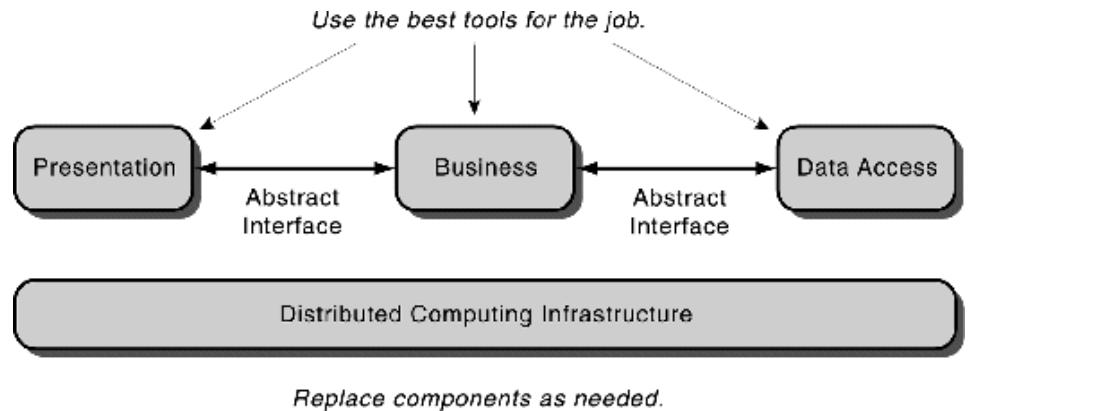


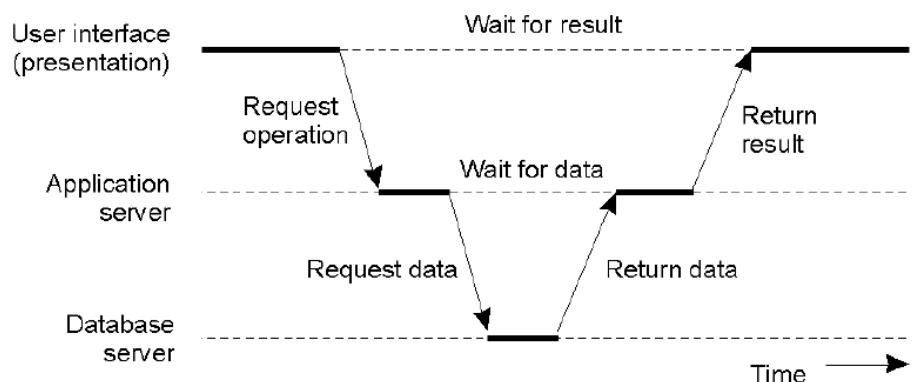
WEB APPLICATION DINAMICHE

ARCHITETTURA MULTITIERED

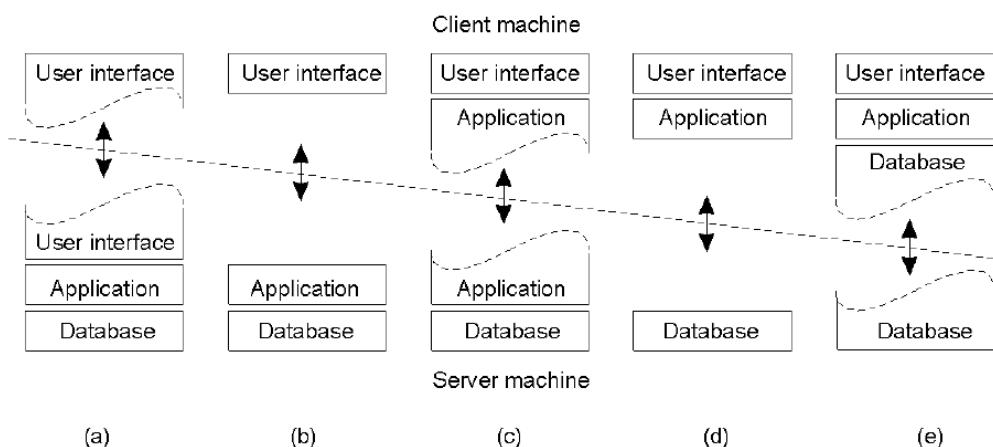
Nell'ingegneria del software, il termine architettura **multi-tier** o **architettura multi-strato** (spesso definita con l'espressione inglese n-tier architecture) indica un'architettura software in cui le varie funzionalità del software sono logicamente separate ovvero suddivise su più strati o livelli software differenti in comunicazione tra loro, nel caso di **applicazioni web** questi strati sono la **logica di presentazione**, **l'elaborazione dei processi** e la **gestione della persistenza dei dati**).



In una **three-tier application** lo strato di elaborazione funge da server per lo strato di presentazione e da client per il livello dati.



È possibile tuttavia organizzare il sistema in organizzazioni alternative come mostrato in figura



AJAX

In informatica **AJAX**, acronimo di **Asynchronous JavaScript and XML**, è una tecnica di sviluppo software per la realizzazione di applicazioni web interattive (**Rich Internet Application**). Lo sviluppo di applicazioni HTML con AJAX si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente. AJAX è **asincrono** nel senso che i dati extra sono richiesti al server e caricati in background senza interferire con il comportamento della pagina esistente. Normalmente le funzioni richiamate sono scritte con il linguaggio **JavaScript**. Tuttavia, a dispetto del nome, l'uso di JavaScript e di XML non è obbligatorio, come non è detto che le richieste di caricamento debbano essere necessariamente asincrone.

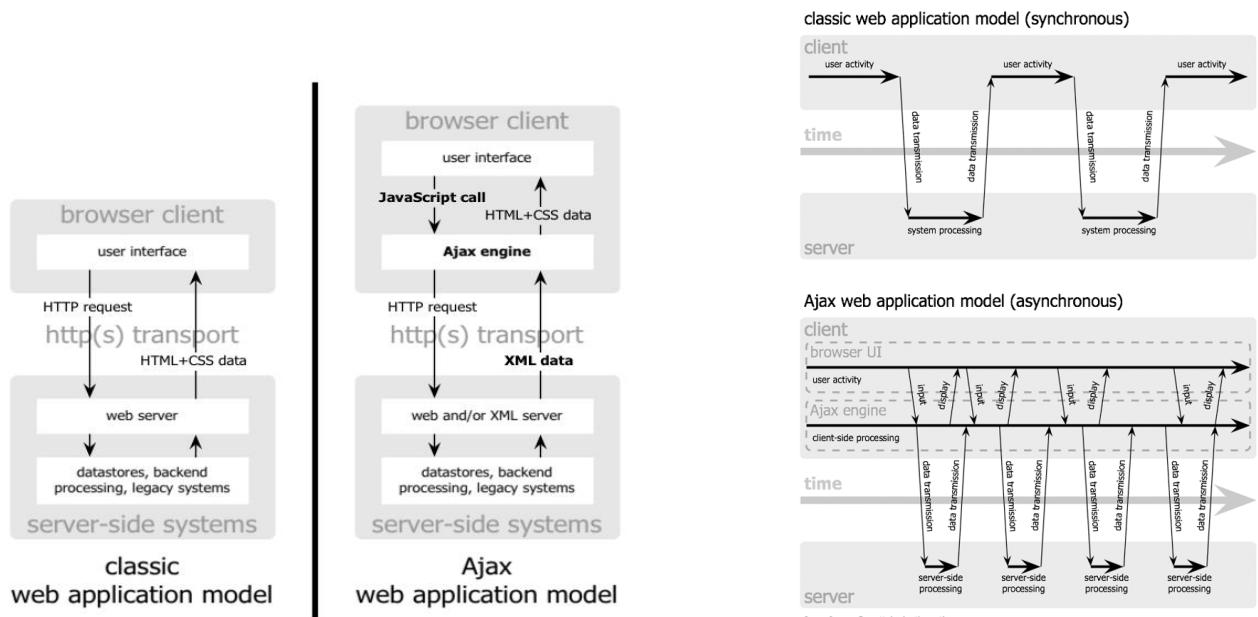
È una tecnica **multi-piattaforma**, utilizzabile cioè su molti sistemi operativi, architetture informatiche e browser web, ed esistono numerose implementazioni open source di librerie e framework.

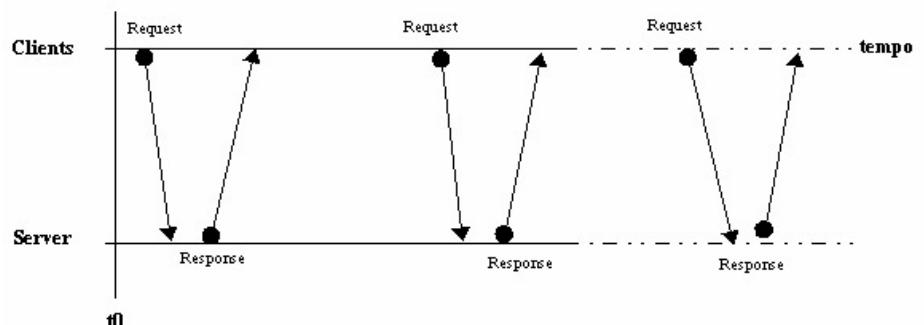
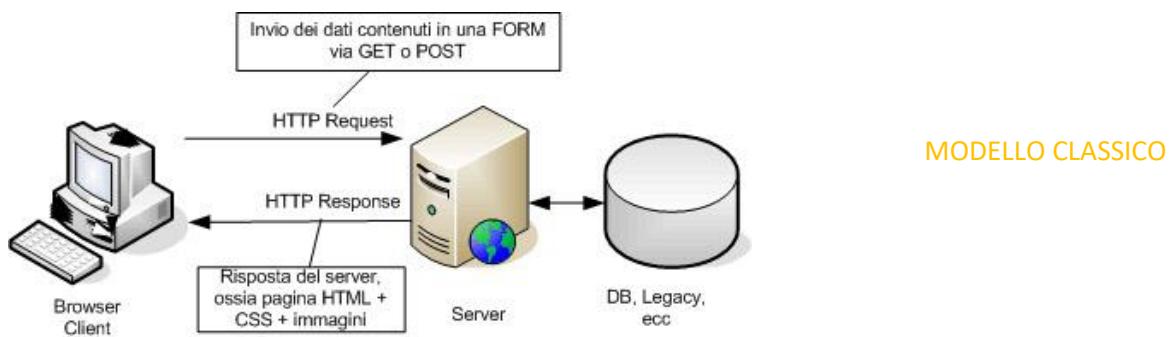
La tecnica AJAX utilizza una combinazione di:

- ❖ **HTML (o XHTML)** e **CSS** per il **markup** e lo stile;
- ❖ **DOM** manipolato attraverso un linguaggio EMCAScript come **JavaScript** o **JScript** per mostrare le informazioni ed interagirvi;
- ❖ l'oggetto **XMLHttpRequest** per l'interscambio asincrono dei dati tra il browser dell'utente e il web server. In alcuni framework AJAX e in certe situazioni, può essere usato un oggetto **Iframe** invece di XMLHttpRequest per scambiare i dati con il server e, in altre implementazioni, *tag <script>* aggiunti dinamicamente (**JSON**);
- ❖ in genere viene usato **XML** come formato di scambio dei dati, anche se di fatto qualunque formato può essere utilizzato, incluso testo semplice, HTML preformatto, JSON, e perfino EBML. Questi file sono solitamente generati dinamicamente da script lato server.

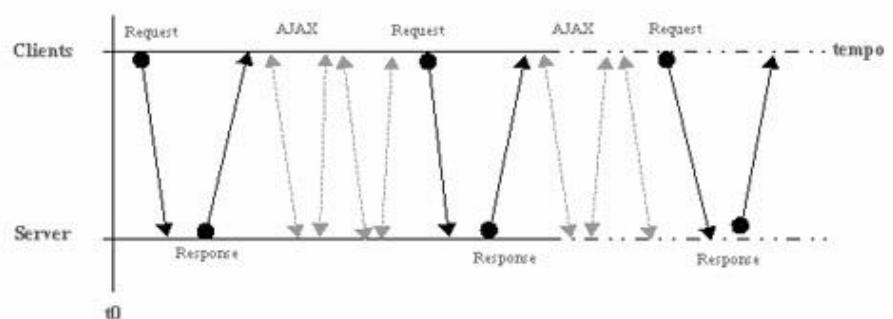
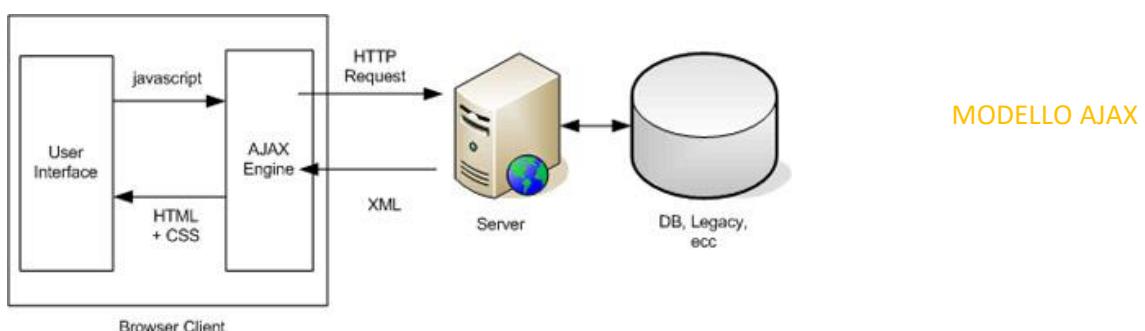
JavaScript è un linguaggio di script interpretato da un **engine**, lato client. Tale linguaggio ha la capacità di effettuare richieste http al server, in maniera trasparente all'utente, ed è in grado di rendere asincrona la comunicazione tra browser e server (web). Il modello prevede di ottenere dati XML da applicazioni web, i quali, combinati con la capacità di JavaScript di modificare gli elementi DOM, permettono di cambiare certe parti della pagina già caricata.

CONFRONTO TRA MODELLO CLASSICO E MODELLO AJAX





Vediamo ora il modello **AJAX**: in quest'ultimo le richieste vengono processate dall'engine, che si interfaccia con il server e l'interfaccia utente tra una richiesta e la successiva. Il trasporto è effettuato usando http.



Le applicazioni web **tradizionali** consentono agli utenti di compilare moduli, quando questi moduli vengono inviati, viene inviata una richiesta al web-server. Il web server agisce in base a ciò che è stato trasmesso dal modulo e risponde bloccando o mostrando una nuova pagina. Dato che molto codice HTML della prima pagina è identico a quello della seconda, viene sprecata moltissima banda. Dato che una richiesta fatta al web server deve essere trasmessa su **ogni interazione** con l'applicazione, il tempo di reazione dell'applicazione dipende dal tempo di reazione del web server. Questo comporta che l'interfaccia utente diventi molto più lenta di quanto potrebbe essere.

Le **applicazioni AJAX**, d'altra parte, possono inviare richieste al web server per ottenere **solo i dati che sono necessari** (generalmente usando **SOAP** e **JavaScript** per mostrare la risposta del server nel browser). Come risultato si ottengono applicazioni più veloci (dato che la quantità di dati interscambiati fra il browser ed il server si riduce). Anche il tempo di elaborazione da parte del web server si riduce poiché la maggior parte dei dati della richiesta sono già stati elaborati.

Esempio:

molti siti usano le tabelle per visualizzare i dati. Per cambiare l'ordine di visualizzazione dei dati, con un'applicazione tradizionale l'utente dovrebbe cliccare un link nell'intestazione della tabella che invierebbe una richiesta al server per ricaricare la pagina con il nuovo ordine. Il web server allora invierebbe una nuova query SQL al database ordinando i dati come richiesto, la eseguirebbe, prenderebbe i dati e ricostruirebbe da zero la pagina web reinviandola integralmente all'utente. Usando le tecnologie AJAX, questo evento potrebbe preferibilmente essere eseguito con un JavaScript lato client che genera dinamicamente una vista dei dati con DHTML.

PRINCIPALI PROBLEMI

Un problema abbastanza degno di nota è che, senza l'adozione di adeguate contromisure, le applicazioni AJAX possono rendere non utilizzabile il tasto "*indietro*" del browser: con questo tipo di applicazioni, infatti, non si naviga da una pagina all'altra, ma si aggiorna di volta in volta una singola parte del medesimo documento. Proprio per questo i browser, che sono **programmi orientati alla pagina**, non hanno possibilità di risalire ad alcuna di tali versioni "**intermedie**".

Altri tipi di problemi relativi al paradigma: allocazione delle operazioni, difficoltà nel debugging e la pesantezza delle richieste asincrone, è complicato mantenere uno "stato" essendo JavaScript a "generare" la pagina e non il server.

TRASMISSIONE DATI

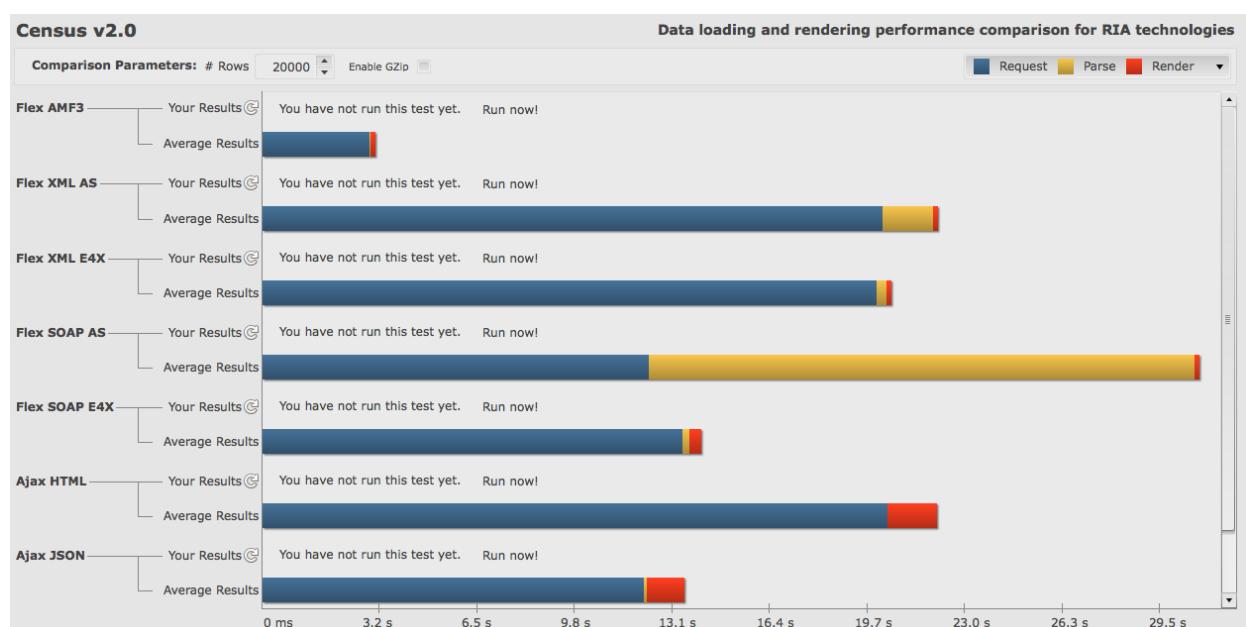
La trasmissione dei dati tra server e applicazioni RIA è un aspetto critico; la scelta del formato infatti influenza la struttura stessa dell'applicazione e le performance.

