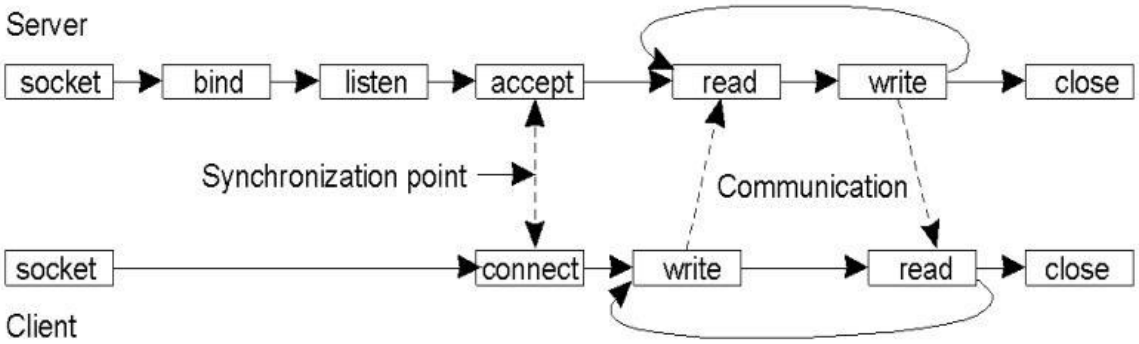
Connessione TCP via socket



Tipicamente la programmazione di sistemi informatici è appoggiarsi su qualcosa che già c'è, noi abbiamo dei servizi offerti dal SO (Linux, Windows, macOS). I SO hanno a disposizione una libreria abilitante per la comunicazione in rete, tali SO implementano il protocollo TCP (con buffer e variabili). Chi programma Chrome/Firefox ecc parla con lo strato che il SO offre per usare TCP, ovvero lo **strato delle socket** (che sono di tipo diverso, ci sono per TCP ma anche per altri protocolli).
Il server ha il compito di rispondere al client ma è pronto prima del client stesso

- **Server**: ha il compito di rispondere al client, ma è pronto prima del client stesso
  - Dichiara un oggetto di tipo ***Socket***
  - Avviene una operazione di ***bind***, che associa una certa porta alla socket perché per il server può essere importante usare una well known port (porta nota). Se c'è già qualcuno che ascolta su quella porta il sistema operativo deve avvisare. Questo punto è già un punto di differenza con il client.
  - La socket non è ancora pronta, ma deve essere resa capace di ricevere richieste e fare sincronizzazione, per cui avviene una ***listen***. Essa rende la socket sul server passiva, non sarà usata per connessione ma è in grado di ricevere richieste di connessione (che saranno gestibili su una porta diversa da quella di ascolto well known). Il server deve essere in grado di gestire poi anche una coda di richieste di connessione (e creazione di nuove socket per la comunicazione vera e propria). Il client deve fare handshake (per TCP) con il server, per cui il server deve essere predisposto ad essa perciò serve la listen. —>Il sistema Operativo fa l'handhake
  - Con la ***accept*** il server accetta connessioni su socket passive, quando accept ritorna vuol dire che é arrivata una richiesta e si può iniziare la comunicazione (con protocollo stabilito, che può essere HTTP). Accept crea una nuova socket dedicata non alla ricezione di nuove richieste, ma a comunicare col protocollo stabilito. La nuova socket viene creata quando ha luogo la accept (non prima) e si possono avere molte socket diverse -> il server deve continuare a eseguire le operazioni precedenti, indipendentemente dal fatto che ci sia già una comunicazione con un client (c'è ancora la prima socket che è aperta a ricevere nuove richieste). Il server deve riuscire a fare più cose contemporaneamente (più processi) oppure un solo processo che deve essere in grado di dedicarsi alle due cose.
  - Si comunica con ***read e write***, l'ordine dipende dal protocollo che stiamo usando.
  - Alla fine della comunicazione si ***chiude*** la socket.

- **Client**: inizia la comunicazione, anche se può fallire nel iniziare la comunicazione. Se la comunicazione deve avere successo sappiamo che il server deve essere già pronto prima.
    - Dichiara un oggetto di tipo ***Socket***
    - Con la ***connect*** dico alla socket alcune informazioni (a quale IP e a quale porta) che mi portano al servizio del server

      Avviene un punto di ***sincronizzazione*** con il server con l'operazione ***connect***, il client ha specificato la porta del server ma non ha specificato una sua sua porta (che però il server deve sapere). La porta che usa il client è irrilevante e non é sempre la stessa (dipende dal meccanismo di sincronizzazione con il server) → **NB**: la connect del client non può arrivare prima della accept del server perché sarebbe un errore
    - Si comunica con ***write e read***, l'ordine dipende dal protocollo che stiamo usando. Nel caso di HTTP avviene prima la write, ma in altri potrebbe essere diverso. La richiesta e la risposta HTTP sono nel ciclo read/write -> **NB**: prima di read e write (fino alla connect avvenuta) c'è TCP/IP mentre HTTP è nel ciclo di comunicazione
    - Alla fine della comunicazione si ***chiude*** la socket, in questo caso non c'è altro ma a volte potrebbe esserci una ultima comunicazione al server che dice che è finita la comunicazione. Non posso assumere che non ci sia nulla, è possibile ma non è detto.

