammento del data set viene trasformato nella forma di coppie chiave, valore
  - \* **Reduce:** l'intero insieme di coppie chiave, valore viene raccolto, in modo da **aggregare tutte le coppie con lo stesso valore della chiave**

- Operazioni Svolte dal Framework

- **Partitioning:** il data set originale è partizionato in tanti blocchi relativamente piccoli
- **Map:** per ogni blocco di dati, un task di Map è eseguito (output: mappatura del blocco → coppie chiave, valore)
- **Shuffling:** Le coppie chiave, valore sono raccolte: per ogni chiave si forma un blocco con tutte le coppie
- **Reduce:** un task di reduce è eseguito per ogni valore della chiave



- Distribuzione dei task

- Nelle due fasi parallelizzabili Map e Reduce
- Un task di Map viene attivato per ogni blocco
- Un task di Reduce viene eseguito per ogni valore della chiave

- Il framework cerca di eseguire il maggior numero di task in parallelo, ma dipende dalle risorse disponibili e dal tempo di esecuzione di ogni task

- **Macro-Fasi Map-Reduce**

- Gli algoritmi Map-Reduce più semplici sono composti da una **sola macro-fase** Map-Reduce
- Ma si possono avere macro-fasi Map-Reduce multiple:
  - \* **Uguali**
  - \* **Diverse**

- **Modellare le Primitive**

- *Abbiamo quindi capito che l'attività di programmazione consiste nell'implementare le primitive Map e Reduce*
- Mentre l'attività di design consiste nell'identificare le macro-fasi Map-Reduce
- Conviene ragionare usando dello pseudo-codice
  - \* Un programma scritto in un linguaggio di programmazione infomale
  - \* Non rigorosamente vincolato alla sintassi e alla semantica
  - \* È prassi ispirarsi al Pascal

- **Esempio**

- Dato un testo, contare le occorrenze di ogni singolo termine

- **Quante macro-fasi servono?**

- **Una, perché:**

- Nella fase di partitioning, il testo viene spezzato in blocchi, senza spezzare le parole, per esempio 1000 parole per blocco
- La primitiva di map, per ogni parola nel testo, genera una coppia <termine, contatore> dove «contatore» è il numero di volte che il termine (parola) compare: al massimo, 1000 coppie
- La primitiva di Reduce riceve un insieme di coppie <termine, contatore>, tutte con lo stesso valore per «termine». Sommando tutti i valori di «contatore» (semplice ciclo for), produce una coppia <termine, somma>

- **Framework Principali**

- **Apache Hadoop**

- \* Componenti principali:
      - **YARN (Yet Another Resource Negotiator)**: è il componente software che gestisce la distribuzione dei task e dei dati sui nodi del cluster
      - **HDFS (Hadoop Distributed File System)**: Il file system distribuito di Hadoop, che viene usato per lo scambio dei dati
    - \* **HDFS**
      - è il componente che consente la distribuzione e la raccolta dei dati
      - Il data set da elaborare deve essere pre-caricato su HDFS
      - Tutti i blocchi da processare e le mappe *chiave, valore* vengono salvati su HDFS
      - **HDFS garantisce che vengano replicati e trasferiti su tutti i nodi della rete**
      - Soluzione efficiente, richiede poca memoria centrale sui nodi, ma lenta (a sincronizzazione del file system è un'attività costosa)

- **Apache Spark**

- \* Non è basato su un file system distribuito, come HDFS
    - \* Per contro, lavora esclusivamente in memoria centrale → **introducendo il concetto di RDD (Resilient Distributed Dataset)**
    - \* *Un RDD è un blocco di dati, che può essere utilizzato come input di un task oppure come output*
    - \* *Per ogni blocco, viene memorizzata la catena di elaborazione necessaria per calcolarlo*
    - \* **Resilient**: se serve fare spazio in memoria centrale, cancellando un RDD, rimane comunque il modo in cui è stato calcolato. Se serve il suo contenuto, la catena di elaborazione viene rieseguita
    - \* Lavorando in memoria centrale, è molto più veloce di Hadoop
    - \* con data set molto grossi, in relazione alla memoria centrale disponibile, il continuo ricalcolo degli RDD può rendere Spark più lento di Hadoop

- \* *Con Spark, non è necessario ragionare secondo il paradigma Map-Reduce*
- \* **Spark SQL** è un potente linguaggio di trasformazione di dati in forma tabellare
  - Il programmatore specifica le trasformazioni sulla tabelle (tipo algebra relazionale)
  - Il sistema le implementa come macro-fasi multiple di tipo Map-Reduce
- \* un'altro servizio costruito sopra il layer Map-Reduce consente di gestire flussi di **streaming**

## 12 BlockChain

- Problemi principali

- Fidarsi che il sistema centralizzato funzioni correttamente
- Andare a fidarsi di un intermediario che abbia il compito di regolare una transazione tra due parti
  - \* Svolgono operazioni che le parti non sono in grado di svolgere
  - \* Devono garantire la regolarità delle procedure
- **Il problema principale risiede della fiducia di un qualcosa che abbia il compito di tutelarci e garantirci un servizio**

- Risoluzione dei problemi

- Immutabilità:

- \* **la storia dell'intero processo** che si vuole rendere trasparente deve essere **archiviata in modo immutabile**, cioè non può essere manipolata né volontariamente né accidentalmente
    - \* Serve **un Ledger, un registro** dove si annotino tutti i passaggi in modo immutabile

- Condivisione:

- \* **il registro deve essere condiviso**, cioè devono esistere molte copie continuamente allineate e gestite da sistemi in **competizione/distribuiti** tra loro
    - \* **L'autorizzazione a cambiare i dati**, cioè a registrare una nuova versione di un certo processo (mantenendo la precedente) deve essere **data da molti, non da pochi (consenso condiviso)**

- Le monete virtuali

- Non esiste fisicamente
- Esiste perché c'è un sistema informatico che la gestisce:
  - \* Certificare il possesso di importi della valuta da parte degli utenti
  - \* Gestire le transazioni, cioè lo scambio/trasferimento di importi da un utente ad un altro
  - \* Evitare le frodi
  - \* Evitare la perdita dei dati

- Le monete virtuali sono comunque legate alle monete reali, perchè per possedere monete virtuali → **devi pagare con monete reali**
- Quindi la perdita di informazione da parte del sistema provoca la → **perdita soldi veri**
  - \* Per questa ragione, fino al 2009, le monete virtuali hanno avuto scarso successo

- **bitcoin**

- Nel 2010 nasce il bitcoin
- Il successo di bitcoin risiede nella tecnologia su cui si basa la piattaforma Bitcoin(in maiuscolo)
- La piattaforma Bitcoin gestisce gli scambi di bitcoin (la moneta, in minuscolo) e certifica il loro possesso
- Bitcoin è basata sulla tecnologia *BlockChain*

- **Peer-to-peer Network**

- Bitcoin è una **peer-to-peer network**
- Non esiste un master o controllore
- Tutti i peer o nodi partecipano al processo, dando il loro **consenso** all'operazione
- La distribuzione del consenso serve per non creare singole vulnerabilità