Esistono diverse alternative tecnologiche

- **SOAP (Simple Object Access Protocol)**
- **XML-RPC (Remote Procedure Call con XML per l'encoding)**
- **JSON (JavaScript Object Notation)**
- **AMF (Action Message Format, basato su SOAP)**

Ogni formato richiede un tempo diverso per la **predisposizione** dei dati lato server, per il **trasferimento**, per il **parsing** dei dati, e per il **rendering** dell'interfaccia

Di seguito sono mostrate alcune comparazioni delle prestazioni per i diversi formati di messaggi



STRUTTURA DI UNA PAGINA

```
<!DOCTYPE html>
<html>
<body>
```

HTML include il contenuto della pagina web, l'interfaccia e lo spazio per input/output;

```
<script>
```

JavaScript ha lo scopo di ospitare il codice di scripting per la pagina

```
</script>
```

```
</body>
```

```
</html>
```

Esempio: l'esempio seguente mostra come una funzione JavaScript può essere inclusa in una pagina web e di come può avvenire la comunicazione tra il browser e il motore JavaScript; l'obiettivo di questa applicazione è fornire suggerimenti all'utente mentre digita caratteri in una text box;



Struttura HTML

```
1. <html>
2. <head>
3. ...
4. </head>
5. <body>

6. <h3>A sample page that invoke a JavaScript method</h3>
7. <form action="">
8.   First name: <input type="text" id="txt1"
9.                      onkeyup="showHint(this.value)">
10. </form>

11. <p>Suggestions: <span id="txtHint"></span></p>
12. <script>
13.   function showHint(str) ...
14. </body>
15. </html>
```

Funzione JavaScript

In questo caso possiamo implementare il solo lato client

```
1. <script>
2. function showHint(str) {
3.   var names = new Array( "Anna", "Brittany", "Cinderella", "Diana", "Eva", "Fiona",
4. "Gunda", "Hege", "Inga", "Johanna", "Kitty", "Linda", "Nina", "Ophelia", "Petunia",
5. "Amanda", "Raquel", "Cindy", "Doris", "Eve", "Evita", "Sunniva", "Tove", "Unni",
6. "Violet", "Liza", "Elizabeth", "Ellen", "Wenche", "Vicky" );
7.
8.   var hint = "";
9.   if (str.length == 0) {
10.     document.getElementById("txtHint").innerHTML = "";
11.     return;
12.   }
13.   // lookup all hints from array if length of str>0
14.   for (i = 0; i < names.length; i++) {
15.     if (names[i].indexOf(str) == 0) {
16.       if (hint.length == 0)
17.         hint = names[i];
18.       else
19.         hint = hint + ", " + names[i];
20.     }
21.   }
22.   document.getElementById("txtHint").innerHTML = hint;
23. }
24. </script>
```

JavaScript non possiede nessuna funzione built-in di stampa o visualizzazione. Quest'ultimo può mostrare i dati nei seguenti modi:

- Scrivendo in una finestra di avviso, utilizzando `window.alert();`
- Scrivendo in output HTML utilizzando `document.write();`
- Scrivendo in un elemento HTML, utilizzando `innerHTML;`
- Scrivendo nella console del browser, utilizzando `console.log();`

Il codice all'interno di una funzione verrà eseguito quando “qualcosa” **invoca** (chiama) la funzione, per esempio quando si verifica un **evento** (un utente fa click su un pulsante), quando viene richiamata da codice JavaScript, o per autoinvocazione.

Gli **eventi JavaScript** sono azioni che coinvolgono elementi HTML, i quali percependo questi ultimi portano ad una reazione da parte del codice JavaScript

```
<some-HTML-element some-event="some JavaScript">
```

CONTROLLO DEI DATI E INPUT

I dati possono essere passati e controllati con invocazioni di funzioni; di seguito vengono riportati alcuni esempi

```
1. <!DOCTYPE html>
2. <html>
3. <body>
4.   <p>Click the button to display the date.</p>
5.   <button onclick="displayDate()">The time is?</button>
6.   <p id="demo"></p>

7.   <script>
8.     function displayDate() {
9.       document.getElementById("demo").innerHTML = Date();
10.    }
11.   </script>
12. </body>
13. </html>
```

The time is?

Tue Apr 05 2016 17:33:31 GMT+0200 (CEST)

```
1. <html>
2. <body>
3. <form action="">
4.   Text input: <input type="text" id="txt1"
   onkeyup="echo(this.value)">
5. </form>
6. <p>Echo: <span id="demo"></span></p>

7. <script>
8.   function echo(str) {
9.     document.getElementById("demo").innerHTML = str;
10.   }
11. </script>
12. </body>
13. </html>
```

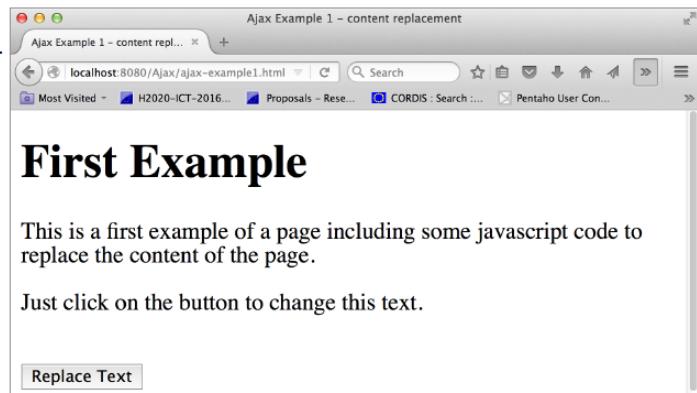
Recupero di dati da HTML

```
1. <html>
2. <body>
3. <form action="">
4.   Text input: <input type="text" id="txt1"
   onkeyup="echo()">
5. </form>
6. <p>Echo: <span id="demo"></span></p>
7. <script>
8.   function echo() {
9.     document.getElementById("demo").innerHTML =
       document.getElementById("txt1").value;
10.  }
11. </script>
12. </body>
13. </html>
```

COINVOLGIMENTO DI UN SERVER

Vediamo ora un esempio che comporta la comunicazione con un server

```
1. <body>
2.   <div id="demo">
3.     <h1>First Example</h1>
4.     <p>This is a first example of a page including some javascript
5.       code to replace the content of the page.</p>
6.     <p>Just click on the button to change this text.</p>
7.   </div>
8.   <br>
9.   <button type="button" onclick="loadDoc()">Replace Text</button>
10.  <script>
11.    function loadDoc() { ... }
12.  </script>
13. </body>
```

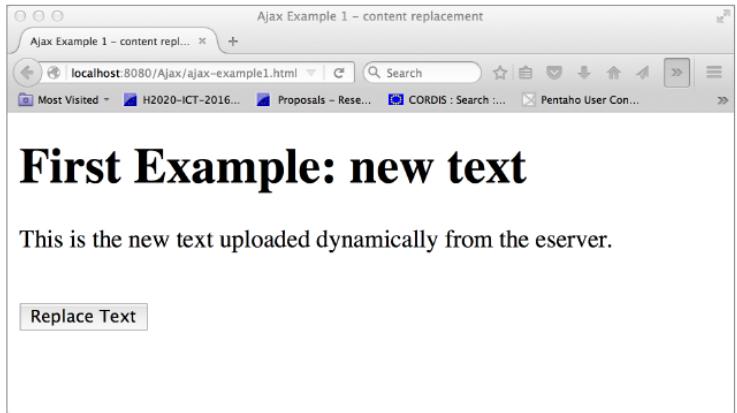


La pressione del pulsante richiama la funzione JavaScript `loadDoc()`; l'effetto di quest'ultima rimpiazzerà il testo nel div "demo";

```

1. <script>
2.   function loadDoc() {
3.     var xhttp = new XMLHttpRequest();
4.     xhttp.onreadystatechange = function() {
5.       if (xhttp.readyState == 4 && xhttp.status == 200) {
6.         document.getElementById("demo").innerHTML = xhttp.responseText;
7.       }
8.     };
9.     xhttp.open("GET", "ajax_text.html", true);
10.    xhttp.send();
11.  }
12. </script>

```



La funzione `loadDoc()` si occupa di caricare il file `ajax_text.html`, e se la risorsa è caricata correttamente (status code 200) allora il testo ricevuto viene applicato al div “demo”.

L’oggetto `XMLHttpRequest` viene utilizzato per lo scambio di dati con un server dietro le quinte. Ciò significa che è possibile aggiornare parti di una pagina web senza ricaricare quest’ultima.

La sintassi per creare un nuovo oggetto XMLHttpRequest è la seguente:

```
variable = new XMLHttpRequest();
```

La sintassi per inviare una richiesta è la seguente:

```
xhttp.open("GET", "ajax_text.html", true);
xhttp.send();
```

Più in generale:

- ⊕ **open(method, url, async):** method indica il tipo di richiesta (GET, POST), url , **async** (true o false);
- ⊕ **send():** invia la richiesta al server (utilizzata quando method = “GET”);
- ⊕ **send(string):** invia la richiesta al server (utilizzata quando method = “POST”);
- ⊕ **setRequestHeader(header, value):** aggiunge headers http alla richiesta, header : specifica il nome dell’header, value il suo rispettivo valore;

ASYNC

Async = true: specifica una funzione da eseguire quando la risposta è pronta nell’evento `onreadystatechange`;

Async = false: non è raccomandato, infatti il codice JavaScript non continuerà ad eseguire script finchè la risposta non è pronta, se il server è lento o occupato, l’applicazione si blocca;

ONREADYSTATECHANGE

Quando viene inviata una richiesta ad un server, vogliamo eseguire delle azioni correlate alla risposta di quest'ultimo; l'evento **onreadystatechange** viene attivato ogni volta che il **readyState** cambia; il **readyState** è una proprietà contenente lo stato del XMLHttpRequest. Vediamo le 3 importanti proprietà dell'XMLHttpRequest:

onreadystatechange	Memorizza una funzione (o il nome di una funzione) per essere chiamata automaticamente ogni volta che cambia la proprietà readyState
readyState	Contiene lo stato del XMLHttpRequest. Il suo valore varia da 0 a 4: 0. UNSENT 1. OPENED 2. HEADERS_RECEIVED 3. LOADING 4. DONE
stato	200: ok 404: page not found

Dunque l'evento onreadystatechange specifica che cosa accade quando la risposta del server è pronta per essere elaborata; quando readyState è 4 e lo stato è 200, la risposta è pronta:

```
1. function loadDoc() {  
2.     var xhttp = new XMLHttpRequest();  
3.     xhttp.onreadystatechange = function() {  
4.         if (xhttp.readyState == 4 && xhttp.status == 200) {  
5.             document.getElementById("demo").innerHTML = xhttp.responseText;  
6.         }  
7.     };  
8.     xhttp.open("GET", "ajax_text.html", true);  
9.     xhttp.send();  
10. }
```

FUNZIONE DI CALLBACK

Una **funzione di callback** è una funzione, o un "blocco di codice" che viene passata come parametro ad un'altra funzione. In particolare, quando ci si riferisce alla callback richiamata da una funzione, la callback viene passata come argomento ad un parametro della funzione chiamante. In questo modo la chiamante può realizzare un compito specifico (quello svolto dalla callback) che non è, molto spesso, noto al momento della scrittura del codice. Nel caso di AJAX, se si deve eseguire più di un compito su un server è buona cosa creare una funzione standard per la creazione del XMLHttpRequest, e chiamare quest'ultima per ogni attività AJAX.

```
1. function loadDoc(cFunc) {  
2.     var xhttp = new XMLHttpRequest();  
3.     xhttp.onreadystatechange = function() {  
4.         if (xhttp.readyState == 4 && xhttp.status == 200) {  
5.             cFunc(xhttp);  
6.         }  
7.     };  
8. }
```

JSON

In informatica, nell'ambito della programmazione web, **JSON**, acronimo di **JavaScript Object Notation**, è un formato adatto all'interscambio di dati fra applicazioni client/server. È basato sul linguaggio JavaScript Standard ECMA-262 3^a edizione (dicembre 1999), ma ne è indipendente. Viene usato in **AJAX** come alternativa a XML/XSLT.

La semplicità di JSON ne ha decretato un rapido utilizzo specialmente nella programmazione in AJAX. Il suo uso tramite JavaScript è particolarmente semplice, infatti l'interprete è in grado di eseguirne il **parsing** tramite la funzione **JSON.parse()**. Questo lo ha reso velocemente molto popolare a causa della diffusione della programmazione in JavaScript nel mondo del Web.

I tipo di dati supportati da questo formato sono:

- **Booleani**;
- **Interi**, numero in virgola mobile;
- **Stringhe** racchiuse tra doppi apici
- **Array** (sequenze ordinate di valori, separati da virgolet e racchiusi in parentesi quadre []);
- **Array associativi** (sequenza di coppie chiave-valore separate da virgolet racchiuse in parentesi graffe);
- **null**;

La maggior parte dei linguaggi di programmazione possiede un **typesystem** molto simile a quello definito da JSON per cui sono nati molti progetti che permettono l'utilizzo di JSON con altri linguaggi quali, per esempio: C, C#, Common LISP, Java, JavaScript, Perl, PHP, Python, Ruby e Rust.

Uno **stream JSON** dovrebbe avere un'intestazione HTTP
Content-Type: application/json

Esempio: a fianco viene mostrato un esempio che mostra i dati di un'ipotetica persona in formato JSON

```
{  
    "name": "Mario",  
    "surname": "Rossi",  
    "active": true,  
    "favoriteNumber": 42,  
    "birthday": {  
        "day": 1,  
        "month": 1,  
        "year": 2000  
    },  
    "languages": [ "it", "en" ]  
}
```

Esempio: segue un semplice esempio di richiesta AJAX in JavaScript, ad un URL che risponda con dati JSON.

```
var httpRequest = new XMLHttpRequest();  
  
// callback  
httpRequest.addEventListener( 'load', function () {  
    // parsificazione della risposta (si presume sia in formato JSON)  
    var data = JSON.parse( this.responseText );  
  
    // fare qualcosa con i dati  
    console.log( data );  
} );  
  
// inizializzazione ed invio  
httpRequest.open( 'GET', 'https://it.wikipedia.org/w/api.php?  
action=query&prop=info&titles=Pagina+principale&format=json' );  
httpRequest.send();
```

JAVASCRIPT & JQUERY

LINGUAGGI DI SCRIPTING

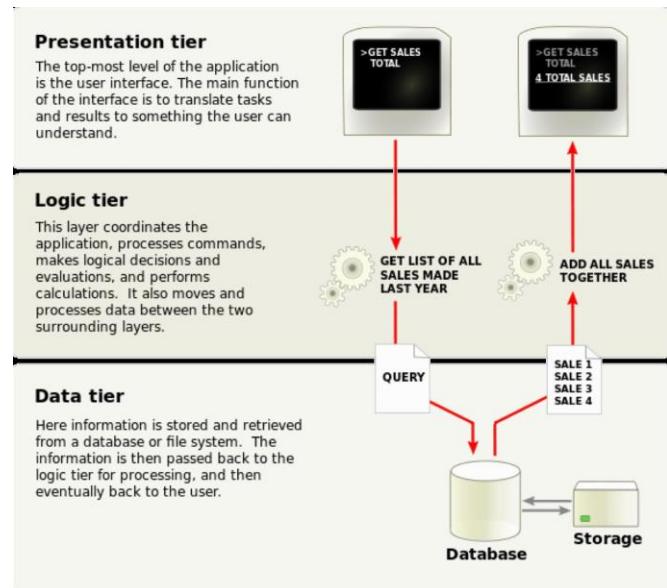
I **linguaggi di scripting** sono linguaggi di programmazione per l'automazione di compiti altrimenti eseguibili da un utente umano all'interno di un ambiente software. Questi ultimi hanno diversi scopi, ma generalmente sono **semplici, interpretati e orientati a funzionalità limitate**.

JAVASCRIPT

JavaScript è un linguaggio di programmazione **interpretato** inizialmente progettato per permettere l'esecuzione di script all'interno del browser, lato client per l'interazione con l'utente, la validazione di dati all'interno di form, la modifica di documenti web senza effetto 'pagina bianca'; è un linguaggio **dinamico, debolmente tipizzato**, la cui sintassi è stata influenzata dal C e da Java. In questo capitolo verrà mostrato attraverso una serie di esempi concreti come quest'ultimo possa essere utilizzato per costruire pagine web dinamiche, capaci di riutilizzare servizi web resi disponibili da terze parti, componendo, più che programmando, applicazioni web.

LATO CLIENT

JavaScript è di norma eseguito lato client (fatta eccezione per **Node.js** che lo vede impiegato lato server), a differenza delle altre tecnologie come JSP, PHP, Ruby etc. le quali rappresentano tecnologie lato server, e nella tipica architettura **3 tier** queste ultime realizzano gran parte delle logiche applicative e accesso ai dati.



OUTLINE

Vengono di seguito presentati i seguenti componenti del linguaggio:

- Variabili e Operatori
- Array
- Strutture di controllo del flusso
- Funzioni

VARIABILI E OPERATORI

Esempio:

```
<div class="content">
    <div class="main">
<h1>Using a Variable</h1>
<script>
    var firstName = 'Cookie';
    var lastName = 'Monster';
    //var lastName = 'Jar';
    document.write('<p>');
    document.write(firstName + ' ' + lastName);
    document.write('</p>');
</script> </div>
</div>
```

Esempio:

```
<!DOCTYPE html>
<html>
<head>
[...]
<script>
var name = prompt('What is your name?', '');
</script>
</head>

<body>
<div class="wrapper">
    <div class="header"> [...]
    </div>
    <div class="content">
        <div class="main">
            <h1>Using a Variable, Part II</h1>
            <script>
                document.write('<p>Welcome ' + name + '</p>');
            </script>
        </div>
    </div>
</div>
```

Come si può notare le variabili in JS sono **non tipizzate** (tipo dinamico")

```
var lastName = 'Monster';
lastName = 3; // No problem, here...
```

Per quanto riguarda gli **operatori** bisogna prestare particolare attenzione alla semantica di questi ultimi, infatti il loro risultato varia a seconda del contenuto delle variabili

```
var numshoes = '2';
var numsocks = 4;
document.write('<p>' + numshoes + numsocks + '</p>')
document.write('<p>');
document.write(+ numshoes + numsocks);
document.write('</p>');
```

24

6

Il **casting** è automatico, gli interi vengono trasformati in stringhe se concatenati con altre stringhe tramite il '+', ma il singolo '+' davanti a una stringa la converte in un intero.