NB: tutte le funzioni sono bloccanti, nel senso che se non arriva una risposta non si prosegue da un determinato step. Inoltre read, write e close sono le stesse funzioni che il SO offre per la gestione dei file, in particolare le socket vengono viste dai programmatori come file speciali. Tutti i canali di comunicazione nel modello UNIX sono stream di byte per cui posso usare le stesse funzioni read/write/close per tutti i canali.
NB: le funzioni in grassetto si chiamano system call

Le socket trasportano "stream" (= flussi) di bytes, quindi
- non c'è il concetto di "messaggio"
- la lettura/scrittura avviene per un numero arbitrario di byte

TCP (e gli altri protocolli di trasporto) non possono fare assunzioni sul contenuto, per cui TCP deve lavorare su flussi di byte (non c'è un concetto di messaggio), per cui leggere da una socket si usa: ***byteLetti read(socket, buffer, dimBuffer)***

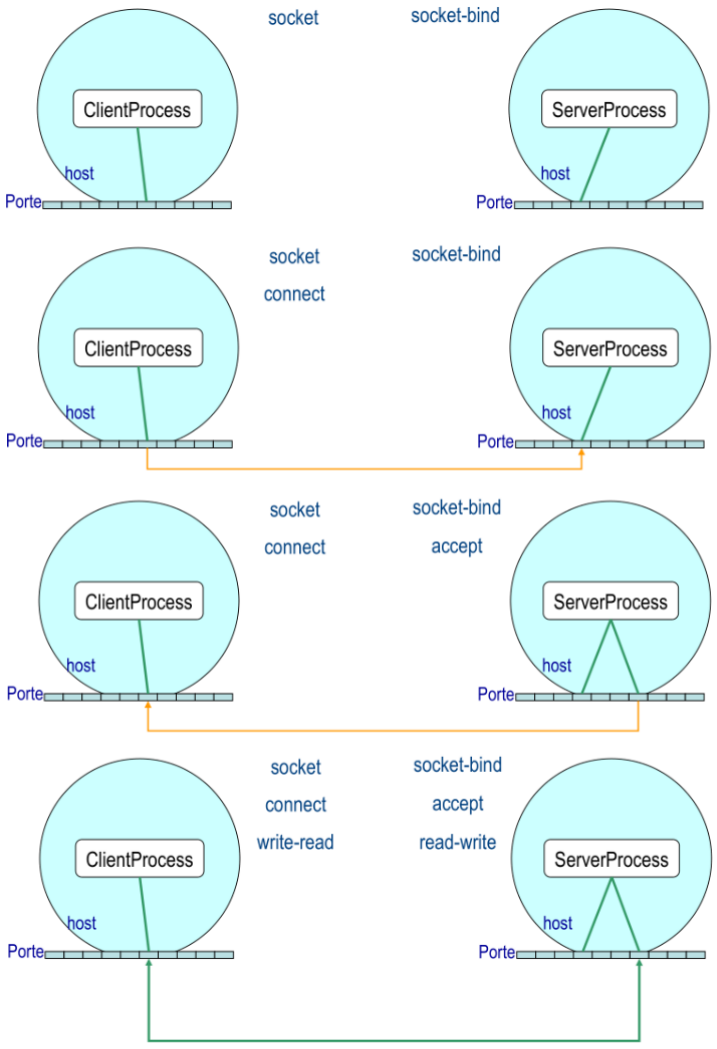
- byteLetti: byte effettivamente letti
- socket: canale da cui leggere
- buffer: spazio di memoria dove trasferire i byte letti
- dimBuffer: dimensione del buffer, ovvero il numero max di caratteri che si possono leggere

Quello che viene letto è inserito in un buffer e ne viene dichiarata la dimensione, ma i byteLetti non è detto che siano pari alla dimBuffer per cui chi legge deve essere pronto ad usare cicli di lettura, che terminano quando ho letto i byte che mi aspettavo (dimBuffer), inoltre nel ciclo devono essere incluse le condizioni di fallimento. Ciò che accade nel ciclo è affare di chi progetta il protocollo, sapendo che da TCP si hanno solo stream di byte. Lo stream di byte è potenzialmente infinito, ma la lettura va a segmenti di byte letti, che poi serviranno per ricostruire il messaggio.
Riferimento: insieme di indirizzo IP e la porta

PROCESSI E SOCKET

Il server crea una nuova socket collegata (binded) a una nuova porta per comunicare con il client, in questo modo la well-known port resta dedicata a ricevere richieste di connessione
Non si usa la stessa porta perché la prima rimane in ascolto. Read e Write sono gestite dall'applicazione, mentre le altre dal Sistema Operativo.

Il client e il server hanno molte porte e il server con la bind dice che il processo si connette a una certa porta specifica. Il client invoca socket e usa quella (non sa che porta è poiché lato client è irrilevante), poi con la connect si sincronizza con il server. Il formato della richiesta è definito dal protocollo di trasporto (ad es. TCP). Quando si avviene la accept non si usa la stessa porta perché l'altra deve restare aperta per richieste di connessione (si userà qui ancora TCP) mentre la seconda si usa per la comunicazione (con il protocollo HTTP o altro). In alcune situazioni c'è un processo che continua a gestire la prima socket, mentre la seconda è gestita da un altro processo. Ci sono poi le operazioni di read/write (uno legge e l'altro scrive), quando si legge c'è ovviamente un ciclo di lettura (per la write l'uso di un ciclo dipende se è necessario a seconda del caso).



Per trasporto non c'è solo TCP. TCP è orientato alla connessione (necessità setup prima di avere flusso affidabile di comunicazione), inoltre esso gestisce il flusso (mittente rallenta/accelera) e della congestione (mittente rallenta se la rete è congestionata), ma

non offre garanzie di banda e ritardi minimi perché effettivamente la comunicazione avviene a livello fisico, per cui è possibile che livello di Internet (rete di reti) possono esserci dei problemi anche perché Internet non può fornirmi garanzie (è best effort). Il controllo di flusso e congestione dipendono dal fatto che TCP si basa su IP che lavora a pacchetti e non da alcuna garanzia, per cui tcp per essere affidabile deve fare questi controlli. TCP deve essere affidabile, uno degli indicatori di performance è il ritardo (che dipende da Internet, dal provider ecc).