- **BlockChain**

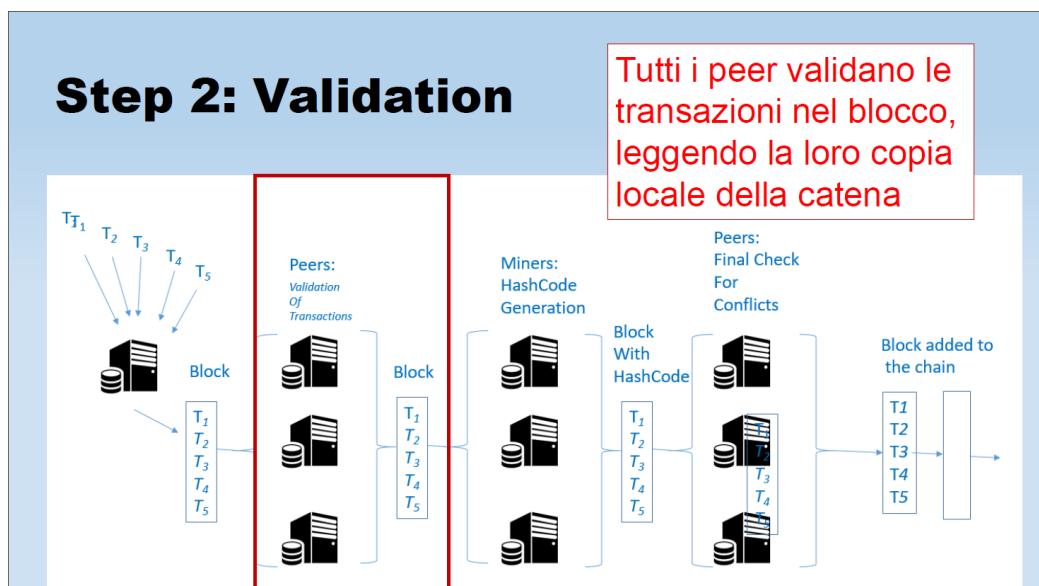
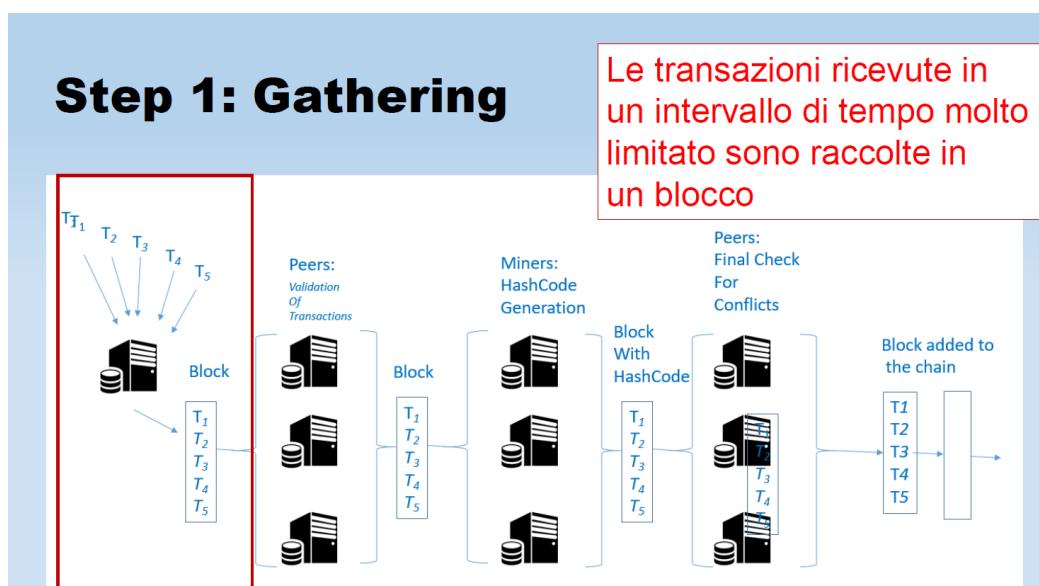
- *BlockChain* significa catena di blocchi
  - \* Denota il modo in cui i dati vengono memorizzati, al fine di essere immutabili
  - \* Un blocco contiene un gruppo di **transazioni** avvenute più o meno contemporaneamente
  - \* Ogni blocco punta al blocco precedente, come in una lista
  - \* Quando si deve inserire un nuovo blocco, questo diventa la nuova testa della lista
- *Per ottenere l'immutabilità, occorre usare il meccanismo degli HashCode*

- \* Un hashcode è un codice che viene generato da una funzione matematica → **viene utilizzato per identificare un blocco**
- \* **L'identificatore/hashcode del blocco è generato a partire dal contenuto del blocco stesso**, in base ad un meccanismo di cifratura
- \* Ogni blocco contiene al suo interno l'hashcode del blocco precedente
- \* *Quindi l'hashcode del blocco, oltre che dal contenuto stesso dipende anche dal hashcode del blocco precedente*
- \* **un tentativo di alterare la catena, cambiando il contenuto del blocco o il riferimento al blocco precedente renderebbe non più valido l'hashcode del blocco stesso**
- \* hashcode deve soddisfare dei requisiti:
  - debba avere una lunghezza elevata
  - un certo numero di bit deve essere uguale a zero
  - La funzione di cifratura usata si basa su un numero casuale, detto **chiave di cifratura**
  - *Si apre una sfida:* esiste una chiave di cifratura che permette di generare un hashcode che rispetta i vincoli?  
→ *Mining*

### • Mining

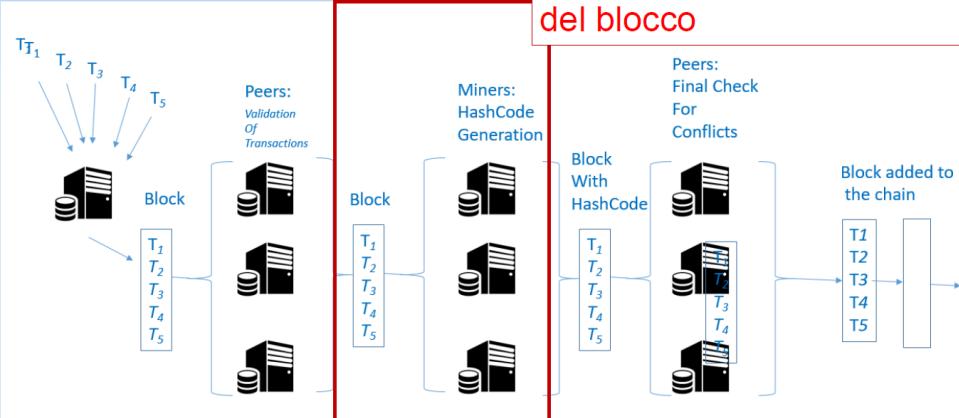
- *La sfida viene chiamata mining, cioè scoprire la chiave di cifratura*  
→ **è molto impegnativa**
- Per un solo computer, potrebbero essere necessari anni per trovare la chiave di cifratura
- Se **esistono moltissimi miners**, cioè molti computer che portano avanti la sfida, generando in modo casuale le chiavi di cifratura, allora la probabilità di risolvere la sfida in tempi brevi aumenta notevolmente
  - \* Appena un miner trova la chiave, comunica a tutti i nodi l'hashcode e la chiave di cifratura
  - \* Gli altri nodi verificano che sia corretta
- *Così facendo, il consenso ad aggiungere il nuovo blocco alla catena è distribuito*

- anche ammesso che un nodo della rete sia in grado di generare un blocco con transazioni fraudolente, non avrebbe mai la capacità di generare un hashcode che rispetta i vincoli in tempi ragionevoli
- Il meccanismo che abbiamo appena visto si chiama *Proof of Work*
  - comunica a tutti i nodi e deve avere l'approvazione da parte di tutti