In secondo luogo va posta attenzione agli operatori di comparazione == e !=, ma anche === e !==. I primi due, qualora le espressioni confrontate non siano dello stesso tipo, cercano di effettuare una conversazione, quindi possono generare effetti sorprendenti ed errori

Esempio:

- 0 == '0' ma 0 != '0'

In generale si suggerisce l'utilizzo della versione che non effettua la conversione di tipo

ARRAY

```
<div class="content">
<div class="main">
    <h1>Creating and Using Arrays</h1>
    <script>
        var authors = [ 'Ernest Hemingway',
                        'Charlotte Bronte',
                        'Dante Alighieri',
                        'Emily Dickinson'
                    ];
        document.write('<p>The first author is <strong>');
        document.write(authors[0] + '</strong></p>');
        document.write('<p>The last author is <strong>');
        document.write(authors[authors.length-1] + '</strong></p>');
        authors.unshift('Stan Lee');
        document.write('<p>I almost forgot <strong>');
        document.write(authors[0]);
        document.write('</strong></p>');
        authors.push('Umberto Eco');
        document.write('<p>And finally... <strong>');
        document.write(authors[authors.length-1]);
        document.write('</strong></p>');
    </script> </div>
</div>
```

```

17 <div class="main">
18   <h1>Creating and Using Arrays</h1>
19   <script>
20     var authors = [ 'Ernest Hemingway',
21       'Charlotte Bronte',
22       'Dante Alighieri',
23       'Emily Dickinson'
24     ];
25     document.write('<p>The first author is <strong>');
26     document.write(authors[0] + '</strong></p>');
27     document.write('<p>The last author is <strong>');
28     document.write(authors[authors.length-1] + '</strong></p>');
29     authors.unshift('Stan Lee');

```

Add watch expression
set: undefined
authors: [object Array]
0: "Ernest Hemingway"
1: "Charlotte Bronte"
2: "Dante Alighieri"
3: "Emily Dickinson"
length: 4
► __proto__: [object Array]

La sintassi come si può notare è analoga a quella Java ma senza nessuna necessità di inizializzazione. Gli **array** in JS sono di natura essenzialmente dinamica

```
authors.push('Umberto Eco');  
equivale a  
authors[length]='Umberto Eco';
```

Di seguito viene mostrato un esempio delle varie funzioni di utilità, passaggio da array di stringhe a stringhe e viceversa ...

```
var myarr = ["Mack", "the", "Knife"];
document.write("<p>myarr: " + myarr + " </p>");
var flattened = myarr.join(" ");
document.write("<p>flattened: " + flattened + " </p>");
var newarr = flattened.split(" ");
document.write("<p>myarr: " + newarr + " </p>");
```

In JavaScript è possibile definire inoltre delle **strutture dati generalizzate**, ovvero degli oggetti usabili per rappresentare record e strutture composte. La sintassi impone di utilizzare come delimitatori parentesi graffe, basando la struttura interna su **property**. Sono strutture dinamiche non tipizzate.

Esempio:

```
var mailArchive = {0: "Dear nephew, ... (mail number 1)",
1: "(mail number 2)",
2: "(mail number 3)"};
for (var current = 0; current in mailArchive; current++) {
  print("Processing e-mail #" + current);
  print(": " + mailArchive[current]);
}
```

Nota: *null* e *undefined* sono oggetti veri e propri, il primo rappresenta una variabile definita ma con valore ‘vuoto’, la seconda una variabile non definita; inoltre:

- *null == undefined* ma *null != undefined*

STRUTTURE DI CONTROLLO DEL FLUSSO

```
<div class="content">
    <div class="main">
        <h1>Using Conditional Statements</h1>
        <script>
            do {
                var luckyNumber = prompt('What is your lucky number?', '');
                luckyNumber = parseInt(luckyNumber, 10);
            } while (isNaN(luckyNumber));
            if (luckyNumber == 7) {
                document.write("<p>Hey, 7 is my lucky number too!</p>");
            } else if (luckyNumber == 13 || luckyNumber == 24) {
                document.write("<p>Wooh. " + luckyNumber);
                document.write("<p>? That's an unlucky number!</p>");
            } else {
                document.write("<p>The number " + luckyNumber);
                document.write("<p>is lucky for you!</p>");
            }
        </script> </div>
</div>
```

- Strutture di selezione - if

```
if (cond1) {
    // door #1
} else if (cond2) {
    // door #2
} else {
    // door #3
}
```
- Strutture di selezione – switch

```
switch(var) {
    case 'val1':
        // door #1
        break;
    case 'val2':
        // door #2
        break;
    default:
        // door #3
}
```
- Strutture di iterazione – while

```
while(cond) {
    // code to iterate
}
```
- Strutture di iterazione – do while

```
do {
    // code to iterate
} while(cond);
```
- Strutture di iterazione – for

```
for(init; cond; step) {
    // code to iterate
}
```

FUNZIONI

```
<div class="content"> <div class="main"> <h1>A Simple Quiz</h1>
    <script>
        var score=0;
        var questions = [
            ['How many moons does Earth have?', 1],
            ['How many moons does Saturn have?',31],
            ['How many moons does Venus have?', 0]
        ];
        function askQuestion(question) {
            var answer = prompt(question[0], '');
            if (answer == question[1]) {
                alert('Correct!');
                score++;
            } else {
                alert('Sorry. The correct answer is ' + question[1]);
            }
        }
        for (var i=0; i<questions.length; i++) {
            askQuestion(questions[i]);
        }
        var message = 'You got ' + score;
        message += ' out of ' + questions.length;
        message += ' questions correct.';
        document.write('<p>' + message + '</p>');
    </script> </div>
</div>
```

-----0-----0-----

JQUERY

jQuery è un framework JavaScript che rende molto semplice la scrittura di applicazioni web offrendo funzionalità come la manipolazione di HTML/DOM e CSS, metodi per eventi HTML, effetti ed animazioni, supporto e programmazione AJAX, e varie altre utilità (anche tramite plugin).

Esempio: selezione e modifica di elementi di una pagina



The screenshot shows two side-by-side browser windows. Both windows display the same page from 'JavaScript & jQuery: The Missing Manual'. The left window shows the original page content, which includes several paragraphs of text. The right window shows the same page after jQuery has been applied to automatically pull in quotes from a specific class, resulting in a list of highlighted quote snippets.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Using Functions</title>
<link href="../../_css/site.css" rel="stylesheet">
</head>
<body>
<div class="header"> [...] </div>
<div class="content">
<div class="main">
<h1>Auto-Pull Quotes</h1>
<h2>Vestibulum semper</h2>
<p>Vestibulum semper [...] <span class="pq">Vivamus justo mi, aliquam vitae, eleifend et, lobortis quis, eros.</span>
[...] </p>
<h2>Morbi dictum</h2>
<p>Donec at sapien [...] <span class="pq">Etiam mattis. Donec sed diam nec odio molestie iaculis.</span>
[...] </p>
<h2>Praesent sed est</h2>
<p>Praesent sed est ac metus facilisis [...] </p>
<p>Integer lacus. [...] </p>
</div>
</div>
<div class="footer"> [...] </div>
</body>
</html>
```

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Using Functions</title>
<link href="../../_css/site.css" rel="stylesheet">
<style>
.pullquote {
    float: right;
    clear: right;
    width: 200px;
    border: 1px solid black;
    padding: 10px;
    font-size: 20px;
    background-color:rgb(195,151,51) ;
    margin: 20px 0 10px 10px;
    font-style: italic;
    color: rgb(255,255,255) ;
}
</style>

```

```

<head>
[...]
<script src="../../_js/jquery-1.7.2.min.js"></script>
<script>
$(document).ready(function() {
    $('span.pq').each(function() {
        var quote=$(this).clone();
        quote.removeClass('pq');
        quote.addClass('pullquote');
        $(this).before(quote);
    }); // end each
}); // end ready
</script>
</head>

```

ELEMENTI BASE DI JQUERY

La sintassi di jQuery è specificamente orientata a permettere una rapida e semplice selezione di elementi nel documento HTML. La sintassi di base di un comando jQuery è la seguente:

`$(selector).action()`

Il '\$' definisce l'accesso a funzionalità offerte da jQuery, *selector* viene utilizzato per specificare una sorta di query per selezionare una parte del documento HTML, *action* specifica un'azione da effettuare sull'elemento stesso.

La porzione

```

$(document).ready(function() {
    [...]
}); // end ready

```

indica che vogliamo effettuare l'azione *ready* sull'oggetto *document* passando come parametro una funzione anonima, specificata per esteso. L'azione *ready* specifica che il parametro ricevuto, una funzione, va richiamata non appena il documento è "pronto", ovvero è stato completamente caricato.

Quello che vorremmo fare nel nostro esempio è selezionare le porzioni di documento negli *span* di classe *pq*, clonarli dandogli al contempo lo stile desiderato. Il selettore per individuare tutti gli *span* di classe *pq* è

`$('span.pq)`

Per iterare su un insieme di elementi jQuery offre la funzione *each*.

La porzione

```
$('span.pq').each(function() {  
    [...]  
}); // end each
```

permette di specificare una funzione anonima che va applicata ad ogni *span* di classe *pq*.

A questo punto possiamo operare sui singoli *span*. In particolare il selettore *\$(this)*, trovandoci all'interno di una funzione chiamata (implicitamente) su uno degli *span* stessi, ci permette di ottenere il riferimento che ci serve.

La porzione

```
var quote=$(this).clone();  
quote.removeClass('pq');  
quote.addClass('pullquote');  
$(this).before(quote);
```

permette quindi di clonare lo *span* nella variabile *quote*, rimuovere la classe *pq*, aggiungere la classe *pullquote* e aggiungere la variabile stessa al documento, appena prima dello *span* stesso.

Esempio: gestione di eventi e modifica dinamica di una pagina



```
<!DOCTYPE html>  
<html>  
<head> <meta charset="UTF-8">  
<title>A One Page Faq</title>  
<link href="../../css/site.css" rel="stylesheet">  
</head>  
<body>  
<div class="header"> [...] </div>  
<div class="content">  
<div class="main">  
<h1>A One Page FAQ</h1>  
<h2>I've heard that JavaScript is the long-lost fountain of youth. Is this true?</h2>  
<div class="answer"> <p>Why, yes it is! Studies prove that learning JavaScript freshens the mind and extends life span by several hundred years. (Note: some scientists disagree with these claims.)</p> </div>  
<h2>Can JavaScript really solve all of my problems?</h2>  
<div class="answer"> <p>Why, yes it can! It's the most versatile programming language ever created and is trained to provide financial management advice, life-saving CPR, and even to take care of household pets.</p>  
</div>  
<h2>Is there nothing JavaScript <em>can't</em> do?</h2>  
<div class="answer"> <p>Why, no there isn't! It's even able to write its own public relations-oriented Frequently Asked Questions pages. Now that's one smart programming language!</p> </div>  
</div>  
</div>  
<div class="footer"> [...] </div>  
</div>  
</body>  
</html>
```

```

<!DOCTYPE html>
<html>
<head> <meta charset="UTF-8">
<title>A One Page Faq</title>
<link href="../../_css/site.css" rel="stylesheet">
<style type="text/css">
h2 {
    background: url('../../_images/open.png') no-repeat 0 11px;
    padding: 10px 0 0 25px;
    cursor: pointer;
}
h2.close {
    background-image: url('../../_images/close.png');
}
.answer {
    margin-left: 25px;
}
</style>

<script src="../../_js/jquery-1.7.2.min.js"></script>
<script>
$(document).ready(function() {
    $('.answer').hide();
    $('.main h2').toggle(
        function() {
            $(this).next('.answer').slideDown();
            $(this).addClass('close');
        },
        function() {
            $(this).next('.answer').fadeOut();
            $(this).removeClass('close');
        }
    ); // end toggle
}); // end ready
</script>

```

ULTERIORI ELEMENTI BASE DI JQUERY

Tramite la tecnica precedentemente illustrata definiamo il da farsi al caricamento della pagina. La porzione di codice

```

$('.answer').hide();
$('.main h2').toggle(
    [...]
)

```

specifica che in primis, i `div` di classe `answer` devono essere nascosti, in seconda battuta, ai tag `h2` che siano figli di `div` di classe `main`, va applicata la funzione `toggle`. La funzione `toggle` permette di associare a un elemento due diverse funzioni, da chiamare rispettivamente per ogni click dispari e pari.

In questa porzione di codice

```
$('.main h2').toggle(
    function() {
        $(this).next('.answer').slideDown();
        $(this).addClass('close');
    },
    function() {
        $(this).next('.answer').fadeOut();
        $(this).removeClass('close');
    }
); // end toggle
}
```

si associano due funzioni anonime:

1. seleziona il primo *div* di tipo *answer* che segue l'oggetto corrente (il tag *h2*) e vi applica la funzione *slideDown*, aggiunge la classe *close* all'oggetto corrente
2. seleziona il primo *div* di tipo *answer* che segue l'oggetto corrente e vi applica la funzione *fadeOut*, rimuove la classe *close* all'oggetto corrente.

NODE.JS

INTRODUZIONE

Node.js è un ambiente asincrono ed **event driven** per la costruzione di applicazioni distribuite basate su **Javascript**. È basato sul motore JavaScript Chrome V8, ed è stato creato nel 2009, la versione attuale è 6.10.3 (versione LTS).

Node.js **non è un framework**, come erroneamente indicato da alcune risorse che lo descrivono, lavora infatti a livello più basso essendo sostanzialmente un **interprete** con alcune librerie di base non native del linguaggio JavaScript, ad esempio per l'I/O sul file, per l'implementazione di server per vari protocolli standard. Esistono framework per applicazioni web scritti in/su Node.js



Node.js ha anche stimolato la creazione di una sorta di ecosistema (in particolare librerie dette moduli e un sistema di gestione dei pacchetti NPM).

È open source, e spesso è definito “JavaScript lato server”.

CARATTERISTICHE

Node.js ha un approccio profondamente diverso, per esempio, da quello della piattaforma Java, in particolare a livello architettonico oltre che banalmente a livello linguistico. Quest'ultimo infatti adotta un approccio **event driven**, sfruttando il loop di gestione degli eventi di JavaScript, in particolare sulla possibilità di definire funzioni di callback per implementare uno schema di I/O asincrono (non bloccante). Da sottolineare inoltre che in Node.js avviene tutto all'interno di **un singolo thread di controllo**.

I programmi in Node.js sono scritti in JavaScript ma dobbiamo considerare il fatto che il codice non è ospitato da un browser, dunque verrà a mancare la libreria **DOM**, per esempio (anche se è possibile installare dei moduli che replicano le funzionalità di jQuery come per esempio Cheerio).

PROGRAMMAZIONE EVENT DRIVEN

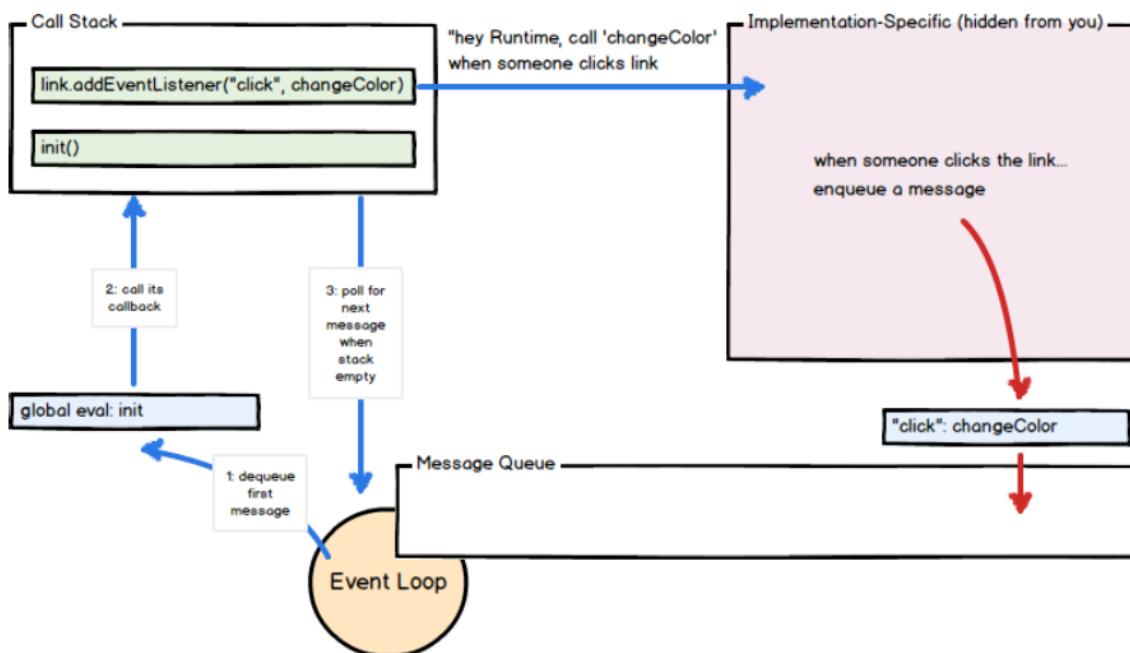
Cerchiamo di dare una breve spiegazione riguardante la programmazione a eventi senza avere la pretesa di dare una definizione esaustiva ...

Per farlo, ricordiamo come si lavora nella realizzazione di pagine web dinamiche con approccio **Ajax**: nel documento HTML vengono tipicamente definiti gli elementi di interfaccia, in script JS vengono definite funzioni per la gestione di eventi relativi a questi elementi, o all'intera pagina. In generale questo è l'approccio dominante nella realizzazione di GUI.

L'idea è che in un approccio orientato agli eventi ci sia un loop principale di un singolo thread di controllo che di volta in volta preleva un evento da una coda (nella quale vengono inseriti gli eventi che hanno luogo) e lancia il callback associato alla gestione dell'evento stesso.