UDP esiste perché con tutti i controlli TCP è più lento, è un protocollo connectionless, non è affidabile e non offre alcun controllo, esso fa solo la frammentazione dei pacchetti
NB: usare un protocollo o un altro dipende dai requisiti richiesti dall'app

CICLO DI VITA DEI COMPONENTI

Ogni componente React di tipo classe ha 3 fasi di vita:

1. **Mounting**: il componente viene montato nel DOM ed è stato renderizzato una prima volta
   - **constructor()** → viene chiamato prima di qualsiasi altra cosa quando il componente viene inizializzato, nel costruttore viene definito lo stato iniziale del componente ed esso si occupa di controllare che un attributo se è importante, esista e abbia un valore di default (es. se ho this.state = {attributo: valore})
   - **getDerivedStateFromProps()** → viene chiamato appena prima che l'elemento venga renderizzato nel DOM ed è il metodo corretto per assegnare uno stato a partire dal valore delle props, poichè nel costruttore le props potrebbero non essere ancora state assegnate. Questo metodo serve a ispezionare le props e a trovare l'informazione necessaria per lo stato, esso non viene chiamato dal programmatore ma automaticamente da React dopo la chiamata al costruttore e non appena è certo che il valore delle props è stabile.
   - **render()** → è un metodo <u>obbligatorio</u> e il suo compito è esaminare this.props e this.state e restituire in output l'HTML generato. La funzione render() deve essere pura, ovvero:
     - non deve modificare lo stato del componente, altrimenti ciò porterebbe all'invocazione di un'altra render (una specie di loop)
     - deve restituire sempre lo stesso risultato ogni volta che viene invocata (ritorna un HTML diverso se cambia lo stato, ma se non cambiano stato e props deve sempre ritornare la stessa cosa)
     - non deve interagire direttamente con il browser, poichè essa ritorna l'HTML poi è il browser che deve capire cos'è

       NB: react non è dichiarativo perché qui nella render fa l'algoritmo che noi gli diciamo

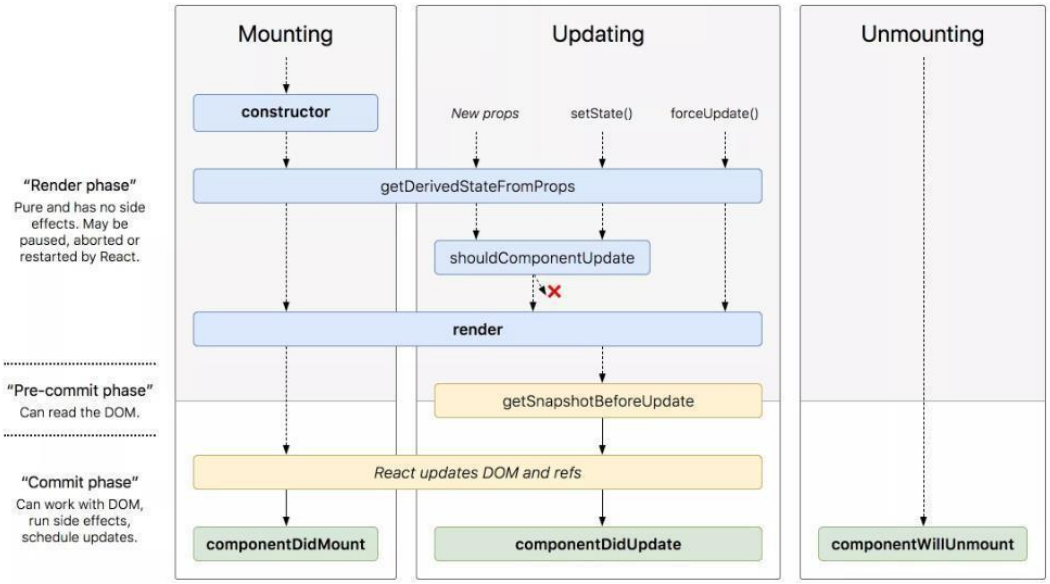
   - **componentDidMount()** → se definita viene chiamata dopo che il componente è stato renderizzato e serve per fare delle richieste al server per ottenere i dati necessari, a seguito di ciò il gestore dell'evento cambierà lo stato e si rifarà la render. Questo è il metodo corretto dove caricare dei dati da API esterne o dove eseguire qualsiasi logica che richieda che il componente esista già all'interno del DOM.

→ Qui il componente è già renderizzato, ma non ha la visualizzazione definitiva

2. **Updating**: avviene quando il componente viene aggiornato, ovvero quando le sue props o state cambiano.

- **getDerivedStateFromProps()** → serve per aggiornare lo stato, non viene chiamato solo in fase di visualizzazione
- **shouldComponentUpdate()** → è gestita internamente da React e ritorna un valore booleano (true/false) che specifica se il componente deve continuare ad aggiornarsi oppure no. In un normale programma React non dobbiamo modificarlo per evitare danni.
- **render()** → Renderizza il nuovo HTML;
- **getSnapshotBeforeUpdate()** → salva lo stato in attesa di fare update, se questo metodo è presente allora è necessario includere anche il metodo componentDidUpdate() altrimenti verrebbe sollevato un errore
- **componentDidUpdate()** → chiamato dopo che il componente è stato aggiornato nel DOM;

3. **Unmounting**: si verifica quando il componente viene smontato e quindi rimosso dal DOM (es. si chiude il tab o cambio l'url)
   - **componentWillUnmount()** → Chiamato quando il componente sta per essere rimosso.



GLI HOOKS

Con l'introduzione degli Hooks la gestione del ciclo di vita dei componenti si è semplificata enormemente. Non tutti i metodi precedentemente analizzati hanno un corrispettivo hook, alcuni dei meno frequenti non sono stati ancora implementati. Tuttavia, i metodi più importanti possiedono un hook che si occupa di ottenere lo stesso effetto nei componenti funzione.