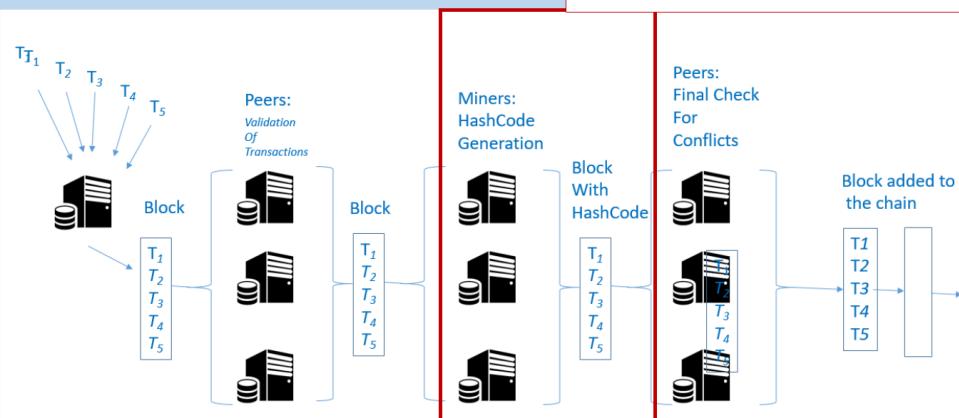
## Step 3: Hash-Code Generation

Si deve generare un Hash Code univoco, come identificatore univoco del blocco



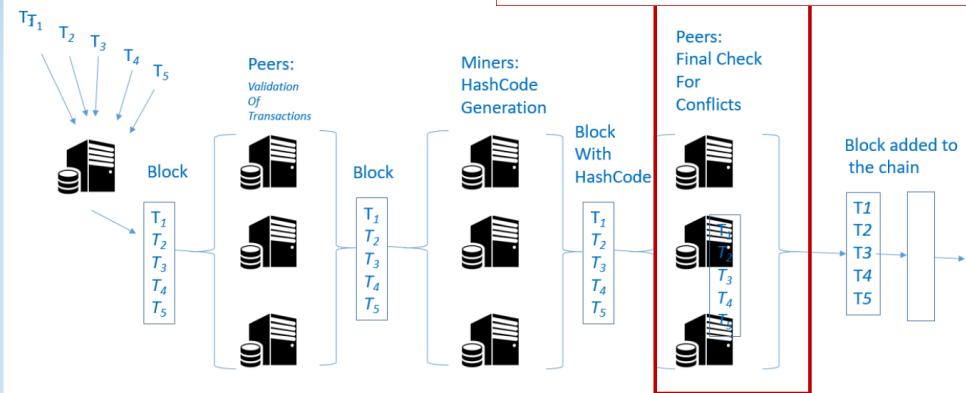
## Step 3b: Hash-Code Trovato

Un miner trova l'hash code e lo manda ai peer per validerlo



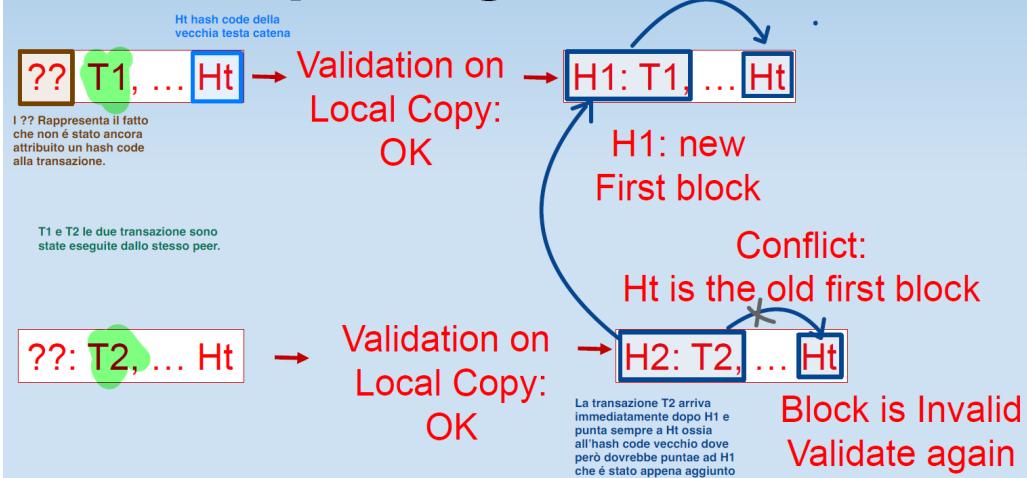
## Step 4: Checking for Conflicts

I peer validano l'hash cpde e rendono il blocco il nuovo primo blocco, se quello vecchio non è cambiato



- come si evita il double spending?
- Il conflitto si verifica perché uno dei due blocchi processati in parallelo arriva tardi
- Entrambi puntano allo stesso vecchio primo blocco della catena
- Il primo che arriva, viene inserito
- Il secondo, quando arriva, punta ad un blocco che non è più il primo della catena

## Double-Spending Transactions



- Le transazioni nel blocco risultato non valido vengono validate di nuovo → *La transazioni fraudolente vengono rifiutate*
- **Problema:**
  - \* L'approccio funziona, però richiede un grande sforzo computazionale
  - \* È abbastanza lento (circa 10 minuti per transazione)
  - \* Si consuma tanta energia elettrica

- **Smart Contract**

- Un contratto tra due o più parti, che non richiede l'intervento umano per essere portato avanti
- Si tratta di un programma che contiene le regole di correttezza e le regole di trasformazione dei dati associati al contratto
- *La BlockChain garantisce l'integrità temporale di tutti gli stati del contratto*
- Ma se le operazioni sul contratto non vengono validate da un essere umano, il contratto può **perdere la sua validità legale**
- Esistono **smart contract che non hanno bisogno di validità legale e altri che ne hanno bisogno**
- ***Supporto agli Smart Contract***
  - \* **InternalCode:** Il codice è memorizzato ed eseguito all'interno della piattaforma
    - **Contract-Specific Code:** il codice non viene condiviso
    - **Contract-Family Code:** una famiglia di contratti condivide il codice (template)
    - **Global Code:** il codice è globale, può operare su tutti i dati
  - \* **ExternalCode:** Il codice è memorizzato ed eseguito al di fuori della piattaforma

- **Accesso alla Piattaforma**

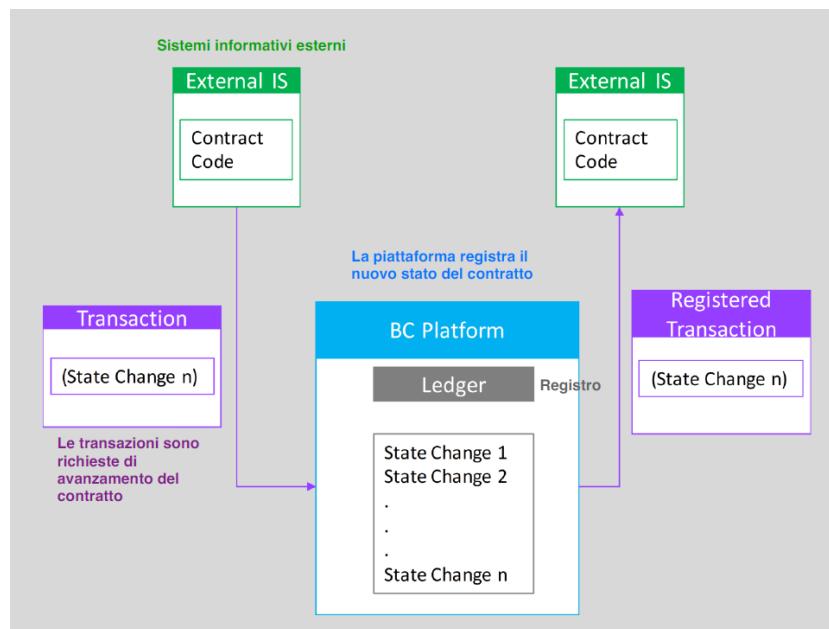
- **Permissionless BlockChain:** Chiunque può entrare nella rete, basta installare il software necessario e rispettare le regole di comportamento