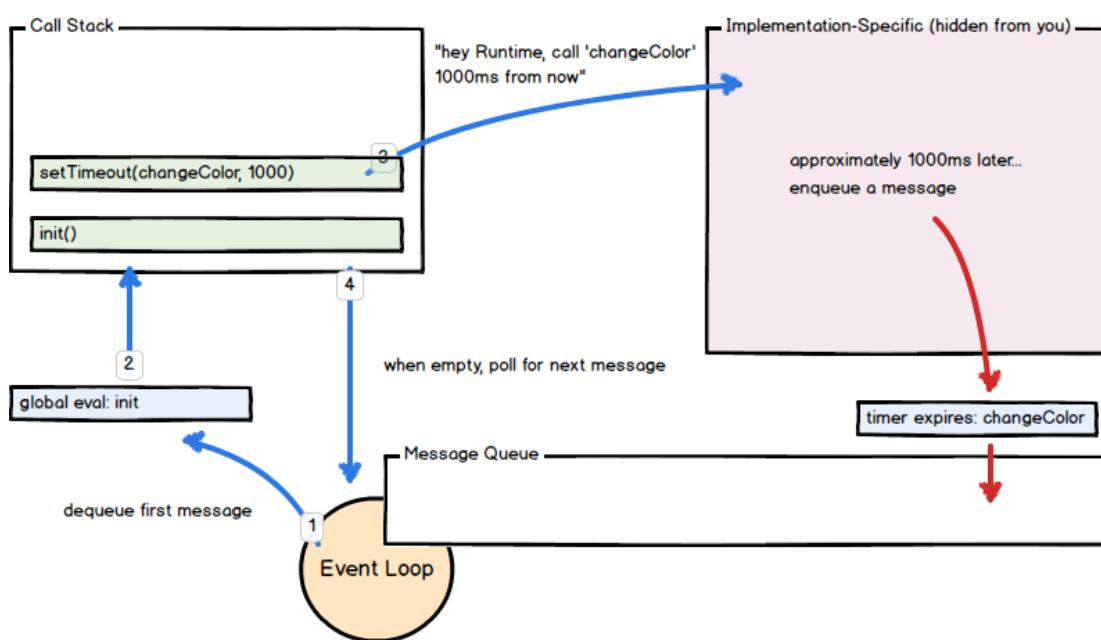
Esempio: di seguito viene mostrato un esempio di event loop di JavaScript

```
function init() {  
  var link = document.getElementById("foo");  
  link.addEventListener("click", function changeColor(){  
    this.style.color = "burlywood";  
  });  
}  
  
init();
```



Esempio: un altro esempio sempre in JS

```
function init() {  
  var link = document.getElementById("foo");  
  setTimeout(function changeColor() {  
    link.style.color = "burlywood";  
  }, 1000);  
}  
  
init();
```



I/O NON BLOCCANTE

- I/O tradizionale

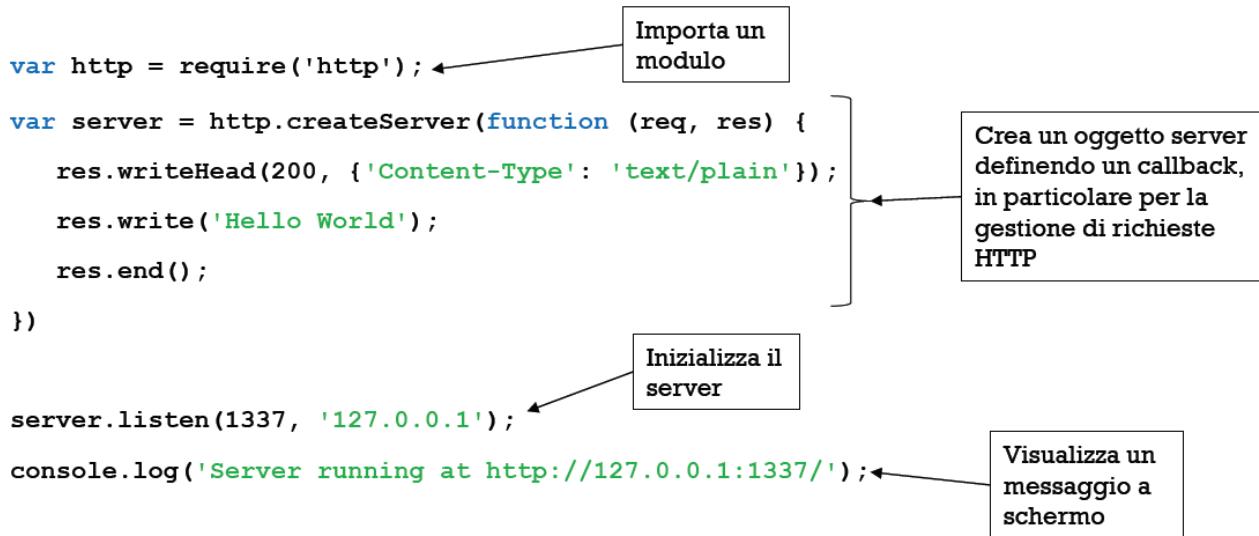
```
var result = db.query("select x from table_Y");  
doSomethingWith(result); //wait for result!  
doSomethingWithOutResult(); //takes place after I/O...
```

- I/O non bloccante

```
db.query("select x from table_Y", function (result){  
  doSomethingWith(result); //wait for result!  
});  
doSomethingWithOutResult(); //executes immediately...
```

SERVER NODE.JS

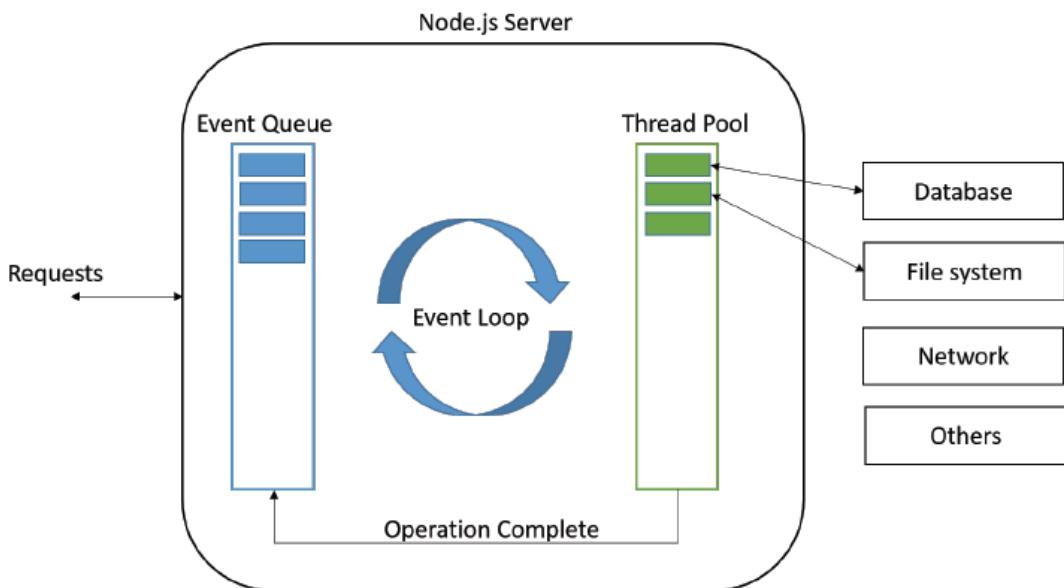
Di seguito viene illustrato un esempio di **server** in Node.js



Osservazioni:

Un tipico server in **Java** è dotato di diversi thread: sicuramente almeno un thread principale per catturare le richieste da parte dei client, ognuna delle quali viene gestita da un nuovo thread creato appositamente. L'accesso alle risorse condivise va gestito con particolare attenzione.

Il nostro **server in Node.js** è gestito invece da un singolo thread di controllo, che si può avvalere di un pool di thread offerto dalla piattaforma per gestire funzioni di I/O asincrone.



I cambi di contesto (**context switch**) sono presenti anche in sistemi basati su Node.js, ma non c'è un cambio di contesto quando termina la chiamata di un callback e si passa ad un altro. Questo rende Node.js un sistema **molto veloce**.

Esempio: esempio di “Hello World” in Node.js (TCP)

```
// Load the net module to create a tcp server.  
var net = require('net');  
  
// Creates a new TCP server. The handler argument is automatically set as a  
// listener for the 'connection' event  
var server = net.createServer(function (socket) {  
  
    // Every time someone connects, tell them hello and then close  
    // the connection.  
    console.log("Connection from " + socket.remoteAddress);  
    socket.end("Hello World\n");  
});  
  
// Fire up the server bound to port 7000 on localhost  
server.listen(7000, "localhost");  
  
// Put a friendly message on the terminal  
console.log("TCP server listening on port 7000 at localhost.");
```

Esempio: esempio di “Hello World” in Node.js (HTTP)

```
// Load the http module to create an http server.  
var http = require('http');  
  
// Configure our HTTP server to respond with Hello World to all requests.  
var server = http.createServer(function (request, response) {  
    response.writeHead(200, {"Content-Type": "text/plain"});  
    response.end("Hello World\n");  
});  
  
// Listen on port 8000, IP defaults to 127.0.0.1  
server.listen(8000);  
  
// Put a friendly message on the terminal  
console.log("Server running at http://127.0.0.1:8000/");
```

Esempio: server Echo (TCP) e un client

```
var net = require('net');
var server = net.createServer(function (socket){
  socket.write('Echo server');
  socket.pipe(socket);});
server.listen(3000, '127.0.0.1');

var net = require('net');
var client = new net.Socket();

client.connect(3000, '127.0.0.1', function() {
  console.log('Connected');
  client.write('Hello, server! Love, Client.');
});

client.on('data', function(data) {
  console.log('Received: ' + data);
  client.destroy();
});

client.on('close', function() {
  console.log('Connection closed'));
```

-----o-----o-----

L'ECOSISTEMA DI NODE.JS

Node.js fa pesantemente uso di moduli, sostanzialmente librerie, scritte in Node.js. Tuttavia non è complicato realizzarne di proprie: si tratta di collezioni di funzioni alcune delle quali esportate, visibili all'esterno del modulo. Queste librerie a volte sono organizzate in package, insiemi di librerie dotate di una descrizione più sistematica delle dipendenze, gestite da Node Package Manager, un sistema per il loro download e gestione.

Esempio:

```
npm install "package_name";
//package should be available in
npm registry @ npmjs.org
```

PACKAGE ASCIIFY

Consente di convertire una normale stringa in un disegno che la rappresenta nel quale i singoli elementi dell'immagine siano dei caratteri del codice ASCII. Ci sono molte varianti sul tema (ad esempio per convertire un'immagine bitmap in una basata su caratteri ASCII).

Esempio: di seguito viene mostrato un esempio di server ASCIIIFY

```
var express = require('express');
var app = express();
var asciiify = require('asciiify');
var http = require('http');

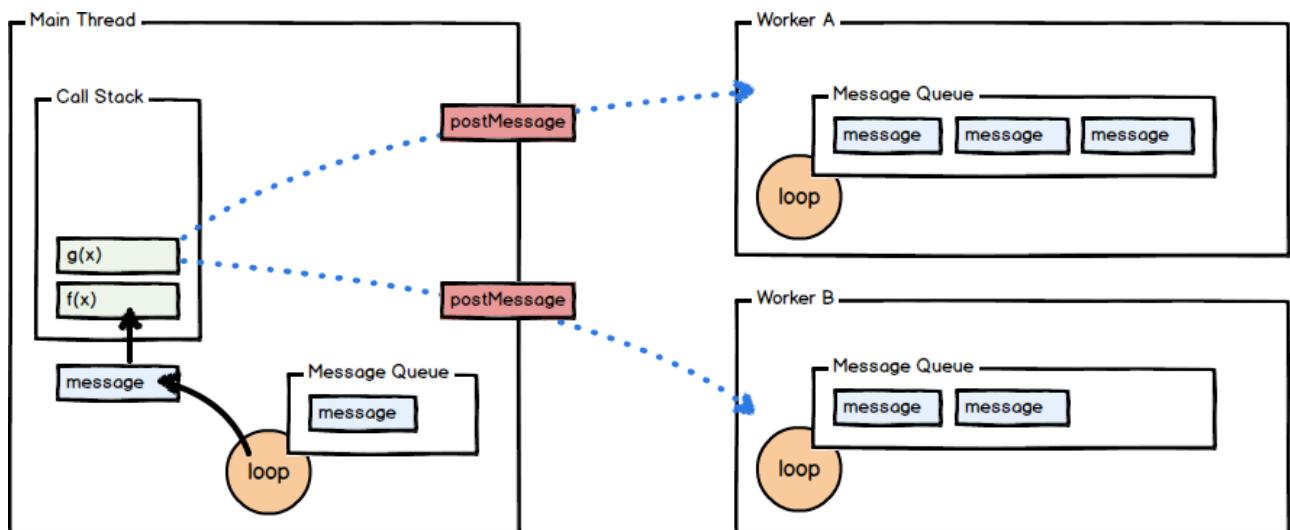
app.get('/:text', function (req, res) {
  var text = req.params.text;
  asciiify(text, function(err, asciified){
    asciified = '<pre>' + asciified + '</pre>';
    res.send(asciified);
  });
})

var server = app.listen(3000, '127.0.0.1', function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

-----o-----o-----

WEB WORKERS

Si tratta di thread addizionali creabili per sgravare quello principale da operazioni pesanti. Ognuno dei nuovi thread ha propri event loop, coda di messaggi e stack. La comunicazione con il thread principale avviene tramite messaggi. Questa possibilità è stata introdotta nel contesto di HTML5, ma esistono package per Node.js



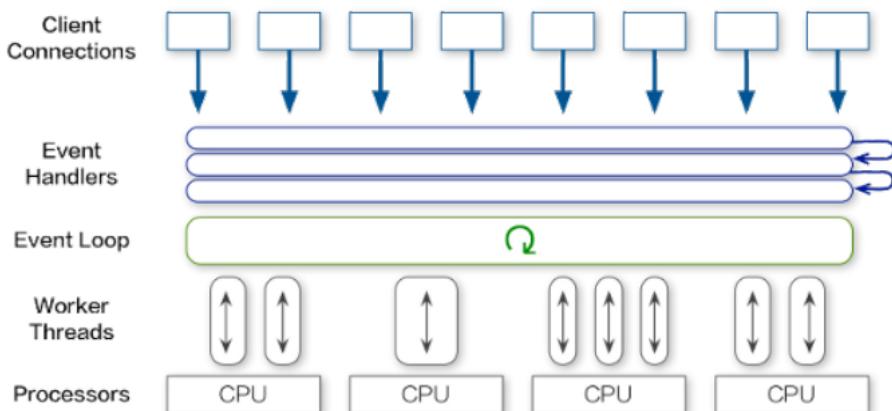
Esempio: di seguito è mostrato un esempio di uso di WEBWORKER-THREADS

```
var Worker = require('webworkerthreads').Worker;
// var w = new Worker('worker.js');

// You may also pass in a function:
var worker = new Worker(function(){
    postMessage("Working before postMessage('ali').");
    this.onmessage = function(event) {
        postMessage('Hi ' + event.data);
        self.close();
    };
});

worker.onmessage = function(event) {
    console.log("Worker said : " + event.data);
};

worker.postMessage('ali');
```

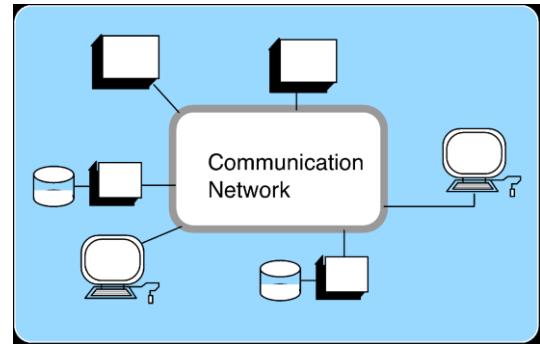


CONCORRENZA IN JAVA

PROGRAMMA CONCORRENTE

Un programma **sequenziale** ha un singolo thread di controllo.
Un programma **concorrente** ha più thread di controllo che gli permettono di svolgere **diversi compiti in parallelo** ed eventualmente di **controllare molteplici attività** che hanno luogo allo stesso tempo.

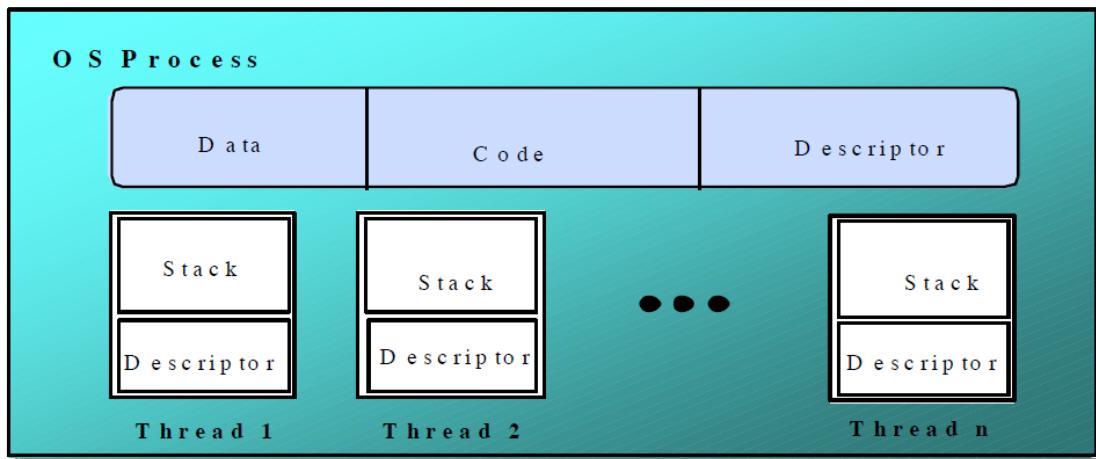
Un **software concorrente** può eventualmente essere **distribuito** tra diverse macchine utilizzando una comunicazione via rete, dove ogni nodo deve svolgere un compito diverso in parallelo agli altri.



PERCHÈ FARE PROGRAMMAZIONE CONCORRENTE

Può essere utile utilizzare questo tipo di programmazione per sfruttare gli attuali processori multi-core, per evitare di bloccare l'intera esecuzione di un'applicazione a causa dell'attesa del completamento di un'azione di I/O, o ancora per sfruttare in modo più adeguato un programma. In particolare programmi che interagiscono con l'ambiente, controllano diverse attività, gestiscono diversi tipi di eventi etc, magari mentre forniscono funzionalità ad un utente umano.

PROCESSI E THREAD VISTI DAL SISTEMA OPERATIVO



Un **processo** nel sistema operativo è rappresentato dal suo codice, uno spazio di indirizzamento privato esclusivo (dati) e un descrittore contenente informazioni di vario genere (tra cui, ad esempio, lo stato dei registri). Per poter consentire l'esecuzione di diversi **thread** di controllo (**lightweight process**) è dotato di stack multipli, uno per thread. Anche i thread sono dotati di un descrittore (per l'informazione di stato e variabili locali).

I processi **non condividono risorse** tra loro, certo possono essere eseguiti dalla stessa CPU, ma a parte questo, per comunicare devono utilizzare risorse del SO, come file, system call (es: pipe, socket), oppure servizi/funzionalità offerte da middleware o altre applicazioni (es: DBMS presente sulla macchina).

I **thread** di uno stesso processo condividono lo spazio di indirizzamento, quindi **possono condividere oggetti in memoria**. Naturalmente l'accesso a risorse condivise va gestito onde evitare problemi.

THREAD IN JAVA

I thread rappresentano il paradigma fondamentale per l'esecuzione dei programmi in ambiente Java; il linguaggio Java, con la propria API, è provvisto di una ricca gamma di funzionalità per la generazione e la gestione dei thread. Tutti i programmi scritti in Java incorporano almeno un thread di controllo, persino un semplice programma costituito da un metodo `main()`, è eseguito dalla JVM come un singolo thread.

In un programma Java vi sono due tecniche per la generazione dei thread. Una è creare una nuova classe derivata dalla classe `Thread` e “sovrascrivere” (override) il suo metodo `run()`. L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia `Runnable`, definita come segue:

```
public interface Runnable
{
    public abstract void run();
}
```

Per implementare `Runnable`, una classe è tenuta a definire il metodo `run()`. Il codice che implementa `run()` sarà eseguito in un thread distinto.

Esempio: il seguente esempio mostra un programma multithread che calcola la somma degli interi da 0 a N.

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