- **useState**: permette di aggiungere uno stato locale al function component che si sta definendo e permette di gestire semplicemente lo stato. useState accetta un solo argomento che corrisponde al valore iniziale dello stato (42) e restituisce:
  - il valore dello stato corrente (age)
  - una funzione che permette di aggiornare lo stato (setAge)

  es.const [age, setAge] = useState(42);

  NB: È possibile dichiarare un numero arbitrario di useState, inoltre per convenzione il setter che viene ritornato da esso si chiama setNomeVarStato.

- **useEffect**: un componente funzione utilizza props e state per determinare l'output finale. Le operazioni eseguite dal componente che non hanno come target diretto il valore di output vengono definite **side-effects** (es. recupero di dati tramite chiamata ad una API, la manipolazione diretta del DOM o l'utilizzo di setTimeout).
  Non è possibile eseguire queste operazioni direttamente nel corpo della funzione.

  Il rendering del componente e i side-effects devono essere indipendenti, questo è reso possibile da **useEffect(callback, [dependencies]);** → **NB**: va importata!
  - **callback**: funzione di callback che contiene la logica del side-effect, essa viene eseguita quando il componente deve mostrare dei cambiamenti nell'interfaccia
  - **dependencies**: parametro opzionale che permette di limitare l'esecuzione della callback solo se cambiano alcuni specifici valori
    - Se non viene passato NULLA: la callback viene eseguita ad ogni nuovo rendering
    - Se viene passato un ARRAY VUOTO [ ], il side effect viene eseguito una volta sola dopo il rendering iniziale → equivale alla componentDidMount perché il componente viene montato una sola volta ovvero la prima
    - Se vengono passate PROPS O STATE [prop1, prop2, ..., state1, tate2], la callback viene eseguita quando uno dei valori elencati cambia

useEffect cleanup

A volte è necessario "resettare" alcuni side-effects useEffect() invoca la funzione di "clean up" che viene restituita dalla funzione di callback

```
useEffect(() => {
        // Side-effect...

        return function cleanup() {
                // Side-effect cleanup...
        };
}, [dependencies]);
```

1. Dopo il primo rendering useEffect() invoca la callback; la funzione di cleanup non viene invocata
2. Nei successivi rendering, useEffect() invoca la funzione di cleanup della precedente esecuzione della callback e successivamente esegue il side-effect
3. Quando il componente viene smontato, useEffect() chiama la funzione di cleanup invocata dall'ultima esecuzione del side-effect

NB: non è obbligatorio chiamarla cleanup ma la cosa importante è che viene ritornata dalla funzione di callback e si sa che va invocata quando il componente viene smantellato.

React e CSS

React non impone alcuna metodologia di integrazione dei CSS.
Alcune delle modalità possibili sono:

- **Inline CSS**: sono equivalenti agli stili inline usati nell'HTML tradizionale (sono sconsigliati come in HTML normale), ma vi sono delle differenze di sintassi poichè devono essere trattati come oggetti Js:
  - I nomi delle proprietà devono essere scritti in camelCase

    - Tutti i valori devono essere trattati come stringhe (es. “10px”, “blu”)
- **CSS e SCSS stylesheet**: volendo creare dei file esterni per gestire il CSS, il metodo più semplice è l’adozione di file .css (o eventualmente .scss se si vuole utilizzare SASS). Pratica comune è creare un singolo file .css per ciascun componente nel quale vi sono solo le regole di stile dedicate, esso deve essere poi importato nel codice .js del nostro componente.
  NB: Questo approccio però porta con sè un effetto collaterale, ovvero tutte le proprietà CSS definite sono accessibili globalmente e a volte questo può essere un comportamento voluto, ma nella maggior parte dei casi non è così
  NB: un componente dovrebbe essere auto contenuto e senza avere troppa dipendenza dal resto
- **CSS/SCSS modules**: permettono di definire il css in maniera tradizionale, ma queste regole di stile saranno limitate allo scope del componente in cui sono state importate (in fase di compilazione del progetto i nomi delle classi vengono sostituiti per essere univoci)
  Per usare questo metodo è necessario:
    - Creare un file chiamato nomefile.module.css
    - Importarlo nel file JS del componente in maniera esplicita assegnandogli un nome (questo permette di creare un oggetto JS a partire dal CSS)
    - Assegnare il nome della classe desiderata facendo riferimento all’oggetto importato
- **Styled Component (CSS in JS o JSS)**: permette di definire del CSS direttamente dentro i file JS e richiede l’installazione di un modulo aggiuntivo (styled-components).

Ad eccezione degli stili inline, in tutti gli altri casi è necessario assegnare delle classi ai propri elementi. Vanno dichiarati con className poiché class è una parola riservata per la creazione di classi in ES6.

Reactstrap è una libreria per utilizzare Bootstrap con React

Funzioni e dettagli

- **Routing**: spostamento tra diverse parti di un’applicazione quando un utente inserisce un URL o fa click su un elemento → React non ha sistema di routing integrato, si usa **ReactRouterDOM** (variante di ReactRouter) che serve per le app web.
- **Fetch**

Routing

Il routing è la capacità di spostarsi tra le diverse parti di un'applicazione quando un utente inserisce un URL o fa click su un elemento (link, pulsante, icona, immagine).