- **Permissioned BlockChain:** Solo i peer autorizzati possono entrare, c’è un amministratore della rete che concede le autorizzazioni

- **BlockChain più famose**

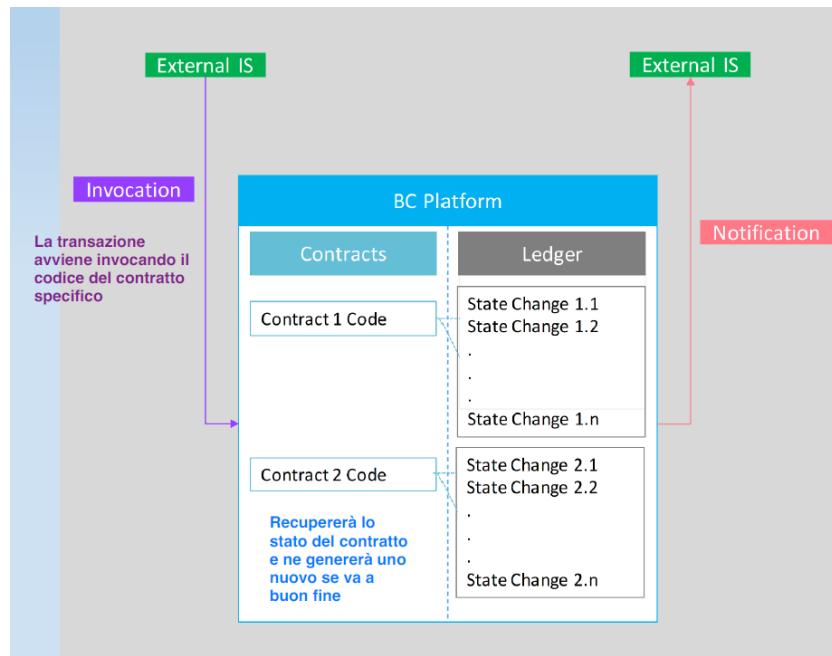
- **Bitcoin:**

- \* Permissionless platform
- \* Moneta virtuale: bitcoin
- \* Nessun supporto per Smart Contracts(eseguiti al di fuori della piattaforma)
- \* External code
- \* Meccanismo di consenso: Proof of Work



- **Ethereum:**

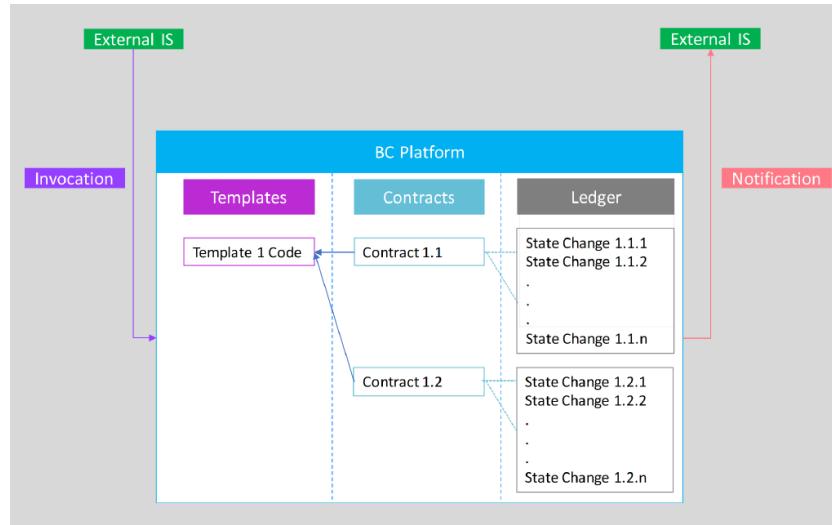
- \* Permissionless platform
- \* Moneta virtuale: Ether
- \* Supporta gli Smart Contracts
- \* Linguaggio di programmazione: Solidity
- \* Internal code, In-platform Code, Contract-specific code
- \* Meccanismo di consenso: Proof of Work



- \* Gli smart contract possono lavorare anche senza la moneta virtuale
- \* Il codice viene eseguito da tutti i peer, durante il processo di validazione

– Corda:

- \* Permissioned platform
- \* No virtual currency
- \* Supporta gli Smart Contracts: Legal prose version
  - Cioè uno smart contract DEVE avere una versione testuale
  - dei termini del contratto
  - delle operazioni svolte sul contratto
  - In questo modo, il contratto mantiene validità legale a tutti gli effetti
- \* Linguaggio di programmazione: Java, Kotlin
- \* Internal code, In-platform Code, Contract-family code
- \* Meccanismo di consenso: Proof of Work



- \* Gli utenti possono creare famiglie di contratti (contract template) che si comportano nello stesso modo, ma cambiano per i dettagli del contratto
- \* Esempio: mutui, Importo finanziato, Tasso di interesse, Numero di rate

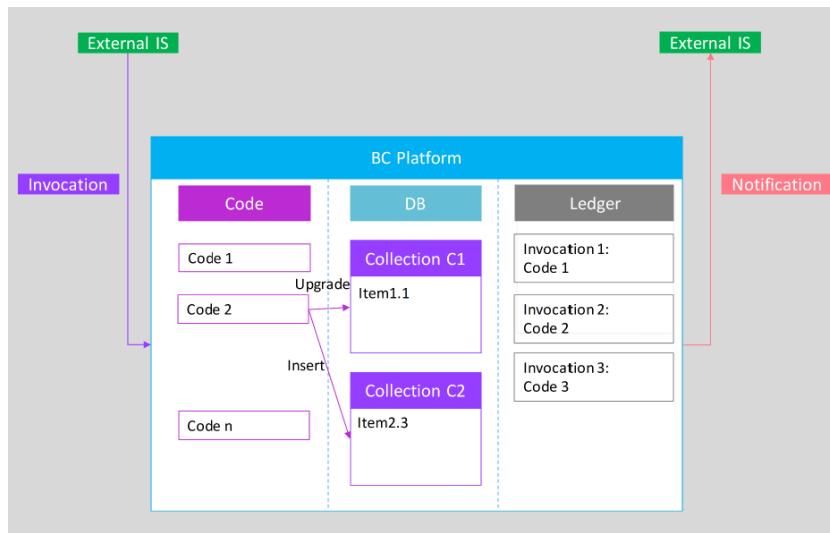
### • Proof of Knowledge

- A differenza del Proof of Work, **basa il consenso sulla conoscenza di ciò che avviene**
- Non tutti i peer vengono coinvolti, ma solo alcuni
- *pochi peer coinvolti equivale ad avere bisogno di poche risorse e quindi si ottiene maggiore velocità*

### • HyperLedger Fabric

- L'obiettivo: sviluppare piattaforme di tipo permissioned
- creare dei database distribuiti e condivisi da sistemi informativi che devono cooperare
- Il prodotto più famoso: **HyperLedger Fabric**
- Caratteristiche
  - \* **Permissioned platform**
  - \* **No Moneta Virtuale**, fornisce il concetto di database distribuito
  - \* **No smart contracts, ma ChainCode**

- \* **In-platform Code, Global Code**
- \* Linguaggi di programmazione: java, JavaScript, Go
- \* Meccanismo di consenso: di tipo Proof of Knowledge, Byzantine Fault Tolerant (BFT) consensus mechanism