```

public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println(
                        ("The sum of "+upper+" is "+sumObject.getSum()));
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
    }
}

```

La classe *summation* implementa l'interfaccia *Runnable*. La generazione del thread prevede che si crei un'istanza della classe *Thread* passando al costruttore un oggetto *Runnable*. La creazione di una classe *Thread* non equivale a generare un nuovo thread, bensì è il metodo *start()* che avvia effettivamente il nuovo thread. L'invocazione di quest'ultimo ha duplice effetto:

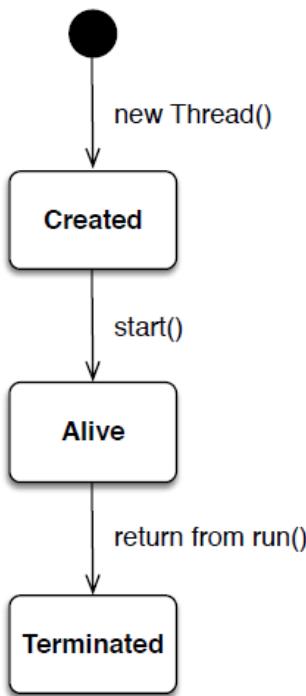
1. Alloca la memoria e inizializza un nuovo thread nella JVM;
2. Chiama il metodo *run()*, cosa che rende il thread eseguibile sulla JVM. Si osservi come quest'ultimo non sia mai chiamato per via diretta, ma solo tramite la chiamata del metodo *start()*;

All'avvio del programma che calcola la somma, la JVM crea due thread: il primo è il thread genitore, che inizia a essere eseguito dal metodo *main()*; il secondo thread, figlio del primo, ha origine quando il metodo *start()* è invocato sull'oggetto di classe *Thread*. L'esecuzione di questo thread figlio inizia dal metodo *run()* della classe *Summation*. Dopo aver restituito il valore della somma, il thread termina all'uscita dal proprio metodo *run()*.

In quanto linguaggio orientato agli oggetti puro, Java non contempla la nozione di **variabile globale**: la **condivisione dei dati fra più thread** in Java avviene tramite il passaggio di riferimenti a uno stesso oggetto.

Nel nostro esempio il thread principale e quello che calcola la somma condividono un oggetto di classe *Sum*. A tale oggetto condiviso si accede attraverso gli appositi metodi *getSum()* e *setSum()*. Ci si potrebbe chiedere perché non si usi un oggetto *Integer* anichè definire una nuova classe *sum*. La ragione è che la classe *Integer* è **immutable**, ovvero, una volta impostato il valore di una sua istanza, non può cambiare.

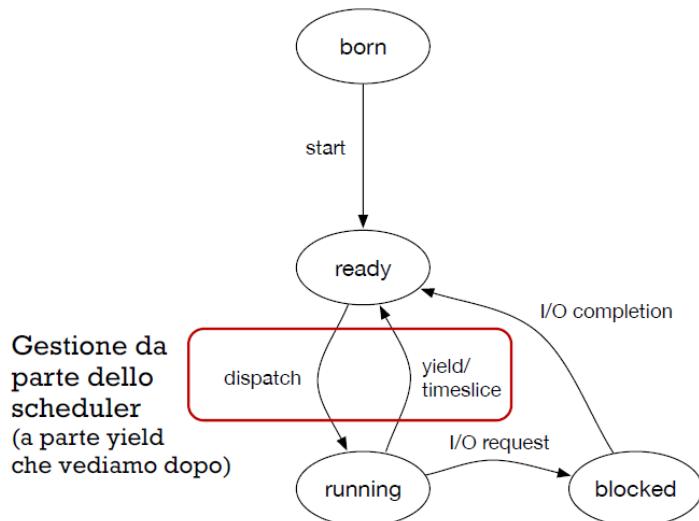
CLICLO DI VITA DI UN THREAD IN JAVA



L'invocazione del metodo `start()` porta il thread ad eseguire il metodo `run()` (proprio per classi che estendono `Thread` o quello del `Runnable` passato come parametro al proprio costruttore).

Esiste un predicato (metodo che ritorna un booleano) `isAlive()` che può essere usato per valutare se il thread sia stato fatto partire e al contempo se non sia stato terminato.

Un thread terminato non può essere fatto ripartire.



Esempio:

```

class RunnableDemo implements Runnable{
private Thread t;
private String threadName;
RunnableDemo (String name){
    threadName = name;
    System.out.println("Creating " + threadName);
}
public void run(){
    System.out.println("Running " + threadName);
    try { for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " +
                           threadName + ", " + i);
        Thread.sleep(50);
    }
    } catch (InterruptedException e) {
        System.out.println("Thread " +
                           threadName + " interrupted.");
    }
    System.out.println("Thread " + threadName +
                       " exiting.");
}
public void start () {
    System.out.println("Starting " + threadName);
    if(t == null) {
        t = new Thread (this, threadName);
        t.start ();
    }
}
}
  
```

```

public class TestThread{
public static void main(String args[]){
    RunnableDemo R1 = new RunnableDemo(
        "Thread-1");
    R1.start();
    RunnableDemo R2 = new RunnableDemo(
        "Thread-2");
    R2.start();
}
  
```

`Thread.sleep(milli)`: ferma l'esecuzione del thread per *milli* millisecondi, liberando CPU (crf. Grafico di due slide precedenti)

Attenzione: questo metodo può lanciare una `InterruptedException`, che va catturata e gestita (qui non facciamo nulla se non visualizzare l'informazione);

PROBLEMI DI GESTIONE DI RISORSE CONDIVISE

Un problema relativo alle gestioni di risorse condivise è il controllo delle operazioni in parallelo e della memoria: può capitare che i thread si interrompano e che i dati contenuti in essi non vengano salvati, non aggiornando i valori

Esempio:

```
class ContoCorrente {  
    private float saldo;  
    public ContoCorrente(float saldoIniziale) {  
        saldo = saldoIniziale;  
    }  
  
    public void deposito(float cifra) {  
        saldo += cifra;  
    }  
  
    public float prelievo(float cifra) {  
        if(saldo > cifra){  
            saldo -= cifra;  
            return cifra;  
        }  
        float prelevati = saldo;  
        saldo = 0.0f;  
        return prelevati;  
    }  
  
    public float saldo() {  
        return saldo;  
    }  
}
```

Supponiamo di avere un thread t1 che deposita 20 ed un thread t2 che preleva 10; può succedere che t2 inizia calcoli il valore da inserire nel nuovo saldo (90) ma mentre sta per salvare il dato nel saldo lo scheduler interrompe t2 e fa partire t1, che calcola il nuovo saldo (120) e lo salva; a questo punto t1 riparte e salva il valore calcolato (90) nel saldo.

Il deposito in pratica **non ha avuto luogo**.

Vediamo di seguito alcuni esempi di esecuzioni differenti

The image shows three separate terminal windows, each with an "input" and "clear" button. Each window displays the same Java code and its execution results.

Terminal 1 (Left): Shows a sequence of operations where a deposit of 100.0 is made followed by a withdrawal of 200.0, resulting in a final balance of 9950.0.

```
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07,
mixed mode)
>
Il saldo iniziale del conto è 10000.0
Effettuato un deposito di 100.0
Effettuato un prelievo di 200.0
Il saldo del conto è 9900.0
Effettuato un deposito di 150.0
Il saldo del conto è 9800.0
Il saldo del conto è 9900.0
Effettuato un prelievo di 100.0
Il saldo del conto è 9900.0
Effettuato un deposito di 50.0
Il saldo del conto è 10050.0
Effettuato un prelievo di 50.0
Il saldo del conto è 9950.0
>
```

Terminal 2 (Top Right): Shows a sequence of operations where a deposit of 100.0 is made followed by a withdrawal of 50.0, resulting in a final balance of 9900.0. This outcome is incorrect because the deposit was not counted.

```
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07,
mixed mode)
>
Il saldo iniziale del conto è 10000.0
Effettuato un deposito di 100.0
Il saldo del conto è 9800.0
Effettuato un prelievo di 200.0
Effettuato un prelievo di 50.0
Il saldo del conto è 9900.0
Effettuato un deposito di 150.0
Il saldo del conto è 9950.0
Effettuato un deposito di 50.0
Il saldo del conto è 9900.0
Il saldo del conto è 9900.0
Effettuato un prelievo di 100.0
Il saldo del conto è 9900.0
>
```

Terminal 3 (Bottom Right): Shows a sequence of operations where a deposit of 100.0 is made followed by a withdrawal of 100.0, resulting in a final balance of 9800.0. This outcome is also incorrect because the deposit was not counted.

```
java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07,
mixed mode)
>
Il saldo iniziale del conto è 10000.0
Effettuato un deposito di 100.0
Il saldo del conto è 9950.0
Effettuato un prelievo di 50.0
Il saldo del conto è 9900.0
Effettuato un deposito di 150.0
Il saldo del conto è 9950.0
Effettuato un deposito di 100.0
Il saldo del conto è 9900.0
Effettuato un prelievo di 200.0
Il saldo del conto è 9800.0
>
```

Corretto (il saldo finale),
ma messaggi
inconsistenti

Non è stato conteggiato
un deposito di 50.0!

Non è stato conteggiato
un deposito di 100.0!

Il problema è dovuto al fatto che le azioni evidenziate non sono atomiche, sono **interrompibili**; lo scheduler può mischiare l'esecuzione di thread differenti portando la risorsa condivisa in uno stato è inconsistente. Un aggiornamento distruttivo di una risorsa dovuto a una combinazione arbitraria di letture e scritture viene detto **interferenza**. La soluzione al problema è poter fornire accesso **mutuamente esclusivo** alla risorsa.

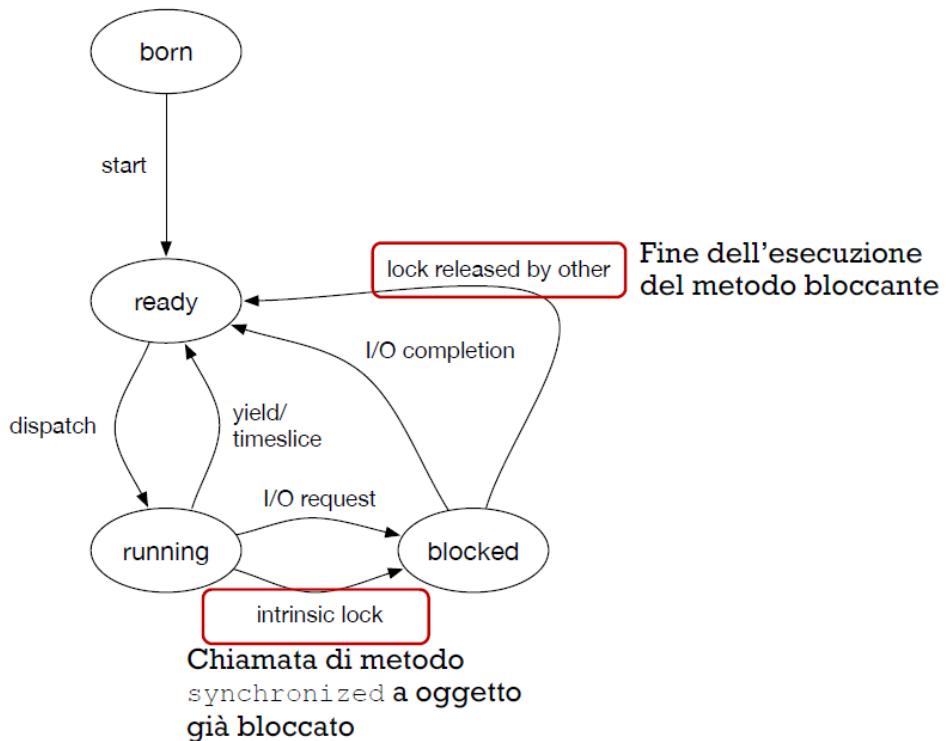
MUTUA ESCLUSIONE A OGGETTI JAVA

L'attivazione di metodi Java può essere mutuamente esclusiva tramite la specifica della parola chiave **synchronized**: Java associa un **intrinsic lock** ad ogni oggetto che abbia almeno un metodo **synchronized**; il fatto che un thread T1 sia in esecuzione all'interno di un metodo (o blocco) **synchronized** fa sì che altri thread che richiedano l'esecuzione di un altro metodo (o blocco) **synchronized** saranno messi in attesa che T1 completi l'esecuzione del metodo.

```
[...]
public synchronized void deposito(float cifra){
    saldo += cifra;
}

public synchronized float prelievo(float cifra){
    if(saldo > cifra){
        saldo -= cifra;
        return cifra;
    }
    float prelevati = saldo;
    saldo = 0.0f;
    return prelevati;
}
[...]
```

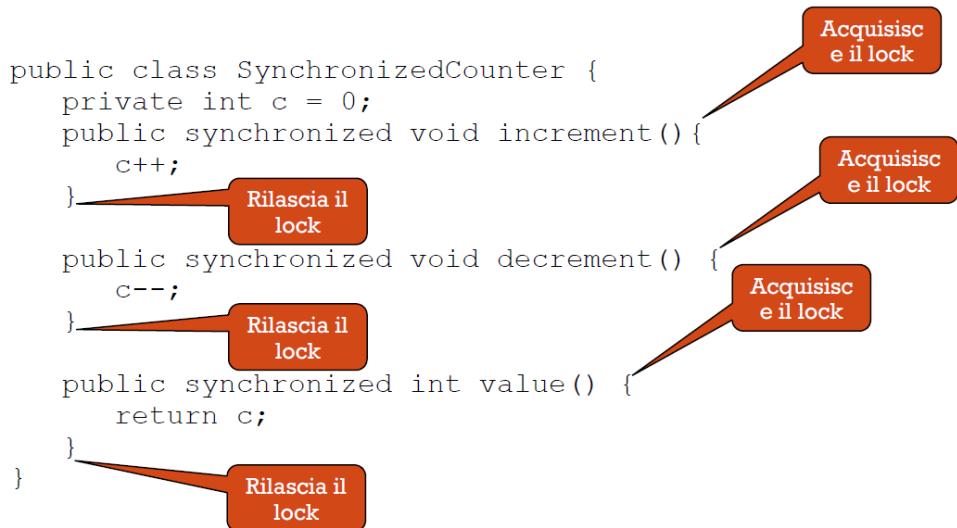
Di seguito è mostrato il **diagramma degli stati di un thread** rivisto per metodi *synchronized*



Esempio:

```

public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
  
```



Quando un metodo *synchronized* viene invocato:

- se l'oggetto **non è bloccato** (locked) (ossia nessun metodo *synchronized* è in esecuzione sull'oggetto), quest'ultimo viene bloccato e quindi il metodo viene eseguito;
- se l'oggetto è **bloccato**, il thread chiamante viene sospeso fino a quando quello “bloccante” non rilascia il lock;

Nota: un metodo *non synchronized* non viene mai bloccato.

L'introduzione della parola chiave *synchronized* nella dichiarazione dei metodi di una risorsa condivisa introduce un **ordinamento nell'esecuzione** di questi metodi da parte di thread concorrenti; quando un metodo *synchronized* termina, si stabilisce automaticamente una **relazione "happens-before"** con ogni invocazione successiva di altri metodi *synchronized* sullo stesso oggetto. Ai fini delle problematiche di interferenza, è come se questi metodi fossero stati resi atomici. L'acquisizione e il rilascio dell'intrinsic lock sono **completamente automatiche**, gestite dalla JVM, "trasparenti" al programmatore.

Naturalmente sincronizzare i metodi può essere vanificato se le risorse sono accessibili via **dot notation**: infatti sia il main che i thread potrebbero modificare le property dell'oggetto senza passare per le barriere della sincronizzazione. Se di norma è consigliabile, per vari motivi, dichiarare *private* gli attributi di un oggetto, in caso di risorse condivise in programmi concorrenti questo diventa davvero cruciale;

Dettagli su synchronized:

- Invocazione di un metodo *static synchronized*: il thread acquisisce l'intrinsic lock per il *Class object* associato alla classe; l'accesso ai campi *static* è controllato da un lock speciale, diverso da quelli associati alle istanze della classe
- I costruttori non possono essere *synchronized*: a meno di forzature (da evitare) solo il thread che crea l'oggetto può avere accesso ad esso mentre viene creato, in quanto l'oggetto non viene normalmente esibito (all'esterno del costruttore) prima che la sua costruzione e inizializzazione siano state completate

ACCESSO CONDIZIONATO ALLE RISORSE

Talvolta, quando per esempio una risorsa condivisa è caratterizzata da stati interni significativi (es. buffer, con dimensioni finite e impossibilità di prelevare se vuoto), vi è la necessità di definire delle condizioni di accesso alle risorse condivise. In questi casi l'uso di *synchronized* non è sufficiente per questo tipo di gestione, necessitiamo infatti di primitive per attendere il **verificarsi** di una condizione, e per **notificare** i thread in attesa.

Esempio:

```
public class Buffer {  
    private final int MaxBufferSize;  
    private char[] store;  
    private int BufferStart, BufferEnd,  
    BufferSize;  
    public Buffer(int size) {  
        MaxBufferSize = size;  
        BufferEnd = -1;  
        BufferStart = 0;  
        BufferSize = 0;  
        store = new char[MaxBufferSize];  
    }  
  
    public synchronized void insert(char ch) {  
        // Non posso inserire se il buffer è pieno!  
        [...]  
    }  
  
    public synchronized char delete() {  
        // Non posso cancellare se il buffer è  
        vuoto!  
        [...]  
    }  
}
```

Supponiamo di avere una risorsa condivisa caratterizzata da un buffer; per quanto i moderni elaboratori abbiano ingenti quantità di memoria **il buffer ha dimensione finita**; inoltre, se quest'ultimo è vuoto non è possibile prelevarne il contenuto.

I metodi di inserimento possono essere *synchronized* ma come anticipato questo non basta a gestire questo tipo di problema.

Ricapitolando quindi, vogliamo tener conto delle seguenti condizioni di accesso alla risorsa:

- Non inserire ed attendere finchè il buffer non è pieno;
- Non cancellare ed attendere finche è il buffer non è vuoto;

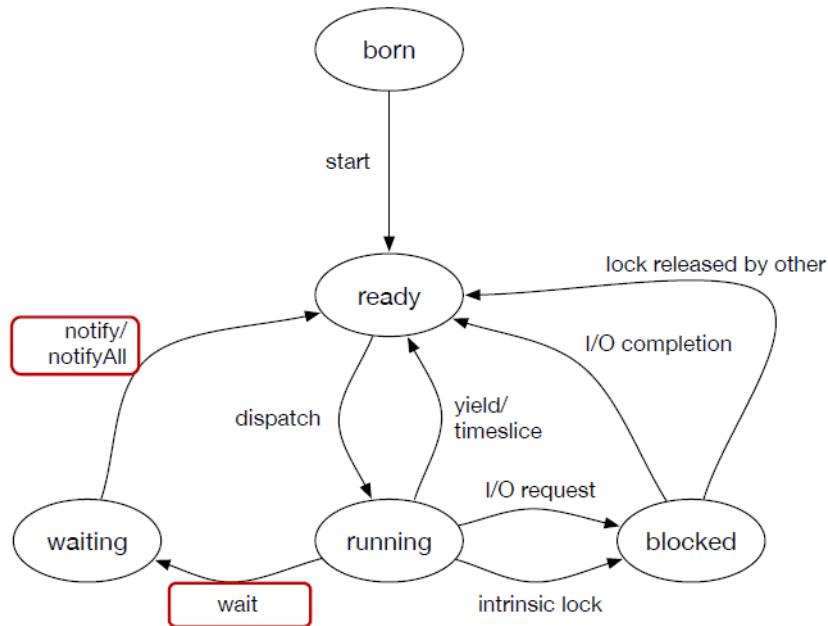
```
public class Buffer {  
    [...]  
    public synchronized void insert(char ch)  
    {  
        // Finché il buffer è pieno attendi  
        // Inserisci nel buffer  
        // Avvisa chi è in attesa sul buffer  
        [...]  
    }  
  
    public synchronized char delete() {  
        // Finché il buffer è vuoto attendi  
        // Cancella dal buffer  
        // Avvisa chi è in attesa sul buffer  
        [...]  
    }  
}
```