React non ha un sistema di routing integrato, la libreria più popolare è **React-Router**. Essa ha tre varianti:

1. **react-router**: la libreria principale
2. **react-router-dom**: una variante della libreria principale pensata per essere utilizzata per le applicazioni web
3. **react-router-native**: una variante della libreria principale utilizzata con React Native nello sviluppo di applicazioni Android e iOS

Per poter usare la libreria di routing è necessario installarla tramite il comando: *npm install --save react-router-dom*

Dopo aver creato le varie pagine è necessario associarle a specifici URL, tramite i componenti offerti da react-router-dom:

- **BrowserRouter**: è il componente principale che permette di gestire tutte le routes
- **Switch**: è un componente che si occupa di cercare in tutti i suoi figli di tipo quello il cui path corrisponde all'URL corrente; quando trova una corrispondenza la renderizza ed ignora gli altri casi. Nelle nuove versioni Switch ha cambiato nome e si chiama **Roots** (più intuitivo)
- **Route**: è il componente che permette la creazione delle routes e che associa un percorso ad un component
  es*. // route statiche*
  <Route exact path="path/to/be/match" component={Name} />
  // route dinamiche
  <Route exact path="path/to/be/match:param" component={Name} />

Tramite l'uso del componente **NavLink** di react-router-dom e della relativa props **to** (utilizzata per determinare l'href del link) possiamo visualizzare correttamente i link.

Cambiamenti in react-router-dom v6+ :

- Il componente Switch è stato rimpiazzato dal componente **Routes**
- Non esiste più la prop exact
- Il componente Route non ha più un componente "figlio", ma ha una prop element che deve contenere l'oggetto che va renderizzato qualora venga selezionata quella route
- L'uso di parametri presenti nella path associata a una Route richiede l'uso della funzione useParams

LE APPLICAZIONI WEB

Come posso identificare la controparte con cui intendo comunicare?

- **IP** dell' host su cui il processo è in esecuzione
- **Numero di porta**, permette all'host ricevente di identificare il processo locale destinatario del messaggio

Come può un processo comunicare attraverso un canale? Tramite le **API**, che definisce l'interfaccia tra app e canale di comunicazione (cioè lo strato di trasporto TCP). Le **socket** sono API Internet, ovvero due processi comunicano inviando/leggendo dati alla/dalla socket (con il modello client-server). Socket è il nome di canali di rete che possono essere usati in internet.
NB: è il trasporto che stabilisce il formato della richiesta

Il Web supporta l'interazione tra client (Browser) e server (HTTP Server o Web Server) via HTTP.

Perché realizzare applicazioni Web

Pro	Contro
Nessun software da installare sui propri device	Impossibilità di utilizzare l'applicazione se la rete è fuori servizio
Utilizzabile da qualsiasi punto della rete dalla quale sia accessibile il server	Transito in rete di dati sensibili / personali
Sempre aggiornata: poiché tutti gli utenti condividono la stessa installazione, questa è sempre l'ultima versione per tutti	Ridotta personalizzazione
Possibilità di integrazione: uso più applicazioni per ottenere un risultato	
Possibilità di collaborazione: web 2.0 (es: wikipedia, facebook)	

WEB 2.0: INNOVAZIONE NELLA COMUNICAZIONE

Nel web 1.0 gli editori pubblicano i contenuti, modello di comunicazione adottato: "pochi a molti" (portale)

Nel Web 2.0 si forma il concetto di comunità online tra utenti, distinte in base al modello di comunicazione adottato:

- "uno a uno" (e-mail e istant messaging)
- "uno a molti" (blog)
- "molti a molti" (wiki e blog considerando le risposte).

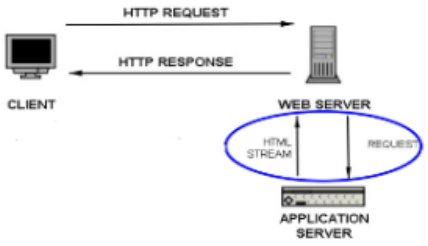
L'unica innovazione tecnologica di rilievo risiede nell'assemblaggio (Innovation in Assembly) di tecnologie e servizi preesistenti: i **mash-up** che hanno il vantaggio di essere semplici da realizzare e di non aver bisogno di conoscenze informatiche approfondite
La combinazione di tecnologie e standard esistenti (HTML5, CSS3, XML-JSON, DOM, JavaScript) per realizzare nuovi paradigmi (come AJAX) consentono lo sviluppo di **Rich Internet Applications** (RIA). → es. di mashup è dato dall'unione di Google Maps e Flickr che consente di visualizzare su una mappa le foto relative alla zona visualizzata.

DAL WEB ALLE APPLICAZIONI
Funzionamento

Web Server: il Web supporta l'interazione statica tra client e server via http.
Il Web server, quindi, non ha una logica di controllo che permetta di interpretare del codice e quindi è capace di gestire contenuti puramente statici: pagine HTML contenenti testi, foto, ecc. Ma non è in grado di garantire alcuna interattività con il visitatore tranne quella del saltare da una pagina ad un'altra tramite i link ipertestuali, cioè degli oggetti o del testo cliccando sui quali si accede ad un'altra pagina Web.

Web Server + Application Server = **Web Application**
Il client comunica con il Web Server, che interagisce a sua volta con un Application Server tramite la specifica **CGI** (Common Gateway Interface), che è la specifica di riferimento. Esistono implementazioni specifiche per i principali linguaggi di programmazione.

Questo modello applicativo è divenuto piuttosto popolare con la diffusione di Internet (Web 2.0). Infatti da la possibilità di realizzare applicazioni più complesse dove è prevista una interattività con l'utente. Per esempio, nei portali, siti di e-commerce, ecc., occorre uno strumento più evoluto del Web server: l'application server, che permette di rispondere alle richieste dell'utente tramite veri e propri programmi che vengono eseguiti sul server e che danno la possibilità di calcolare la risposta in tempo reale.