- \* Il database è un NoSQL database, è un JSON Document Store → CouchDB
- \* Le query possono essere fatte direttamente sul db
- \* *Il Ledger fa da log del DB: registra tutte le operazioni che hanno portato allo stato corrente del DB*

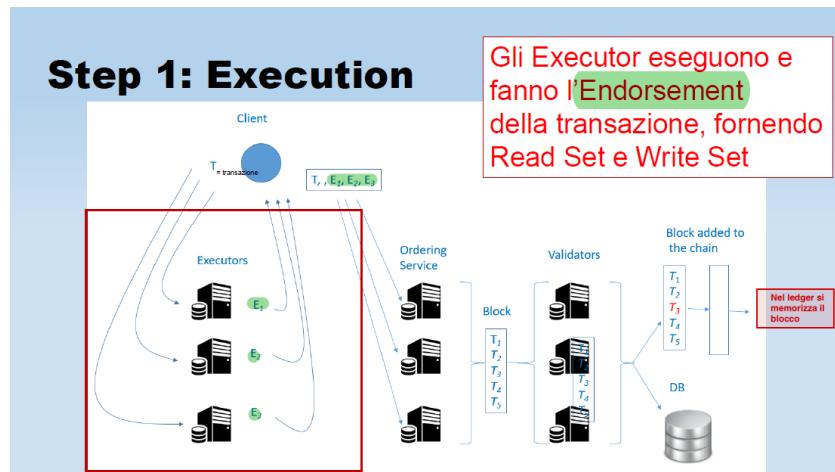
- **Meccanismi di consenso:**

- Proof of Work
- Proof of Knowledge
- Byzantine Fault Tolerant Mechanism

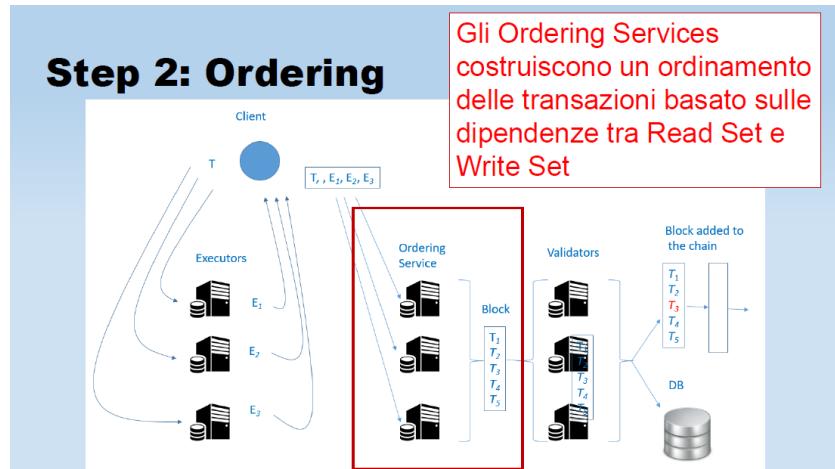
- **Byzantine Fault Tolerant Mechanism**

- Si gestisce la transazione, verificando i conflitti a posteriori, lavorando sulle dipendenze tra gli aggiornamenti
  - \* **Read Set:** insieme degli oggetti (JSON) modificati dalla transazione
  - \* **Write Set:** insieme dei nuovi oggetti generati dalla transazione

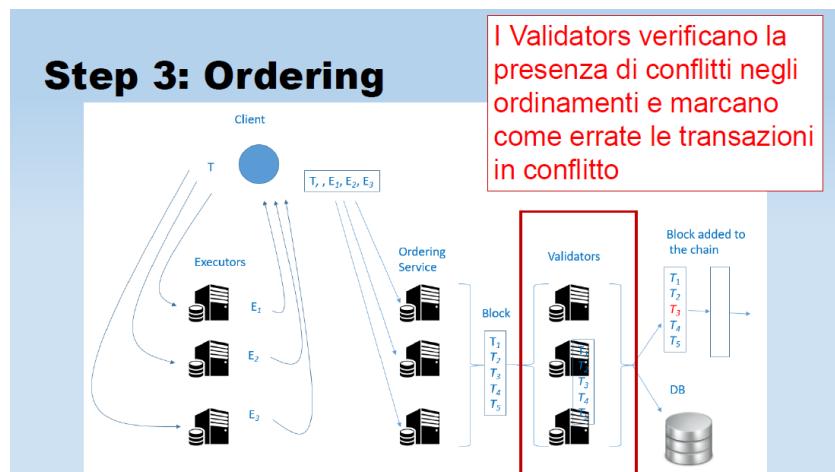
## Step 1: Execution

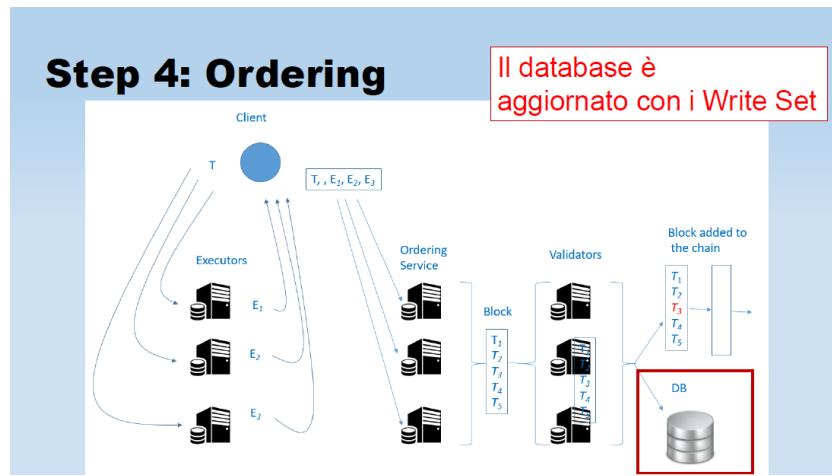
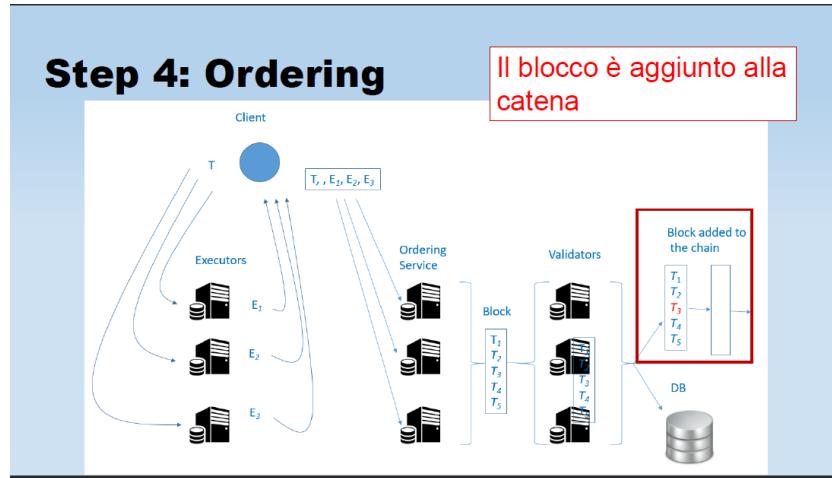


## Step 2: Ordering



## Step 3: Ordering





- **Vantaggi:**
  - \* Questo meccanismo di consenso richiede limitata capacità computazionale → ottiene velocità molto alte
  - \* Ma per processare transazioni di carte di credito, è ancora troppo alto
- **Conclusioni**
  - \* La tecnologia BlockChain ha dimostrato di essere molto interessante
  - \* Ma è lenta
  - \* E nella versione permissionless consuma troppa energia

## 13 Micro-Services e ReST

- Il Processo di Deploy

- **Ambiente di produzione:** I sistemi che forniscono il servizio agli utenti.
- **Ambiente di sviluppo:** I sistemi sui quali si sviluppa il nuovo software.
- **Ambienti di test:** I sistemi sui quali si effettuano i test.