```
public synchronized void insert(char ch) {  
    try {  
        while(BufferSize == MaxBufferSize) {  
            wait();  
        }  
        BufferEnd = (BufferEnd + 1) % MaxBufferSize;  
        store[BufferEnd] = ch;  
        BufferSize++;  
        notifyAll();  
    } catch (InterruptedException e) {  
        System.out.println("Thread interrupted.");  
    }  
}  
  
public synchronized char delete() {  
    try {  
        while (BufferSize == 0) {  
            wait();  
        }  
        char ch = store[BufferStart];  
        BufferStart = (BufferStart + 1) % MaxBufferSize;  
        BufferSize--;  
        notifyAll();  
        return ch;  
    } catch (InterruptedException e) {  
        System.out.println("Thread interrupted.");  
        return '%';  
    }  
}
```

Il metodo **wait** rilascia il lock sull'oggetto e sospende il thread fino alla ricezione di un segnale di notifica sull'oggetto stesso da parte di un altro thread;

Il metodo **notify/notifyAll** risveglia un thread / tutti i thread che si erano messi in attesa volontaria tramite il metodo **wait**;

Di seguito viene mostrato il **diagramma degli stati di un thread** rivisto per metodi *wait* e *notify/notifyAll*



CONCETTO DI MONITOR

Il concetto di **monitor** dà conto di un costrutto (in Java oggetto) che consente ai thread di avere **mutua esclusione** nell'accesso a determinati blocchi di codice, di **poter bloccare** (far attendere) un thread in attesa del verificarsi di una certa condizione e di **poter segnalare** ai thread in attesa che le condizioni in merito al costrutto sono cambiate e va rivalutata la necessità di attendere ulteriormente. Se la gestione del lock intrinseco (mutua esclusione con metodi *synchronized*) è completamente trasparente, la gestione dei meccanismi di accesso condizionato ai metodi, che usino metodi *wait* e *notify/notifyAll* è **a carico del programmatore**.

Attenzione: un thread andato in attesa tramite *wait* può essere svegliato **solo** da una *notify/notifyAll*

Domanda: “*Di quante code di processi in attesa è dotato un monitor (qualsiasi oggetto Java dotato di almeno un blocco Synchronized)?*”

- Una per i thread bloccati dal meccanismo automatico/trasparente di gestione della mutua esclusione
- Una per i thread che si sono messi in attesa per via di una condizione necessaria a eseguire un metodo non ancora verificata

LIVENESS

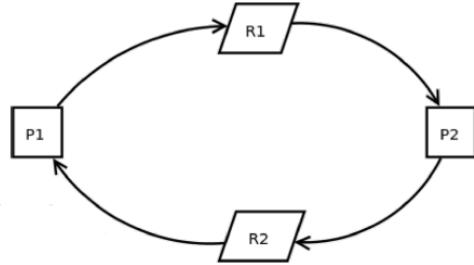
Nella programmazione concorrente, per **liveness** si intende il fatto che nonostante componenti concorrenti siano attivi, e debbano “fare a turno” in cosiddette **sezioni critiche** in un programma (porzioni di codice ad accesso controllato e ristretto, come metodi *sync*), il programma è in grado di progredire correttamente evitando particolari problemi; questo genere di proprietà è naturalmente desiderabile, ma per essere definita più precisamente richiede di specificare quali siano i problemi che si vogliono evitare.

Di seguito verranno discussi due tipi di problemi:

- ❖ **Deadlock**
- ❖ **Starvation**

DEADLOCK

In programmazione concorrente il **deadlock** (o stallo) è una situazione in cui due o più processi (nel nostro caso thread) o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione (es. rilasciare il controllo su una risorsa come un file, una porta input/output ecc.) che serve all'altro e viceversa. Un esempio è rappresentato da due persone che vogliono disegnare. Per disegnare hanno a disposizione solo una riga e una matita. Per disegnare hanno bisogno di entrambe. Potendo prendere un solo oggetto per volta, se uno prende la matita e l'altro prende la riga, e se entrambi aspettano che l'altro gli dia l'oggetto che ha in mano, i due generano un deadlock.



In pratica si tratta di **un'attesa circolare destinata a non terminare mai**.

Affinché un deadlock si presenti esistono delle condizioni necessarie, purtroppo spesso vere in sistemi concorrenti:

- **Mutua esclusione:** le risorse rilevanti per il sistema non devono essere condivisibili;
- **Hold and wait:** o accumulo incrementale: processi che sono in possesso di una risorsa possono richiederne altre senza rilasciare la prima;
- **No preemption:** una risorsa è rilasciabile solamente da chi la detiene, non dallo scheduler
- **Attesa circolare:** dato l'insieme delle azioni $A = \{A_1, A_2, \dots, A_N\}$, A_1 è in attesa di una risorsa da A_2 , che a sua volta è in attesa di una risorsa da A_3 ... ed A_n è in attesa di una risorsa da A_1

STARVATION



In programmazione concorrente, per **starvation** (termine inglese che tradotto letteralmente significa inedia) si intende l'impossibilità perpetua, da parte di un processo (nel nostro caso thread) pronto all'esecuzione, di ottenere le risorse sia hardware sia software di cui necessita per essere eseguito.

Un esempio tipico è il non riuscire ad ottenere il controllo della CPU da parte di processi con priorità molto bassa, qualora vengano usati algoritmi di scheduling a priorità. Può capitare, infatti, che venga continuamente sottomesso al sistema un processo con priorità più alta.

Anche una gestione scorretta della mutua esclusione o una definizione inadeguata dell'algoritmo delle attività date le circostanze (esempio della rotonda con traffico sbilanciato) potrebbe causare **starvation**. Questa condizione è più difficile da definire formalmente, e anche da prevenire.

Date queste proprietà possiamo adesso essere più precisi nel definire il concetto di **liveness**, almeno in alcune sue forme;

Una prima e più **debole definizione di liveness** è legata al concetto di **deadlock**: un sistema con diversi thread di controllo (in senso più ampio potrebbero essere anche processi) e una singola sezione critica è **libero da deadlock** se nonostante la competizione per l'accesso alla sezione critica **almeno un processo** riuscirà ad accedere a quest'ultima e a progredire nella sua esecuzione.

Una seconda e più **stringente definizione** è legata al concetto di **starvation**: un sistema, in circostanze analoghe al precedente è **libero da starvation** se garantisce che **tutti** i thread riescano ad accedere alla sezione critica e progredire nella loro esecuzione.

LIVELOCK

Il concetto di **livelock** è simile a quello di deadlock ma ancora più complicato da caratterizzare formalmente e quindi da poter gestire in modo automatico; l'idea è che in una certa situazione i membri di un gruppo di azioni (nel nostro caso i thread, ma la cosa vale anche per i processi) possono **non essere bloccati, ma ciononostante non progredire effettivamente**.

Immaginiamo due thread che devono “salutarsi” e hanno due modi per farlo, inchinarsi o stringersi la mano: se l'algoritmo prevede che provino alternativamente un modo o l'altro ciclicamente finché l'altro thread non saluta allo stesso modo, se non gestita in modo attento, questa situazione può portare ad un deadlock.

Diversi protocolli distribuiti hanno necessità di effettuare operazioni di sincronizzazione iniziale, anche detta **handshake**, quindi questo genere di problematica è meno caricaturale di quanto possa sembrare.

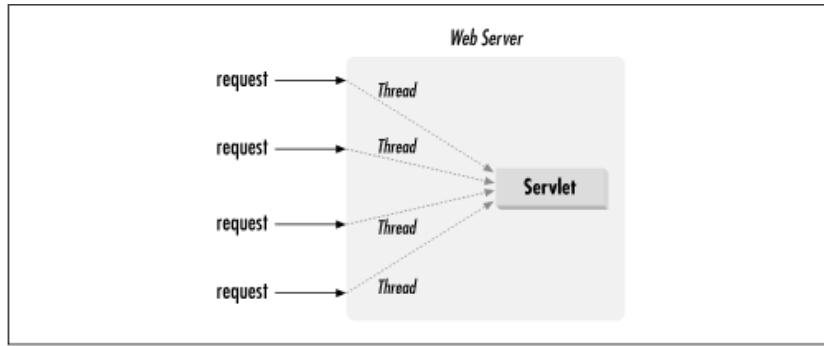
PROBLEMA LETTORI/SCRITTORI

Abbiamo una risorsa condivisa che può essere letta da più thread in contemporanea ma il cui accesso in scrittura deve invece essere esclusivo (per questioni di consistenza del dato). I thread sono classificati in base all'intenzione e del tipo di accesso. In sostanza la soluzione richiede che la lettura non sia interamente sincronizzata, solo il controllo che non ci siano scrittore deve esserlo; la scrittura deve essere invece completamente sincronizzata, ma non deve avvenire se anche un lettore sta effettuando il suo lavoro; nella risorsa condivisa è necessario mantenere due contatori, rispettivamente per il numero di lettori e scrittore attuali

```
public class Database {  
    private int content, readers, writeInt, writers;  
    Database(int content){  
        this.content = content;  
        readers = 0;  
        writers = 0;  
    }  
    public synchronized void prepareToRead(){  
        while(writers>0){  
            try {  
                wait();  
            } catch(InterruptedException e){}  
        }  
        readers++;  
        System.out.println("Readers = " + readers);  
    }  
    public int read(){  
        prepareToRead();  
        // Do the reading...  
        try{  
            Thread.sleep(50);  
        } catch(InterruptedException e){}  
        int contSnapshot = content;  
        doneReading();  
        return contSnapshot;  
    }  
    public synchronized void doneReading(){  
        readers--;  
        System.out.println("Readers = " + readers);  
        if(readers==0) notifyAll();  
    }  
    public synchronized void write(int content){  
        writeInt++; // At this point it is just an intention,  
                    // yet  
        while(readers>0){  
            try {  
                wait();  
            } catch(InterruptedException e){}  
        }  
        writers++;  
        System.out.println("Writers = " + writers);  
        this.content = content;  
        try{  
            Thread.sleep(100);  
        } catch(InterruptedException e){}  
        writers--;  
        writeInt--;  
        System.out.println("Writers = " + writers);  
        notifyAll();  
    }  
}
```

CONCORRENZA SERVLET

SERVLET E THREAD



Di norma un **servlet engine** (es. Tomcat) esegue tutte le servlet in una singola JVM, dunque queste ultime possono condividere risorse. Tuttavia questa possibilità può rappresentare un problema se non si presta attenzione alla gestione delle risorse.

Descriviamo tale problema partendo da un esempio:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SimpleCounter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        count++;
        out.println("Since loading, this servlet has been accessed " +
            count + " times.");
    }
}
```

Ci chiediamo dunque cosa succede quando due client lanciano la servlet in contemporanea: in tal caso il servlet engine crea, alla prima invocazione dell'endpoint associato, un oggetto servlet che **persiste** tra le diverse richieste; ciò significa che le ulteriori richieste di esecuzione della servlet non effettuano una nuova inizializzazione, poiché l'istanza di quest'ultima gestisce tutte le richieste effettuate tecnicamente da thread diversi.

Questa scelta viene adottata per diverse motivazioni, come l'utilizzo di poca memoria, la riduzione del costo di gestione (inizializzazione di molti oggetti che sarebbero spesso identici); inoltre tale metodo abilita la **persistenza (in memoria)** di risorse serenamente **condivisibili** tra diverse richieste (es. una connessione a DB).

Nel caso del nostro esempio vogliamo rendere il contatore (`count`) **thread safe**. Per fare ciò possiamo introdurre la proprietà vista per i thread in Java **synchronized**. In questo modo quando arrivano due richieste contemporanee, una delle richieste viene bloccata in modo da poter sincronizzare la sezione.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SimpleCounter extends HttpServlet {
    int count = 0;
    public synchronized void doGet(HttpServletRequest req,
        HttpServletResponse res) throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        count++;
        out.println("Since loading, this servlet has been accessed " +
            count + " times.");
    }
}
```

È facile accorgersi tuttavia che la sezione critica risulta spropositata rispetto al problema della gestione del contatore ...

Adottiamo quindi la seguente soluzione: applichiamo una zona critica ad un blocco di codice, invece che all'intero metodo *doGet*:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SimpleCounter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        int local_count;
        synchronized(this) {
            local_count = ++count;
        }
        out.println("Since loading, this servlet has been accessed " +
            local_count + " times.");
    }
}
```

SERVLET CONTEXT

Il servlet engine all'atto del deployment di un progetto (file WAR) crea un oggetto che implementa l'interfaccia *ServletContext*. Questa interfaccia è pensata per permettere la comunicazione tra una servlet e il container (l'engine), ma permette anche di accedere a informazioni di configurazione (incluse nel web.xml) e infine di realizzare forme di comunicazione anche inter-applicazione (tra diverse servlet gestite dallo stesso engine). Quest'ultima possibilità richiede una indicazione specifica in fase di configurazione dell'engine, mentre è sempre possibile usare il *ServletContext* per consentire la comunicazione tra diversi thread all'interno della stessa servlet.

Vediamo ora come **ottenere e usare il ServletContext**

```
public class ServletThreadSafety extends HttpServlet {  
    @Override  
    public void doGet(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
  
        resp.setContentType("text/html");  
        PrintWriter writer = resp.getWriter();  
  
        writer.println("test context attributes<br>");  
  
        synchronized(this.getServletContext()) {  
            this.getServletContext().setAttribute("foo", "22");  
            this.getServletContext().setAttribute("bar", "42");  
  
            writer.println(this.getServletContext().getAttribute("foo"));  
            writer.println(this.getServletContext().getAttribute("bar"));  
        }  
    }  
}
```

Ulteriore esempio:

```
import javax.servlet.ServletContext;  
import javax.servlet.ServletContextEvent;  
import javax.servlet.ServletContextListener;  
import javax.servlet.annotation.WebListener;  
  
import com.journaldev.db.DBConnectionManager;  
  
@WebListener  
public class AppContextListener implements ServletContextListener {  
  
    public void contextInitialized(ServletContextEvent servletContextEvent) {  
        ServletContext ctx = servletContextEvent.getServletContext();  
        String url = ctx.getInitParameter("DBURL");  
        String u = ctx.getInitParameter("DBUSER");  
        String p = ctx.getInitParameter("DBPWD");  
  
        //create database connection from init parameters and set it to context  
        DBConnectionManager dbManager = new DBConnectionManager(url, u, p);  
        ctx.setAttribute("DBManager", dbManager);  
        System.out.println("Database connection initialized for Application.");  
    }  
  
    public void contextDestroyed(ServletContextEvent servletContextEvent) {  
        ServletContext ctx = servletContextEvent.getServletContext();  
        DBConnectionManager dbManager = (DBConnectionManager) ctx.getAttribute("DBManager");  
        dbManager.closeConnection();  
        System.out.println("Database connection closed for Application.");  
    }  
}
```

Un **DBMANAGER o connector** è un oggetto ragionevolmente condivisibile tra diversi thread che “passano” nella servlet: in sé, questo oggetto non viene modificato, ma viene usato per creare nuove query da lanciare presso un DBMS, è a tutti gli effetti una risorsa condivisa solo in lettura. Se vogliamo condividere nel tier applicativo, senza tornare al tier dei dati persistenti, dobbiamo avere nel ServletContext qualcosa di più strutturato. Un buon candidato è una struttura dati **thread safe** come una **ConcurrentMap**: una volta creata, all’atto dell’inizializzazione del ServletContext, possiamo usarla senza doverci preoccupare della sincronizzazione, perché questa è fornita built-in.

I/O ASINCRONO IN JAVA (CENNI)

Prendiamo come riferimento il seguente Script:

```
$ (document).ready(function() {
    $("#search").click(function() {
        var title = $("#title").val();
        var URL = "http://www.omdbapi.com/";
        var ajaxURL = URL + "?t=" + title + "&apikey=MYKEYNOTSHOWN";
        $.getJSON(ajaxURL,function(data) {
            /* Do some stuff ...
        });
    });
}); // end ready

// Code here would be executed before the management of
// data returned by OMDB

}); // end click
}); // end ready
```

Questo script richiede un'operazione di I/O (getJSON) e dichiara che, una volta completata l'operazione, andrà richiamata una certa funzione anonima. Il codice non si blocca in attesa del completamento dell'operazione. Il fatto che il controllo venga ritornato dallo script al browser è importante perché altro codice JavaScript potrebbe richiedere il controllo (magari anche solo la gestione di alcuni timer per animazioni). Risulta quindi estremamente compatta la notazione per creare a tutti gli effetti un **listener** per il completamento dell'I/O, cosa fondamentale in Node.js.

Per quanto riguarda l'**I/O asincrono in Java** facciamo uso dell'API Java NIO (non-blocking I/O), le quali permettono di realizzare forme non bloccanti di operazioni di input e output tramite il concetto di canale, un'astrazione più a basso livello delle librerie precedenti, ma potenzialmente in grado di ottenere maggiore efficienza sfruttando le potenzialità della piattaforma sottostante. Il loro utilizzo è di conseguenza meno intuitivo delle librerie standard.

Esempio:

```
import java.nio.channels.AsynchronousFileChannel;
[ ... ]
Path path =
Paths.get("C:\\Java_Dev\\rainbow.txt");
AsynchronousFileChannel afc =
AsynchronousFileChannel.open(path, WRITE, CREATE);
```

OGGETTI FUTURE

Gli oggetti *Future* permettono di rappresentare **placeholder** nei quali i risultati dell'operazione saranno resi disponibili in un secondo momento. Questo consente di effettuare computazione nell'attesa, sebbene poi sia necessario effettuare un'operazione di polling qualora il risultato non sia ancora pronto.

Esempio:

```
ByteBuffer dataBuffer = a buffer;
long startPosition = 0;
Future<Integer> result = afc.write(dataBuffer,
startPosition);

[...]

while (!result.isDone()) {
    Thread.sleep(100);
}
int writtenNumberOfBytes = result.get();
```

Un altro modo di rappresentare oggetti Future è il **listener**: viene definita una classe che implementa un *CompletionHandler* che gestisce le operazioni di completamento o fallimento dell'I/O.

RDF

Il **Resource Description Framework (RDF)** è lo strumento base proposto da W3C per la codifica, lo scambio e il riutilizzo di metadati strutturati e consente l'interoperabilità semantica tra applicazioni che condividono le informazioni sul Web. È costituito da due componenti:

- **RDF Model and Syntax:** espone la struttura del modello RDF, e descrive una possibile sintassi;
- **RDF Schema:** espone la sintassi per definire schemi e vocabolari per i metadati.

L'RDF Data Model si basa su tre principi chiave:

1. Qualunque cosa può essere identificata da un **Uniform Resource Identifier (URI)**;
2. **The least power:** utilizzare il linguaggio meno espressivo per definire qualcosa;
3. Qualunque cosa può dire qualunque cosa si qualunque cosa;

PRINCIPI E MODELLI DI DATI

Qualunque cosa descritta da RDF è detta **risorsa**. Principalmente una risorsa è reperibile sul web, ma RDF può descrivere anche risorse che non si trovano direttamente sul web. Ogni risorsa è identificata da un **URI**.

Il modello di dati RDF è formato da **risorse, proprietà e valori**. Le proprietà sono delle relazioni che legano tra loro risorse e valori, e sono anch'esse identificate da URI. Un valore, invece, è un tipo di dato primitivo, che può essere una stringa contenente l'URI di una risorsa.



L'unità base per rappresentare un'informazione in RDF è lo **statement**. Uno statement è una **tripla** del tipo **Soggetto – Predicato – Oggetto**, dove il soggetto è una **risorsa**, il predicato è una **proprietà** e l'oggetto è un **valore** (e quindi anche un URI che punta ad un'altra risorsa). Il data model RDF permette di definire un modello semplice per descrivere le relazioni tra le risorse, in termini di proprietà identificate da un nome e relativi valori. Tuttavia, RDF data model non fornisce nessun meccanismo per dichiarare queste proprietà, né per definire le relazioni tra queste proprietà ed altre risorse. Tale compito è definito da **RDF Schema**.