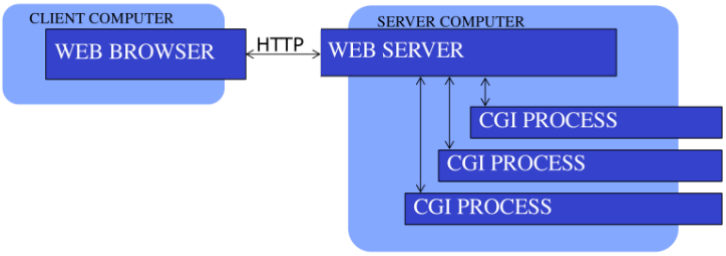


Il client accede all'applicazione connettendosi a funzionalità di elaborazione residenti su un **application server** utilizzando come terminali utente i web browser e appoggiandosi dunque ai consueti protocolli di rete.

- **URL** definisce un naming globale per le risorse in rete
  Le risorse possono essere dinamiche (cioè il risultato di esecuzioni di programmi)
- **HTTP** fornisce un modello tipo RPC (Remote Procedure Call) basato su socket.
  I metodi di HTTP sono le procedure che contengono i programmi
- **CGI** (Common Gateway Interface) permette al Web Server di attivare ed eseguire un programma e di scambiare dati

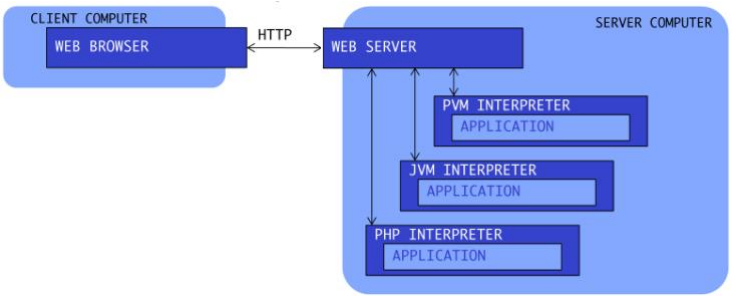
Il Web server per eseguire l'app può utilizzare 2 modelli:

1. **CGI** → l'applicazione implementa il protocollo CGI e la logica dell'applicazione.



I programmi hanno un'API dedicata (tutti la stessa)

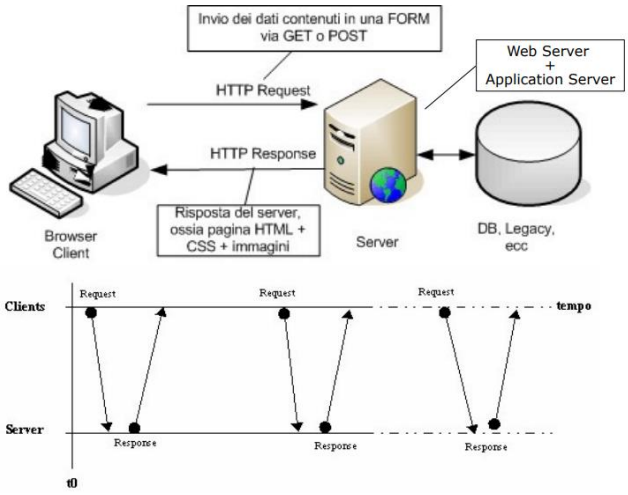
2. Interprete (utilizzo di **linguaggi interpretati**) → L'Application Server implementa il protocollo CGI per parlare con in Web Server e l'interprete per il linguaggio utilizzato.
   In questo caso serve programmare solo le logiche delle applicazioni:
   • Maggior semplicità e controllo per realizzare le applicazioni
   • Maggiore portabilità delle applicazioni (eseguibilità su macchine diverse)
   Es: Java/Servlet, PHP, Python, NodeJS



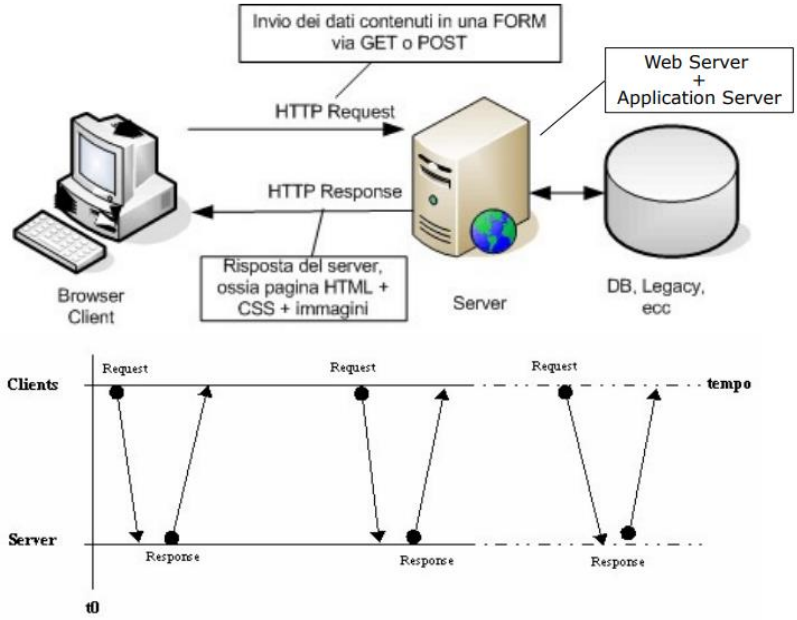
L'interprete include l'app, la esegue e parla con il Web Server

COM'È FATTA UNA APPLICAZIONE "CLASSICA"

- Comunicazione **sincrona** (finchè non arriva la risposta non faccio niente) tra client e server



- Il client effettua una HTTP request tramite GET o POST → Il form (o il link) può far partire la richiesta passando la chiamata alla risorsa



Client side HTML pages

Basata su:

- Link:
  -
    Get HTML Document

- Form:
  - GET

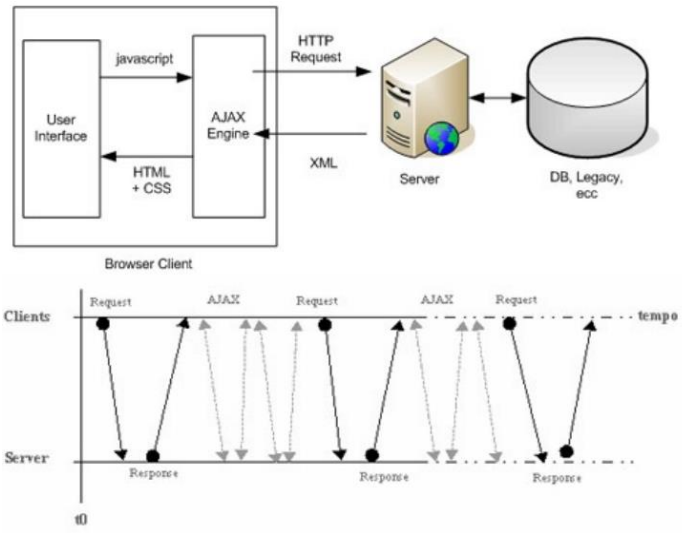
```
<form ACTION=http://localhost:8080/SlideServlet/GetHTTPServlet
METHOD="GET">
    <p>...</p>
    <input TYPE="submit" VALUE="Get HTML Document">
</form>
```

  - POST

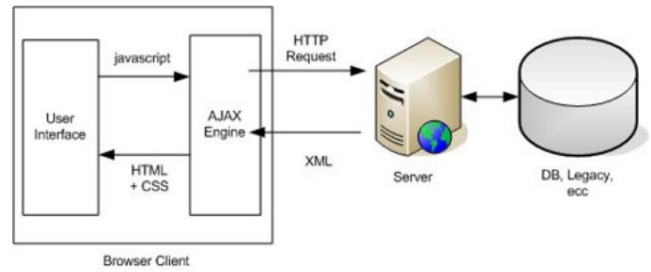
```
<form ACTION="http://localhost:8080/SlideServlet/PostHTTPServlet"
METHOD="POST">
    What is your favorite pet?<br><br>
    <input TYPE="radio" NAME="animal" VALUE="dog">Dog<br>
    …
    <br><br><input TYPE="submit" VALUE="Submit">
    <input TYPE="reset">
</form>
```

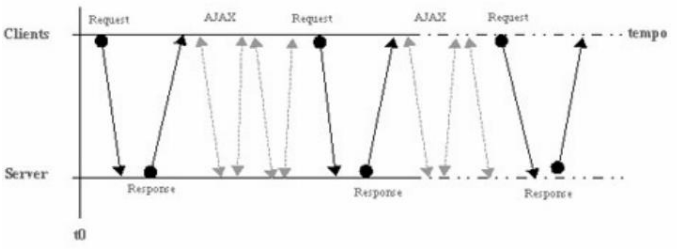
COM'È FATTA UN'APPLICAZIONE WEB AJAX (Asynchronous JavaScript and XML)

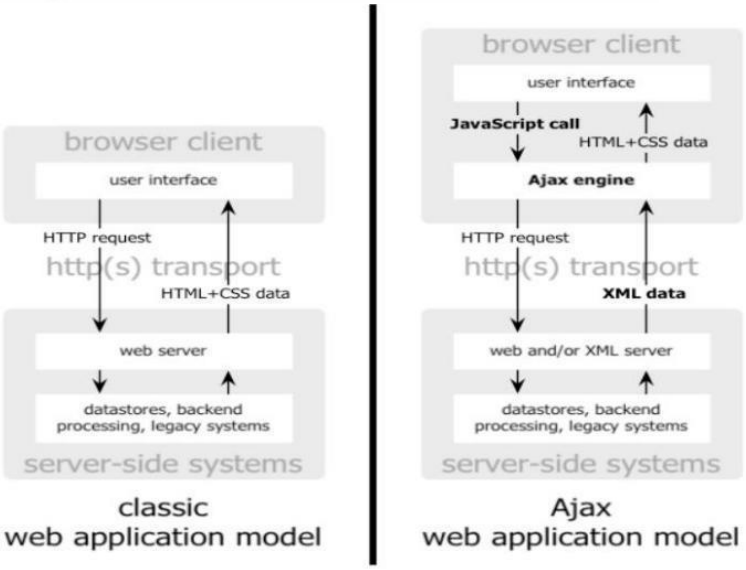
AJAX è una tecnica di sviluppo software per la realizzazione di applicazioni web interattive, consentendo l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente.



Interazione client server non è più diretta ma mediata da Ajax Engine. Non ho più una normale pagina HTML, ma ho anche del codice JS, ho sia la UI ma anche l'AJAX engine che esegue il mio javascript. La UI non chiama il server ma chiama il mio programma JS, è il programma che poi controllerà l'interazione con il server, in particolare gestisce la richiesta HTTP al server. Alla risposta abbiamo dei dati XML (nel caso di AJAX, o Json), non ho più la pagina HTML come risposta perché abbiamo l'AJAX engine che modifica la pagina HTML/CSS visualizzata. Tra client e server non ho più allineamento temporale, c'è una asincronia perché l'engine fa la richiesta quando vuole e quando vuole modifica poi la pagina.