- Ogni passaggio di ambiente si chiama **Deploy**
- Il Deploy in produzione è il più critico perché un errore di installazione causa malfunzionamenti verso gli utenti finali
- Negli ambienti di test si creano delle copie integrali dei dati nei necessari → Alcuni sistemi chiamano gli ambienti di test **sandbox**

- Il Deploy sulle PaaS Cloud

- PaaS deve fornire **supporto allo sviluppo, testing e deploy**
- Con la gestione integrata del **Versioning**
- la PaaS consente di gestire le dipendenze tra parti del software ed automatizzare il processo di deploy
- Il processo di deploy NON DEVE essere effettuato a mano!

- Modularizzazione del Software

- Per controllare la complessità dello sviluppo, è bene **modularizzare il software**
- Suddividere in **moduli/componenti** con ciascuno il proprio compito

- Come avviene la suddivisione:

- **Layler**

- \* Si tratta di applicazioni web che si basano sul paradigma **3 tier architecture**
      - **View:** HTML + JavaScript
      - **Controller o Server side:** programmi lato server che controllano la dinamica e realizzano la business logic dell'applicazione
      - **Model o Data Layer:** il database che raccoglie i dati che rappresentano il modello della realtà su cui si opera
    - \* **Vantaggi**
      - suddivisione per competenze tecniche, cioè il team dell'interfaccia ha il controllo sull'intera applicazione (omogeneità)
    - \* **Svantaggio**
      - team sono grossi e devono continuamente parlarsi tra di loro e tutti con gli utenti finali: elevato tempo di coordinamento necessario

- **sistema monolitico**

- \* Può essere eseguito in un unico processo di sistema
    - \* Le varie parti del sistema vengono sviluppate in base alle necessità dei programmatore, cercando di riusare il codice per quanto possibile
    - \* al crescere della complessità del sistema diventa sempre più difficile fare **modifiche**, perché queste possono avere un **impatto molto diffuso**

- **Modularizzazione verticale**

- \* Si creano i **moduli**
      - In base alla sensibilità e alle necessità dei programmatore
      - Con visione in **piccolo/grande** di funzionalità all'interno
    - \* Un modulo è un insieme di funzioni (e di funzionalità) in qualche modo omogenee tra di loro
    - \* Il concetto di namespace aiuta a evitare i conflitti tra moduli diversi integrati nella stessa applicazione
    - \* Le **Librerie** sono un'evoluzione del concetto di modularizzazione

- **Modularizzazione in Piccolo**

- La modularizzazione in piccolo non risolve i problemi dei sistemi monolitici, perché anche se modularizzati al loro interno rimangono monolitici
- Si cerca di riutilizzare il più possibile il codice, generando **forti indipendenze**
  - \* la forte dipendenza rende molto difficile modificare parti anche piccole del software, perché le modifiche hanno impatto, in cascata

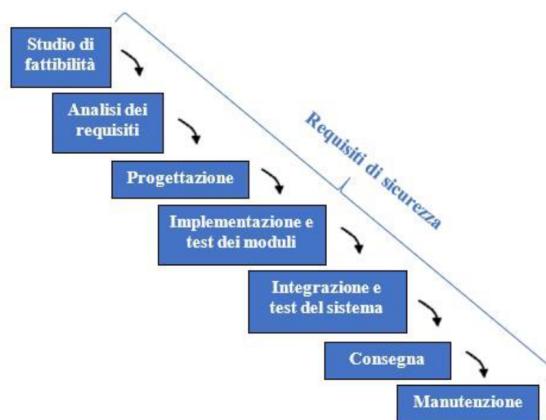
- **La Dimensione dei Team**

- La dimensione dei team di sviluppo è un aspetto non trascurabile
- più un team è grosso, più è rigido e di difficile controllo

- **Cicli di Vita del Software**

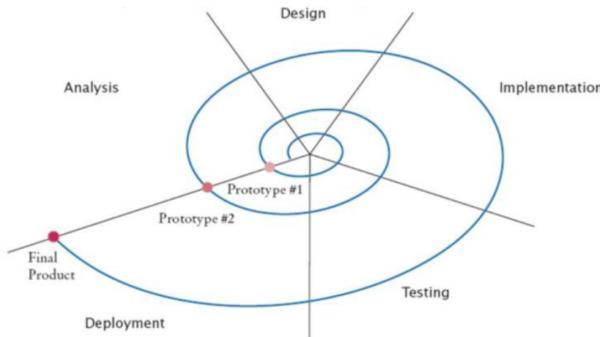
- **A Cascata**

- \* Dalla Specifica dei requisiti alla consegna, senza ripensamenti, poi si fa la manutenzione del software
- \* Nell'approccio a cascata, dopo il rilascio, il team si disinteressa del sistema
- \* Che tipicamente viene preso in carico da un altro team, che non ha partecipato al sviluppo, **quindi fa fatica a tenerne sotto-controllo la complessità e, di conseguenza, a fare le modifiche**



### – A Spirale

- \* Lo sviluppo avviene per affinamenti successivi, ridefinendo le specifiche



### – Aproccio Agile

- \* Evolve il modello a spirale, organizzando il team in unità molto piccole, ma con responsabilità ben precise

## • Scalabilità

- Rappresenta un problema frequente
- Con scalabile si intende la possibilità di adattarsi a carichi di lavoro crescenti
- Il cloud computing è una soluzione, perché consente di aumentare le risorse di calcolo a disposizione con estrema facilità
- Le fonti della **non scalabilità** possono essere diverse
  - \* Codice scritto male
  - \* Strutture dati inefficienti
  - \* Algoritmi intrinsecamente non scalabili
  - \* Funzionalità ridondanti

## • SOA: Service-Oriented Architecture

- L'idea è di organizzare il sistema per composizione di sotto-sistemi
- Ogni sotto-sistema fornisce servizi ben definiti
- Richiamabili attraverso meccanismi di comunicazione: RPC, Protocollo SOAP, Web Services con HTTP, Message Queuing
- Purtroppo l'approccio SOA ha portato alla realizzazione di **sotto-sistemi monolitici**

- **Problema:** Nell'approccio allo sviluppo e nella dimensione dei team
- Dall'esperienza pratica, nasce il concetto di **Micro-services**

- **Microservices**

- Cioè un modo di organizzare l'architettura del sistema informativo  
→ **pattern architettonico**
- *organizzare il sistema come una miriade di sotto-sistemi, totalmente indipendenti, che comunicano tra di loro attraverso la rete*
- **Caratteristiche**
  - \* **Molti stack**
    - Ogni servizio contiene più o meno l'intero stack tecnologico necessario per realizzarlo
  - \* **Molti server**
    - Ogni servizio viene eseguito su un server (o gruppo di server) specifico per il servizio. In questo modo, invece di avere un unico **punto di failure**, se ne hanno tanti
  - \* **Tolleranza ai guasti**
    - Il fatto che un servizio si basi su altri servizi, costringe il programmatore a gestire l'eventuale failure dei servizi su cui si appoggia
  - \* **Logginge Monitoring**
    - I servizi devono continuamente registrare le attività svolte e i malfunzionamenti. Occorre predisporre un servizio di **monitoring**, per tenere sotto controllo lo stato dell'intero sistema
  - \* **Chiara suddivisione dei compiti svolti dai vari micro-servizi**
    - Ogni micro-servizio deve avere un compito ben preciso da svolgere
  - \* **Chiara definizione dei confini del micro-servizio**
    - Il system architect definisce in modo chiaro e preciso i confini del micro-servizio
  - \* **Chiara definizione delle interfacce (API) del micro-servizio**