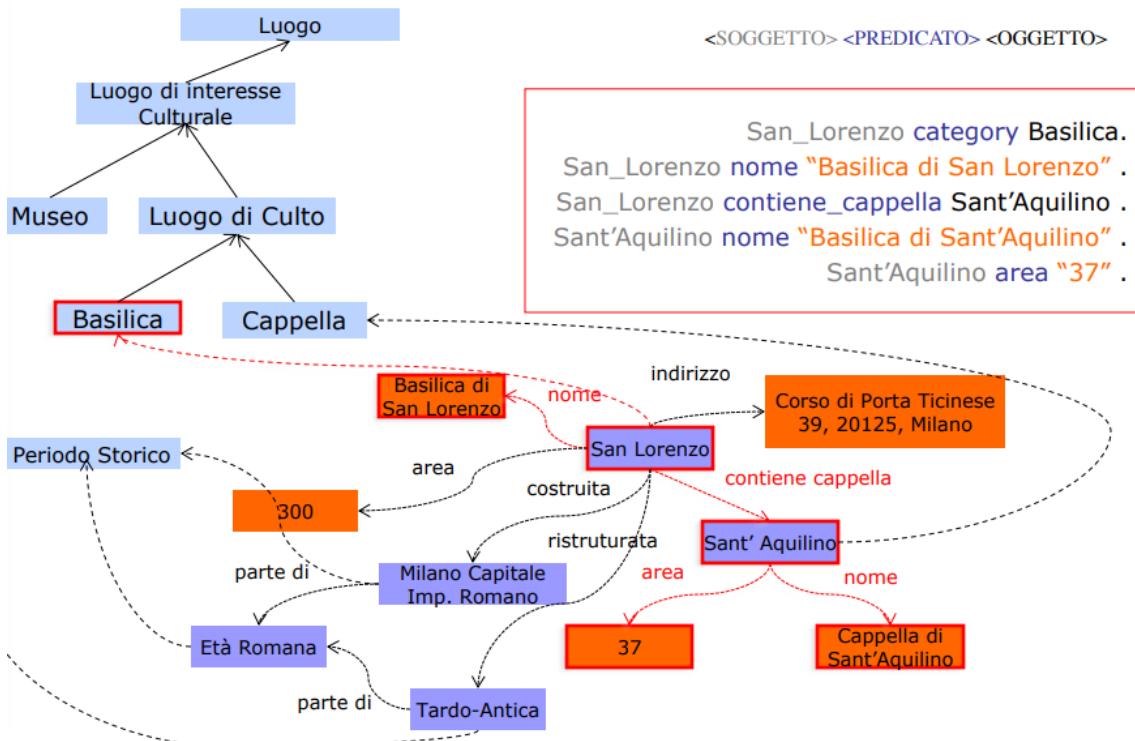
RDF CONTAINER

RDF quando deve far riferimento a più di una risorsa, per esempio per descrivere il fatto che la risorsa è associata a più proprietà, definisce dei contenitori (**container**), ossia liste di risorse. Tre sono i tipi di contenitori:

- **Bag:** è una lista non ordinata di risorse o costanti. Viene utilizzato per dichiarare che una proprietà ha valori multipli. Per esempio i componenti di un convegno;
- **Sequence:** differisce da Bag per il fatto che l'ordine delle risorse è significativo. Per esempio si vuole mantenere l'ordine alfabetico di un insieme di nomi, gli autori di un sito;
- **Alternative:** è una lista di risorse che definiscono un'alternativa per il valore singolo di una proprietà. Per esempio per fornire titoli alternativi in varie lingue;

RAPPRESENTAZIONE FISICA DEL MODELLO

Un modello RDF è rappresentabile da un **grafo orientato** sui cui nodi ci sono risorse o tipi primitivi e i cui archi rappresentano le proprietà. Un **grafo orientato** D è una coppia D = (V, A) dove V è un insieme di vertici e A è un insieme di archi orientati di D. Un **arco orientato** è un arco caratterizzato da una **direzione**.



Un **grafo RDF** è rappresentato fisicamente mediante una **serializzazione**. Le principali serializzazioni adattabili per un grafo RDF sono:

- ❖ **RDF/XML**: documento RDF è serializzato in un file XML;
- ❖ **N-Triples**: serializzazione del grafo come un insieme di triple **soggetto - predicato - oggetto**.
- ❖ **Notation3**: serializzazione del grafo descrivendo, una per volta, una risorsa e tutte le sue proprietà.

In particolare la serializzazione in XML può avvenire secondo due metodi, quello classico e quello abbreviato, più leggibile per l'uomo.

Esempio: Si supponga di voler serializzare la frase

"Mario_Rossi" "è_autore_di" "Rosso_di_sera_bel_tempo_si_spera": il risultato in RDF/XML sarà:

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:au="http://description.org/schema/">
  <rdf:Description about="http://www.book.it/Rosso_di_sera_bel_tempo_si_spera/">
    <au:author>Mario_Rossi</au:author>
  </rdf:Description>
</rdf:RDF>

```

RDF SCHEMA

In **RDF Schema (RDFS)** ogni predicato è in relazione con altri predici e permette di dichiarare l'esistenza di proprietà di un concetto, che permettano di esprimere con metodo sistematico affermazioni simili su risorse simili. RDF Schema permette di definire nuovi tipi di classe. Inoltre specificando il concetto di classe e sottoclasse, consente di definire gerarchie di classi. In RDF si possono rappresentare le risorse come istanze di classi e definire sottoclassi e tipi.

Classi RDF

Ogni risorsa descritta in RDF è istanza della classe *rdfs:Resource*.

Le sottoclassi di *rdfs:Resource* sono:

- *rdfs:Literal* Rappresenta un letterale, una stringa di testo.
- *rdfs:Property* Rappresenta le proprietà.
- *rdf:Class* Una classe dei linguaggi object-oriented.

Proprietà RDF

- *rdf:type* Indica che una risorsa è del tipo della classe che viene specificata.
- *rdfs:subClassOf* Indica la relazione classe/sottoclasse fra due classi.

L'ereditarietà può essere multipla.

- *rdfs:subPropertyOf* Indica che una proprietà è specializzazione di un'altra.
- *rdfs:seeAlso* Specifica che la risorsa è anche descritta in altre parti.
- *rdfs:isDefinedBy* Indica la risorsa "soggetto dell'asserzione" ovvero chi ha fatto l'asserzione.

Vincoli RDF

- *rdfs:range* (codominio) È utilizzato come proprietà di una risorsa; indica le classi che faranno parte di una asserzione con la proprietà.
- *rdfs:domain* (dominio) Indica la classe a cui può essere applicata la proprietà.

Esempio: la classe gatto viene dichiarata sottoclasse della classe animale

```
<rdf:Description rdf:ID="Animale">
  <rdf:type
    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>

<rdf:Description rdf:ID="gatto">
  <rdf:type
    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassof rdf:resource="#Animale"/>
</rdf:Description>
```

CARATTERISTICHE DI RDF

- **Indipendenza**, i predicati sono risorse e sono utilizzabili e creabili da chiunque;
- **Intercambiabilità**, RDF può essere convertito in XML;
- **Scalabilità**, grazie al formato a triple le proprietà sono facili da gestire;
- Le **proprietà sono risorse**, e possono avere loro stesse delle proprietà;
- I **soggetti e gli oggetti sono risorse**, e possono avere loro stesse delle proprietà.

TURTLE/N3 NOTATION

Turtle ("Terse RDF Triple Language") è un formato ideato per esprimere dati di tipo **RDF** con una sintassi adatta a SPARQL. Secondo le convenzioni RDF, le informazioni sono rappresentate per mezzo di "triple", ciascuna delle quali consiste di un soggetto, un predicato e un oggetto. Ognuno di questi elementi è espresso come URI.

La notazione Turtle risulta più compatta e leggibile della notazione XML RDF

Esempio:

L'esempio seguente definisce tre prefissi ("rdf", "dc", and "ex") e li utilizza per esprimere affermazioni sulla redazione della specifica RDF/XML:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix ex: <http://example.org/stuff/1.0/> .  
  
<http://www.w3.org/TR/rdf-syntax-grammar>  
  dc:title "RDF/XML Syntax Specification (Revised)" ;  
  ex:editor [  
    ex:fullname "Dave Beckett";  
    ex:homePage <http://purl.org/net/dajobe/>  
  ] .
```

L'esempio si riferisce ad un grafo RDF di quattro triple, le quali esprimono i seguenti fatti:

- La relazione tecnica del W3C sulla sintassi e grammatica RDF ha come titolo "RDF/XML Syntax Specification (Revised)".
- L'editore della relazione è un certo individuo, il quale a sua volta:
 - Ha come nome "Dave Beckett"
 - Ha la sua homepage presso l'indirizzo "http://purl.org/net/dajobe/"

Segue l'elenco delle tre triple rese esplicite, in notazione N-Triples:

```
<http://www.w3.org/TR/rdf-syntax-grammar> <http://purl.org/dc/elements/1.1/title> "RDF/XML Syntax Specification (Revised)" .  
<http://www.w3.org/TR/rdf-syntax-grammar> <http://example.org/stuff/1.0/editor> _:bnode .  
_:bnode <http://example.org/stuff/1.0/fullname> "Dave Beckett" .  
_:bnode <http://example.org/stuff/1.0/homePage> <http://purl.org/net/dajobe/> .
```

Dove `_:bnode` è una **risorsa anonima**.

Nota: le triple con lo stesso soggetto possono essere abbreviate con `";"`. Le triple con lo stesso soggetto e predicato possono essere abbreviate con `";"`.

-----o-----o-----

FOAF

FOAF (acronimo di **friend of a friend** - Amico di un amico) è un vocabolario descrittivo espresso in **Resource Description Framework (RDF)** ed è definita usando Web Ontology Language (**OWL**). I computer possono usare FOAF, ad esempio, per cercare tutte le persone che vivono in Europa, o tutte le persone che hanno un tuo amico in comune, questo appunto perché permette di definire le relazioni tra persone. Ogni profilo ha un identificativo univoco (come ad esempio l'indirizzo email, l'URI dell'homepage o del blog della persona) che viene utilizzato quando definisci queste relazioni.

Esempio:

Il seguente profilo FOAF (scritto in formato XML) parla di Jimmy Wales, il suo indirizzo e-mail, la sua homepage e la sua fotografia sono delle risorse. Lui ha interesse in Wikipedia e conosce Angela Beesley (che è il nome della risorsa 'Persona').

```
<rdf:RDF  
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
  xmlns:foaf="http://xmlns.com/foaf/0.1/"  
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">  
<foaf:Person rdf:about="#JW">  
  <foaf:name>Jimmy Wales</foaf:name>  
  <foaf:mbox rdf:resource="mailto:jwales@bomis.com" />  
  <foaf:homepage rdf:resource="http://www.jimmywales.com/" />  
  <foaf:nick>Jimbo</foaf:nick>  
  <foaf:depiction rdf:resource="http://www.jimmywales.com/aus_img_small.jpg" />  
  <foaf:interest>  
    <rdf:Description rdf:about="http://www.wikimedia.org" rdfs:label="Wikipedia" />  
  </foaf:interest>  
  <foaf:knows>  
    <foaf:Person>  
      <foaf:name>Angela Beesley</foaf:name>  
    </foaf:Person>  
  </foaf:knows>  
</foaf:Person>  
</rdf:RDF>
```

E di seguito lo stesso esempio nel formato **Turtle**:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  
  
<#JW>  
    a foaf:Person ;  
    foaf:name "Jimmy Wales" ;  
    foaf:mbox <mailto:jwales@bomis.com> ;  
    foaf:homepage <http://www.jimmywales.com/> ;  
    foaf:nick "Jimbo" ;  
    foaf:depiction <http://www.jimmywales.com/aus_img_small.jpg> ;  
    foaf:interest <http://www.wikimedia.org> ;  
    foaf:knows [  
        a foaf:Person ;  
        foaf:name "Angela Beesley"  
    ] .  
  
<http://www.wikimedia.org>  
    rdfs:label "Wikimedia" .
```

SPARQL

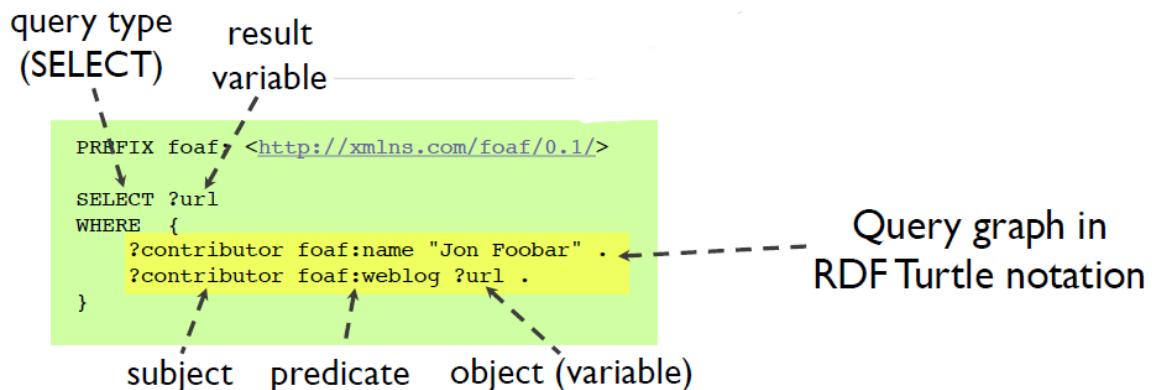
SPARQL (acronimo ricorsivo di **SPARQL Protocol and RDF Query Language**) è un linguaggio di interrogazione per dati rappresentati tramite il **Resource Description Framework (RDF)**. **SPARQL** è uno degli elementi chiave delle tecnologie legate al paradigma noto come web semantico, e consente di estrarre informazioni dalle basi di conoscenza distribuite sul web. Il linguaggio RDF descrive i concetti e le relazioni su di essi attraverso l'introduzione di triple (soggetto-predicato-oggetto), e consente la costruzione di query basate su triple patterns, congiunzioni logiche, disgiunzioni logiche, e pattern opzionali.

Esempio:

Un esempio di interrogazione SPARQL che modella la domanda: "Quali sono tutte le capitali in Africa?":

```
PREFIX abc: <http://example.com/exampleOntology#>
SELECT ?capital ?country
WHERE {
  ?x abc:cityname ?capital ;
      abc:isCapitalOf ?y .
  ?y abc:countryname ?country ;
      abc:isInContinent abc:Africa . }
```

Vediamo in particolare il formato di una query SPARQL



Le query sopra mostrate rappresentano query di tipo `SELECT`, il quale risultato sarà mostrato in una tabella; è tuttavia possibile formulare query booleane le quali mostreranno un risultato positivo o negativo.

Esempio:

```
ASK
WHERE {
  book: 0001 book:author author:01
}
```

La query ritornerà TRUE se il grafo della query rispecchierà quello dei dati.

Esempio: mostriamo un ulteriore esempio con il relativo risultato

Data graph

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
_:author1 <http://www.w3.org/2001/vcard-rdf/3.0#FN> "J.K. Rowling".  
<http://example.org/book/book1> dc:title "Harry Potter and the Chamber of Secrets".  
<http://example.org/book/book1> dc:creator _:author1 .  
<http://example.org/book/book2> dc:title "Harry Potter and the Prisoner Of Azkaban" .  
<http://example.org/book/book2> dc:creator _:author1 .
```

SPARQL Query

```
PREFIX books: <http://example.org/book/>  
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>  
SELECT ?bookTitle ?authorName  
WHERE  
{ ?book dc:creator ?author .  
?book dc:title ?bookTitle .  
?author vcard:FN ?authorName  
}
```

Results

bookTitle	authorName
Harry Potter and the Chamber of Secrets	J.K. Rowling
Harry Potter and the Prisoner Of Azkaban	J.K. Rowling

Esempio: mostriamo ora un esempio con l'utilizzo di **valori costanti**

```
ASK  
WHERE {  
?author vcard:FN "J.K. Rowling" .  
}
```

La query sopra riportata ritorna true se c'è un autore con il nome "J.K. Rowling".

Nota: SPARQL fornisce operazioni per testare le stringhe, basate su espressioni regolari la cui sintassi differisce da quella di "LIKE" dell'SQL; il linguaggio di queste ultime è **xQuery**, la versione codificata che possiamo trovare nel linguaggio **Perl**;

Esempio: di seguito viene mostrato un esempio con l'utilizzo di **FILTER**

```
ASK  
WHERE {  
?author vcard:FN ?name .  
FILTER regex(?name, "rowling", "i")  
}
```

La query sopra riportata ritorna true se un nome di un autore **contiene** la stringa "rowling" (**Filter Pattern**). Per introdurre il case sensitive si utilizza il **Filter Flag** "i".

Esempio: un ulteriore esempio con l'utilizzo di **FILTER**

```
ASK
WHERE {
    person:0001 info:age ?age .
    FILTER (?age >= 18)
}
```

In questo caso viene applicato **FILTER** ad una proprietà di tipo *integer*. La query ritorna true se c'è una persona 0001 la cui età è ≥ 18 .

SPARQL permette di scrivere query per dati e di non fallire quando questi ultimi non esistono.

Esempio: segue un esempio di utilizzo di **OPTIONAL**

```
SELECT ?name ?age
WHERE {
    ?person info:name ?name .
    OPTIONAL { ?person info:age ?age . FILTER ( ?age > 18 ) }
}
```

La query sopra riportata ritorna il nome e l'età di ogni entità con età **sconosciuta** o **>=18**.

DBPEDIA

DBpedia è un progetto in corso, nato nel 2007, per l'estrazione di informazioni strutturate da Wikipedia e per la pubblicazione di queste informazioni sul Web come **Linked Open Data** in formato **RDF (Resource Description Framework)** attraverso interrogazioni (query **SPARQL**).

The screenshot shows the Virtuoso SPARQL Query Editor interface. The browser address bar displays "dbpedia.org/sparql". The main window title is "Virtuoso SPARQL Query Editor". The "Default Data Set Name (Graph IRI)" field contains "http://dbpedia.org". The "Query Text" field contains the following SPARQL query:

```
select distinct ?Concept where { [] a ?Concept} LIMIT 100
```

Below the query text, there are settings for "Results Format" (set to "HTML"), "Execution timeout" (set to 0 milliseconds), and "Options" (with "Strict checking of void variables" checked). At the bottom, there are "Run Query" and "Reset" buttons, and a search bar with the text "Find: win".

Esempio: segue un esempio di ricerca su **DBpedia** da un'applicazione web

```
<script src="../../js/jquery-1.7.2.min.js"></script>
<script>
$(document).ready(function() {
    var URL = "http://dbpedia.org/sparql?default-graph-uri=http://dbpedia.org&query=";
    var query = "select ?name ?abst where " +
    "{_:quart dct:subject <http://dbpedia.org/resource/Category:Districts_of_Milan>;" +
    "foaf:name ?name;" +
    "dbo:abstract ?abst filter(lang(?abst) = 'it')" +
    "}";
    var options = "&debug=on&timeout=&format=json&save=display&fname=";
    var ajaxURL = URL + query + options;
    $.getJSON(ajaxURL, {}, function(data) {
        $.each(data.results.bindings, function(i,row){
            $("div.data").append('<h2><div class="districtName">' +
                row.name.value + '</div></h2>');
            $("div.data").append('<div class="districtAbs"><p>' +
                row.abst.value + '</p></div>');
        }); // end each
        $("div.data").append('<br>');
    }); // end get JSON
});
// end ready
</script>
```

FUTURE INTERNET

PIATTAFORME PROGRAMMABILI

Attualmente si tende a fare affidamento a piattaforme condivise che forniscono meccanismi che garantiscono **interoperabilità** (ad esempio http / TCP + REST), e che supportano specifiche politiche di utilizzo (piattaforme programmabili). Qui entra in gioco il **cloud computing**. Quest'ultimo è uno stile di computing che fornisce scalabilità e capacità elastiche relative all' IT come servizio esterno utilizzabile da utenti esterni.