- L'interfaccia fornita da ogni micro-servizio deve essere ben documentata, perché è attraverso questa interfaccia che gli altri micro-servizi lo usano

\* **Deploy indipendente**

- Ogni micro-servizio è sviluppato in modo **indipendente** dagli altri, su server indipendenti

\* **Interoperabilità**

- Vista l'indipendenza dei microservizi è possibile **sostituire un servizio con uno equivalente, senza che il resto del sistema se ne accorga/risenta**

– **Vantaggi**

\* **Manutenibilità**

- Il codice di un micro-servizio risulta (deve risultare) piccolo, facile da modificare

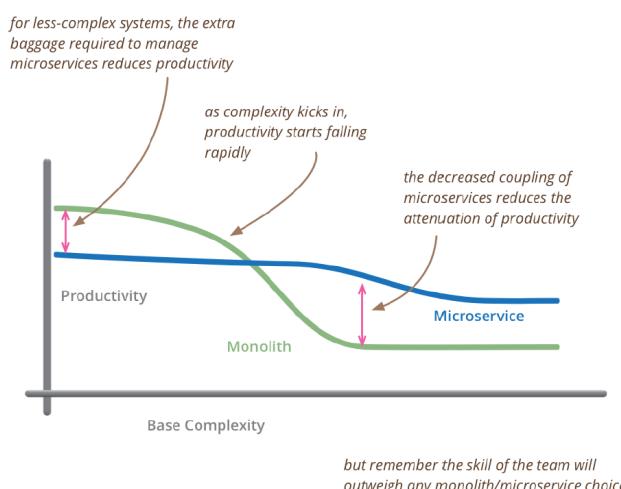
\* **Reimplementazione dei servizi**

- Se un servizio non fornisce prestazioni soddisfacenti, può essere facilmente modificato o sostituito con uno più performante

\* **Scalabilità**

\* **Riduzione dei costi**

- Se un servizio non fornisce prestazioni soddisfacenti, può essere facilmente modificato o sostituito con uno più performante

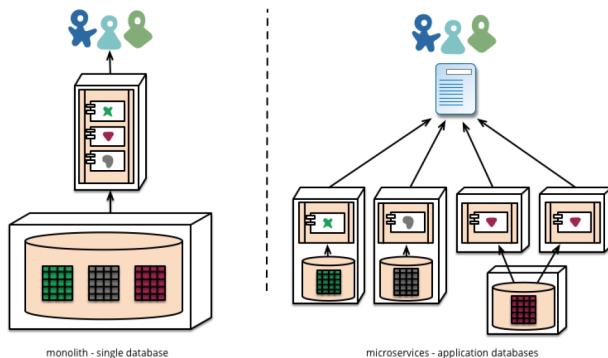


\* **Facile aggiunta di funzionalità**

- Aggiungere funzionalità al sistema informativo vuol dire aggiungere alcuni micro-servizi, da integrare modificando alcuni micro-servizi di coordinamento

### – Svantaggi

- \* **Maggiori costi iniziali**
- \* **Tecnologie eterogenee**
  - Servizi diversi possono essere realizzati con tecnologie diverse. Si sceglie la migliore tecnologia per ogni micro servizio
- \* **Molti data store**
  - Ogni micro-servizio può avere il suo (o i suoi) data store.
  - 1. Si ha più database
  - 2. Si creano ridondanze
  - 3. Le informazioni presenti devono essere consistenti con tutte le altre



### – Aspetti Organizzativi

- \* Si creano piccoli team, di poche persone
- \* Ogni team ha la piena responsabilità di gestire alcuni micro-servizi, per tutta la loro vita operativa
- \* **grandezza team:** approccio Amazon, **Two Pizza Team**
  - Una pizza americana è per 6 persone, quindi 2 pizze fanno mangiare 12 persone
  - In Amazon, i team non superano le 12 persone e coprono tutte lo stack tecnologico
- \* **grandezza team:** Altro approccio, **Half dozen, Half dozen**
  - un team di una mezza dozzina di persone gestisce una mezza dozzina di servizi

- \* **grandezza team: Approccio Agile**
- \* Riduzione dei costi di coordinamento
  - Le necessità di coordinare i team tra di loro si riducono di molto
  - Riducendo, di conseguenza, i costi impliciti dell'attività di coordinamento

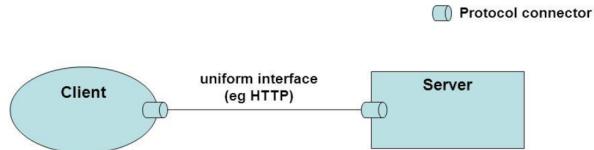
- **Come Invocare un Micro-Service**

- \* Si preferisce utilizzare **HTTP (o HTTPS)**
- \* Perché la soluzione migliore è che i micro-servizi siano realizzati secondo lo stile **architetturale ReST**

- **ReST (Representational State Transfer)**

- È uno stile architetturale. Un modo di organizzare l'architettura con cui i servizi sono definiti
- Un servizio **ReSTful** o architettura **ReSTful** indicano che il servizio o l'architettura rispettano i vincoli/principi definiti dallo stile ReST
- I vincoli/principi di ReST sono focalizzati sul modo in cui i dati vengono trasmessi
- **Principi ReST**
  - \* **Un architettura ReST è client-server.**

## Constraint 1: Client-Server



**Constraints:**

1. Client-Server

Separation of concern

- \* **State-less**

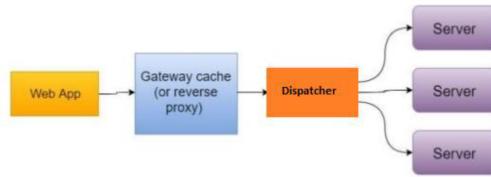
- La comunicazione deve essere senza stato, cioè non deve essere basata sullo scambio di messaggi multipli tra client e server. *Il lavoro richiesto al server si esaurisce con la risposta alla richiesta*

- \* **Cache**

- Questo consente di sfruttare la memoria cache dei browser o dei proxy per non rieseguire le richieste
- \* **Interfaccia Uniforme**
  - messaggi scambiati tra client e server devono essere **uniformi, ossia basati su un *formato standard***
- \* **Interfaccia Uniforme: Risorse**
  - Una risorsa è un oggetto o la rappresentazione di qualcosa di significativo nel dominio applicativo. È possibile interagire con le risorse attraverso le API.
- \* **Interfaccia Uniforme: Manipolazione attraverso Rappresentazioni**
  - La stessa risorsa può essere rappresentata in molti modi: XML, JSON, PNG
  - Il servizio può fornire rappresentazioni diverse per la stessa risorsa
- \* **Interfaccia Uniforme: Hypermedia come motore dell'applicazione**
  - Le azioni sulle risorse sono guidate da link, presenti nella rappresentazione delle risorse stesse

```
<album>
<title>the title</title>
<code>1234</code>
<description>A new piece of art</description>
<link rel="/artist" href="/artist/blackMen"/>
<link rel="/purchase" href="/purchase/1234"/>
</album>
```

- **Ecco perchè ReST:** il server trasferisce al client la rappresentazione dello stato della risorsa, con associati i link che descrivono tutte le azioni possibili che si possono effettuare sulla risorsa stessa
- **L'approccio ReST è l'ideale nei sistemi basati su micro-services** perchè fornisce la rappresentazione di una risorsa con le azioni possibili su di essa e i link ai micro-servicesche le eseguono
- **Il dispatcher** si occupa di indirizzare la richiesta effettuata verso il servizio / server appropriato



- **XML o JSON?**

- XML fornisce di suo il concetto di link
- JSON
  - \* Il campo ”\_links” contiene tutti i link associati all’album, dove il nome del campo descrive il tipo di azione possibile

```

{
  "type": "album",
  "title": "the title",
  "code": "1234",
  "description": "A new pieve of art",
  "_links": {
    "artist": "/artist/blackMen",
    "purchase": "/purchase/1234"
  }
}

```

## 14 XML Schema e WSDL

- XML Schema è nato per sostituire il DTD
- Il prefisso del namespace è **xs**

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
</xs:schema>
```

- Se l'elemento da definire ha un contenuto solo testuale, la sua definizione è molto semplice

```
<xs:element name="orderperson"
  type="xs:string"/>
```

- Con **type** si può definire il tipo del contenuto, in questo caso è una stringa
  - xs:positiveInteger
  - xs:integer
  - xs:decimal
  - xs:string
  - xs:date
  - xs:time
  - xs:dateTime

• Esempio:

```
<xs:element name="title"
  type="xs:string" />
```

Elemento che contiene solo testo. Esempio:

```
<title>My Title</title>
```

- Si definisce un tipo complesso. Cosa può esserci dentro:

- **xs:sequence**: definisce sequenze di elementi
- **xs:all**: indica gli elementi che devono essere presenti, indipendentemente dall'ordine
- **xs:choice**: Uno solo degli elementi indicati può occorrere nel contenuto

```
<xs:element name="shiporder">
  <xs:complexType>
    ...
  </xs:complexType>
</xs:element>
```

- Per indicare il numero di occorrenze, prevede due attributi opzionali
  - **minOccurs**: valori "0" o "1"
  - **maxOccurs**: valore non negativo o "unbound"
- **Riuso di Elementi**

```
<xs:element ref="nome"/>
```

- **Definizione di Attributi**

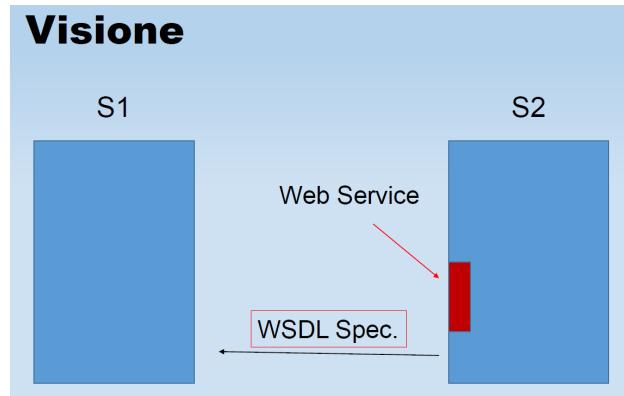
- <xs:attribute name=... typw=... use=.../>
- **use** è opzionale e vale:
  - "optional"
  - "required"

- **Riuso di Attributi**

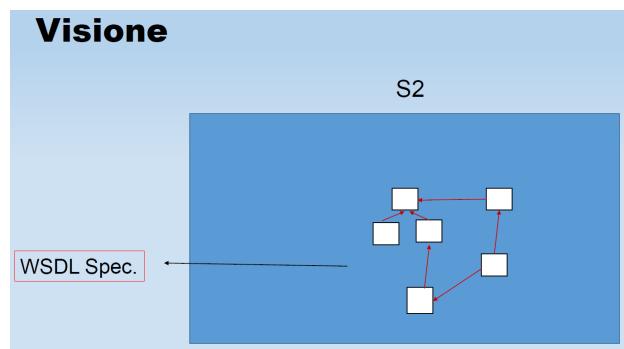
```
<xs:attribute ref="nome"/>
```

- un documento viene detto valido rispetto alla specifica XSD
  - Come validare? Con i parser validanti XML Schema XML-Schema Validating Parser
- **WSDL (Web Service Definition Language)**
  - Lo scopo è definire in modo formale un web service

- Conseguenza: sviluppo di librerie e strumenti per effettuare la comunicazione in modo automatico, con poche linee di codice

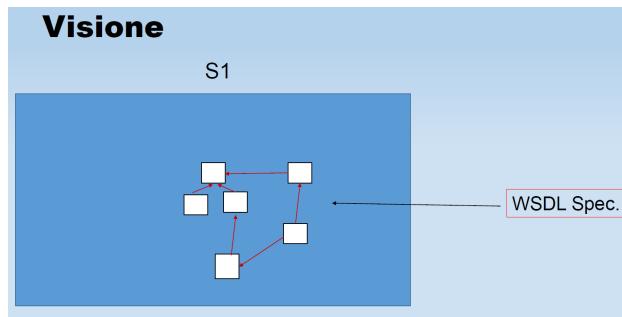


- Ma chi scrive la specifica WSDL? **nessuno**
  - \* Viene generata automaticamente dalle strutture dati **object-oriented** che devono essere popolate con i dati ricevuti dal web service
  - \* Partendo o dalle definizioni delle classi o dalle definizioni del database

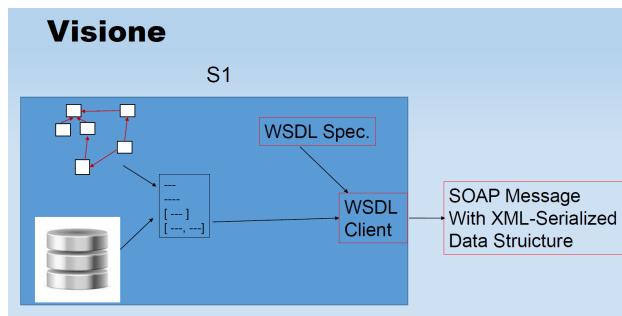


- **Funzionamento:**

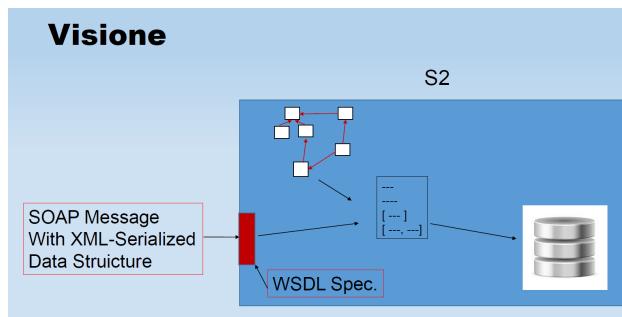
- S1 riceve la specifica WSDL
- I programmati di S1 popolano la struttura dati con i dati da inviare
- Usando le definizioni ottenute dalla specifica WSDL



- Il client WSDL in S1 serializza la struttura dati, usando la specifica WSDL per generare il contenuto XML del messaggio



- Il web service esposto da S2 riceve il messaggio
- Usando la specifica WSDL deserializza il messaggio
- Popola la struttura dati interna a S2, nel suo linguaggio di programmazione
- Questa struttura dati verrà utilizzata dal resto del codice di S2



- **Caratteristiche fondamentali:**

- *La specifica WSDL è basata su XML*

- Fa parte di un namespace specifico
- Definisce i contenuti dei messaggi e come comunicare con il web service esposto
- *Il contenuto dei messaggi è definito tramite una specifica XSD*
- *Il protocollo usato per inviare i messaggi è HTTP*
- Il lato server del web service si appoggia ad un HTTP Server, occorre quindi aprire una porta specifica
- *Il contenuto dei messaggi (inviai via HTTP) è basato sul protocollo SOAP*
- La envelope di SOAP contiene la serializzazione delle strutture dati
- **La specifica WSDL viene generata automaticamente**