L'approccio è quello di sviluppare componenti come **microservices**, e **containers** come componenti di sviluppo.

NIST

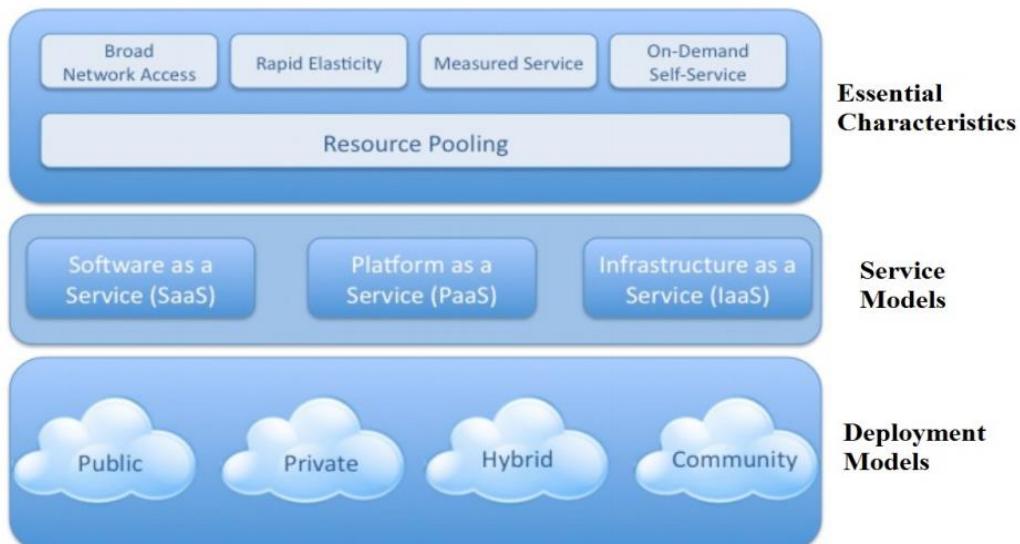
Segue la definizione di cloud data da **NIST**:

Il cloud computing è un modello per abilitare l'accesso alla rete onnipresente, conveniente e **on-demand** a un pool condiviso di risorse di calcolo configurabili (ad esempio reti, server, storage, applicazioni e servizi) che possono essere rapidamente fornite e rilasciate con il minimo sforzo di gestione o interazione con il fornitore di servizi.

Questo tipo di cloud appena descritto è composto da 5 caratteristiche essenziali:

- **On demand self-service:** un consumatore può fornire unilateralmente funzionalità di calcolo come archiviazione di rete su richiesta automaticamente, senza l'interazione umana con un provider cloud;
- **Ampio accesso alla rete:** le funzionalità sono disponibili sulla rete e sono accessibili tramite meccanismi standard che promuovono l'utilizzo da piattaforme client eterogenee thin o thick (ad es. Telefoni cellulari, laptop e PDA), nonché altri servizi software tradizionali o basati su cloud.
- **Pool di risorse:** le risorse di elaborazione del provider sono raggruppate per servire più consumatori utilizzando un modello multi-tenant, con risorse fisiche e virtuali diverse assegnate dinamicamente e riassegnate in base alla domanda dei consumatori.
- **Rapida elasticità**
- **Servizi su misura**

Di seguito è mostrata una rappresentazione grafica del modello **NIST**

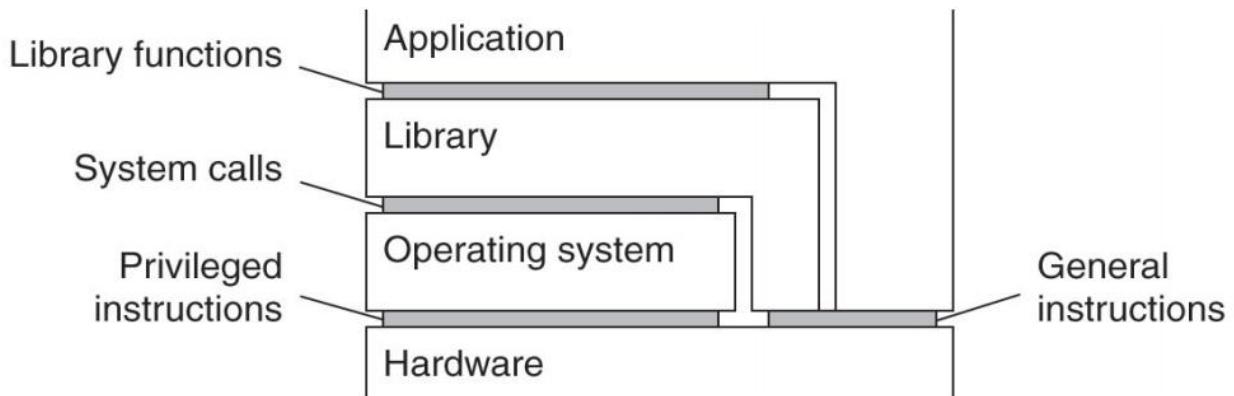


TIPI DI CLOUD

- **Public cloud:** l'infrastruttura è resa disponibile al grande pubblico o a un grande gruppo industriale ed è di proprietà di un'organizzazione che vende servizi cloud; l'accesso viene eseguito per abbonamento (pay-as-you-go)
- **Private cloud:** l'infrastruttura cloud è gestita esclusivamente per una singola organizzazione. Può essere gestito dall'organizzazione o da terze parti e può esistere in sede o fuori sede. L'accesso è limitato ai soli partner e/o autorizzati
- **Community cloud:** l'infrastruttura cloud è condivisa da diverse organizzazioni e supporta una specifica community che ha interessi condivisi (ad es. requisiti di sicurezza, policy o considerazioni di conformità). Può essere gestito dalle organizzazioni o da terze parti e può esistere in sede o fuori sede.
- **Hybrid cloud:** l'infrastruttura cloud è una composizione di due o più cloud (private, community o public) che rimangono entità uniche ma sono unite da una tecnologia standardizzata o proprietaria che consente la portabilità di dati e applicazioni (ad es. Cloud bursting per il bilanciamento del carico tra cloud).

VIRTUALIZZAZIONE

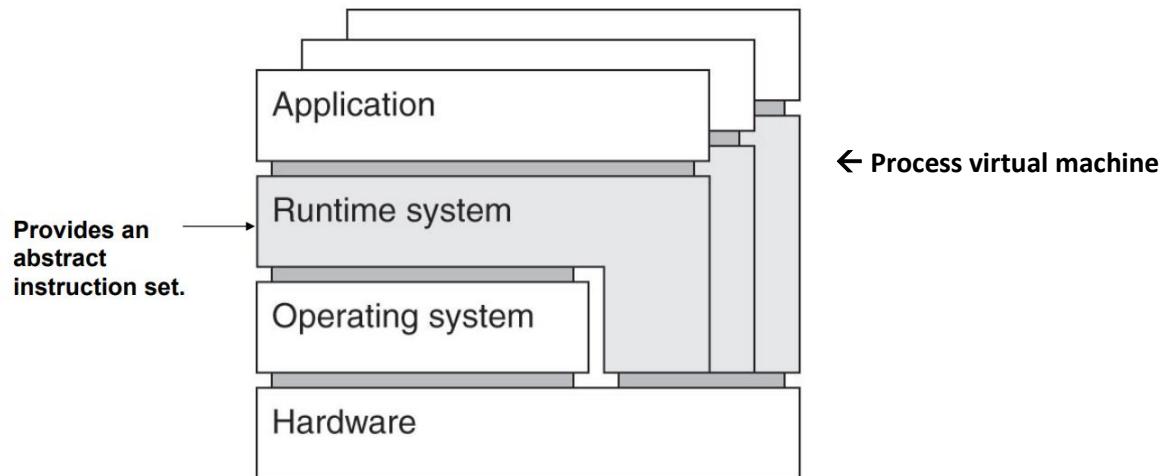
La virtualizzazione consente l'introduzione di nuove funzionalità di sistema senza aggiungere complessità hardware e software complessi già esistenti. I recenti progressi nelle tecnologie di virtualizzazione possono essere applicati a diverse aree di ricerca come la gestione delle risorse virtualizzate, il monitoraggio e il ripristino degli attacchi in Grid computing.



Esistono principalmente due tipi di virtualizzazione

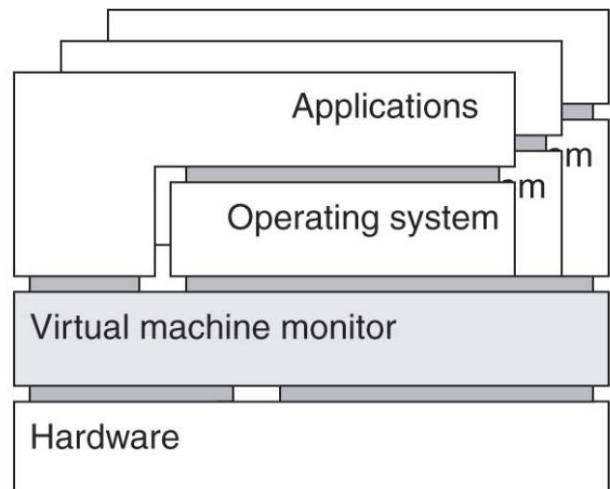
- ⊕ **Process virtual machine:** virtualizzazione attraverso interpretazione o emulazione; solitamente viene fatto per un singolo processo. Questo tipo di virtualizzazione fornisce un set di istruzioni macchina astratto; i programmi sono compilati con questo codice "macchina" che viene quindi interpretato (ad es. Java) o emulato;
- ⊕ **Virtual machine monitor:** questo tipo di virtualizzazione è in grado di fornire una macchina virtuale a molti programmi diversi contemporaneamente, come se fossero in esecuzione più CPU su una singola piattaforma. Questa virtualizzazione risulta particolarmente importante per i sistemi distribuiti poiché le applicazioni sono isolate e gli errori sono limitati a una singola macchina virtuale. (Esempi di questo tipo di virtualizzazione: VMware e Xen);

Di seguito viene mostrata l'architettura tipica di una macchina virtuale per i due modelli sopracitati

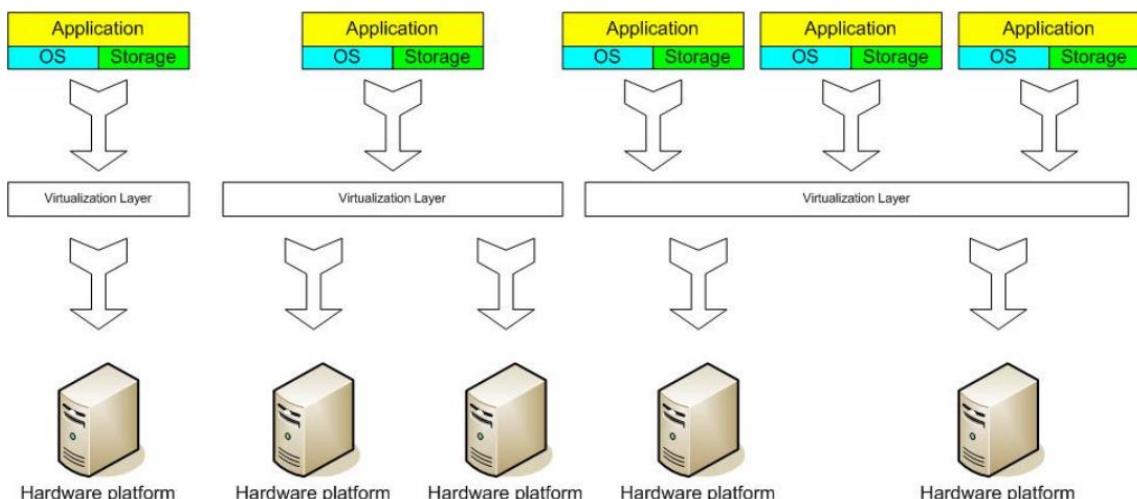


Virtual machine monitor →

The hardware is completely hidden by the VMM



CONCETTO DI VIRTUAL SERVER



Virtual Machine Monitor (VMM) layer between *Guest OS* and hardware

MICROSERVICES

Il web è di fatto l'infrastruttura per la comunicazione: la comunicazione è regolata da protocolli, e le applicazioni sono costruite secondo le linee guida **REST**.

L'architettura a **microservizi** consiste nel mettere ogni elemento che abbia una funzione in un servizio separato, e rendere i servizi indipendenti aumentando la coesione e diminuendo l'accoppiamento. I servizi sono asincroni e seguono i principi della coreografia; il controllo non è centralizzato, e l'infrastruttura è automatizzata.

I microservizi minimizzano la comunicazione e la coordinazione tra parti riducendo l'impatto dei cambiamenti. Similmente a SOA sono un set di servizi, ma i microservizi non hanno specifiche web e favoriscono REST per la sua semplicità.

APPLICAZIONI MONOLITICHE

Le **applicazioni monolitiche** invece usano la logica di business come core dell'applicazione la quale viene implementata con moduli circondati da adattatori che si interfacciano con il mondo esterno. I monoliti hanno grandi dimensioni, sono difficili da mantenere, non sempre affidabili, e i cambiamenti causano problemi con le dipendenze, necessità di rebooting e sono costosi.

I microservizi al contrario offrono un set di servizi più piccoli e interconnessi tramite gateway, che si occupa di gestire le connessioni e gli accessi. Alcuni microservizi contengono inoltre un'interfaccia API e una web UI; a runtime, ogni istanza è rappresentata da una VM su un cloud o un container Docker. La responsabilità è singola e encapsulata, e ogni servizio ha il proprio schema di database.

VANTAGGI E SVANTAGGI DEI MICROSERVIZI

Il grande vantaggio dei microservizi è la poca complessità, ottenuta dividendo lo sviluppo dei microservizi e dei relativi servizi: ciò aumenta la scalabilità, non avendo un numero fisso di servizi e diversi tipi di hardware. Ogni microservizio non dipende dagli altri.

Lo svantaggio è la ridotta dimensione dei servizi, insieme alla difficoltà nell'implementazione del sistema distribuito e dell'architettura con database partizionati (dati duplicati che devono essere aggiornati a ogni transizione). Ogni servizio ha bisogno di essere configurato e monitorato con un alto livello di automazione.

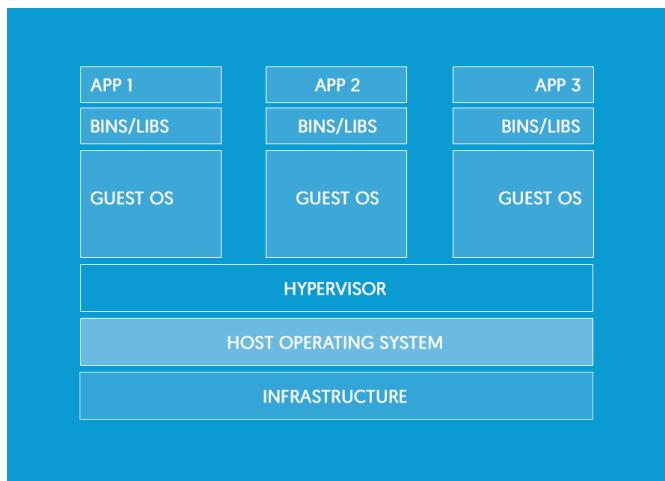
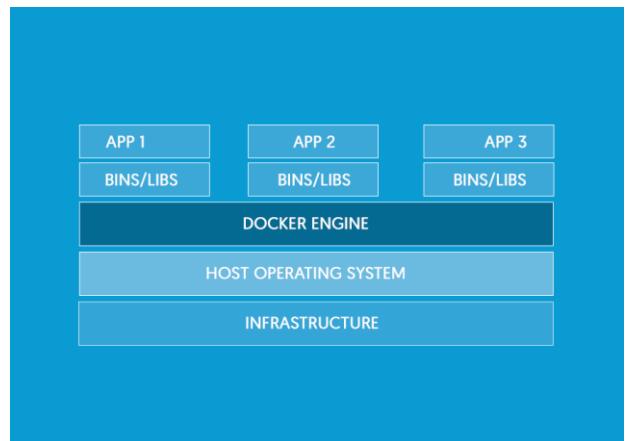
DOCKERS

I **containers** sono un modo standard di raggruppare un'applicazione e tutte le sue dipendenze, per poterle spostare ed eseguire in un ambiente standard esterno. Le unità non sono legate a un'infrastruttura, e le applicazioni sono più sicure e performanti.

VMM MODEL VS CONTAINER MODEL

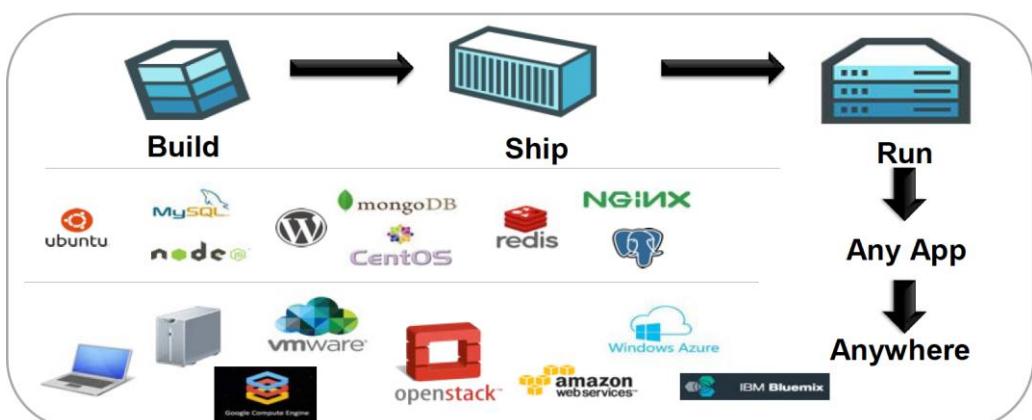
Container: le applicazioni vengono fornite con tutte le loro dipendenze in un'unità standardizzata per lo sviluppo e la distribuzione del software;

Sono unità più leggere e indipendenti dall'infrastruttura specifica. Le applicazioni nei container sono potenzialmente più sicure e performanti



VMM: le applicazioni dipendono dal sistema operativo guest

Docker è una piattaforma open per rendere più leggero ed efficiente l'uso di containers, con possibilità di diversi ambienti e istanze per il testing. Si appoggia su infrastruttura PaaS e permette di sviluppare applicazioni SaaS.



I principali **vantaggi** nell'uso di container Docker sono che consentono di definire un'unità standard di deployment e che isolano dagli elementi esterni un'applicazione e i tool a essa collegati.

Docker Engine è un'applicazione client-server, installata con Docker Machine e composta da un server che esegue un demone, una API REST per comunicare con i programmi e una CLI (Command Line Interface) per i comandi. Il client e il demone possono essere locali e remoti, e la comunicazione via API è tramite interfaccia o socket di Unix.

Un'**applicazione Docker** può essere integrata in un cluster o in uno **swarm** se è composto da più macchine collegate, il quale può essere fisico o virtuale e segue i comandi di uno swarm manager.