Ho in esecuzione il browser e il mio programma, quando clicco bottone o altro invoco un metodo che può causare una richiesta verso il server. Il server è attivo solo quando deve servire una richiesta e dopo la risposta non è più attivo (non è detto che ci sia sempre la chiamata al server per ogni richiesta). C'è un disaccoppiamento tra i comportamenti che diventano asincroni tra loro. A volte ci possono essere punti di sincronizzazione. Nel caso precedente a ogni richiesta c'è subito una risposta per cui è sincrona

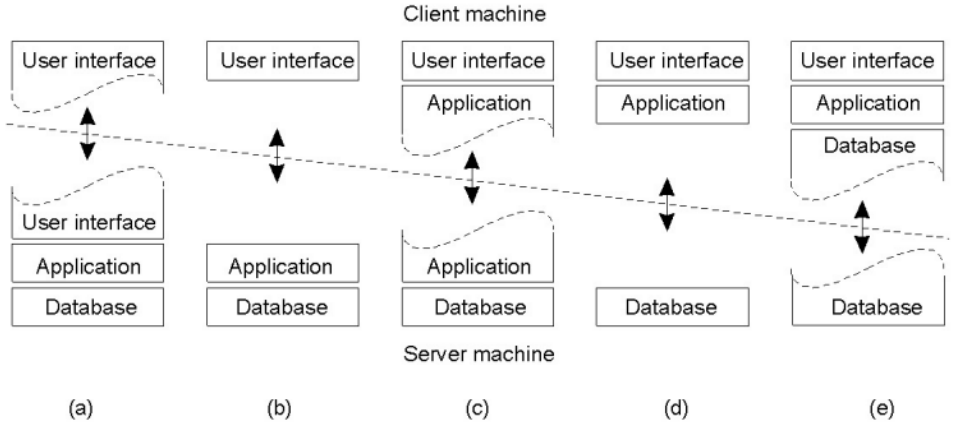
Confronto con applicazioni tradizionali

Il web server agisce in base a ciò che è stato trasmesso 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 e 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 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.

ARCHITETTURE MULTILIVELLO

Esistono diverse tipologie di architettura:



- (a): tutto server based
- (e): tutto client based
- Le web app sono tra (a) e (c)
    - La comunicazione tra client e server avviene tramite HTTP
    - HTTP non permette accesso diretto ai database

RIPARTIRE L'ESECUZIONE TRA CLIENT E SERVER (CASO C)

In questo caso ho una app come AJAX che spezza una parte sul server è una sul client, ciò ripartisce i carichi di lavoro.
Il bilanciamento dell'esecuzione tra client e server migliora le prestazioni e permette la personalizzazione delle applicazioni.
Permette di ottenere un miglioramento delle prestazioni:

- Alleggerisce il server da computazioni elementari (es. controllo ortografico, ordinamento dei dati)
- Riduce il traffico di rete e permette la personalizzazione:
- Scelta dell'organizzazione dei dati sulla pagina (es. scelta dei colori, dei dati visualizzati, dell'ordine delle finestre)
- Applicazioni single-page → caricamento incrementale dei dati (anziché caricare un'intera pagina è possibile caricare solo dati che servono ad aggiornare una parte di essa)

CLIENT SIDE (HTML PAGES + AJAX)

La tecnica AJAX utilizza una combinazione di:

- **JavaScript**:
    - è un programma inserito nella pagina web che ospita l'applicazione
    - definisce il funzionamento delle applicazioni Ajax
    - supporta la comunicazione con le applicazioni server
- **XML** (XHTML)
    - è utilizzato per trasferire i dati per costruire le pagine web identificando i campi per il successivo uso nel resto dell'applicazione
    - oggi prevale l'uso di **JSON** (JavaScript Simple Object Notation) per lo scambio dati
- **DOM**: utilizzato per manipolare tramite JS struttura della pagina HTML e risposte XML restituite dal server.

Problemi con Ajax apps

1. Il bottone “torna indietro” → quando torno indietro potrei andare in uno stato non corretto
2. Cambiamenti inaspettati di parti della pagina
3. È difficile capire a che stato si trova la pagina perché dipende dal comportamento dinamico di JS
4. Aumentare la dimensione del codice sul browser
5. Difficoltà nel debugging
6. Codice sorgente visibile lato client → si può copiare ed è visibile a eventuali hacker
7. L'interazione col server può essere pesante a causa delle richieste asincrone

```
function loadDoc() {
        //creazione del messaggio
        //predispone l'ambiente e crea la socket()
        var xhttp = new XMLHttpRequest();

        //gestione sincronia, onreadystatechenage viene triggerato quando readyState
        cambia
        //utilizza onreadystatechange per segnalare un nuovo stato del ciclo di
        //scrittura/lettura che il programma può conoscere attraverso la variabile
        readyState
        xhttp.onreadystatechange = function() {
        if (xhttp.readyState == 4 && xhttp.status == 200) {
                document.getElementById("demo").innerHTML = xhttp.responseText;
                }
        };

        //mandare la richiesta

        //apre la connessione con la connect() e crea la prima riga di richiesta (obbligatoria)
        in formato HTTP
        //si può popolare l'header del messaggio con funzioni dedicate → NB: header di
        //default vengono aggiunti automaticamente
        xhttp.open("GET", "ajax_text.txt", true);

        //chiude il messaggio, lo invia utilizzando write(), e avvia la lettura della risposta con
        //read()
        //NB: invia senza body perché è una GET
        xhttp.send();
}
```

La classe XMLHttpRequest() definisce funzioni che nascondono le chiamate alla API di Sistema, essa è supportata da tutti i browser moderni e l'oggetto è usato per lo scambio di di dati col server per aggiornare parti della pagina senza ricaricarla completamente.

readyState: restituisce lo stato nel quale si trova una richiesta XMLHttpRequest:

- **0: UNSENT** → il Client è stato creato, ma il metodo open() non è stato ancora invocato.
- **1: OPENED** → il metodo open() è stato invocato
- **2: HEADERS_RECEIVED** → Il metodo send() è stato invocato e sono già disponibili lo status della risposta HTTP ed il suo header

- **3: LOADING** → Sta avvenendo il download dei dati e responseText contiene dati parziali.
- **4: DONE** → l'operazione è completa e la risposta del server è pronta, il messaggio è stato ricevuto e letto. La risposta del server può essere:
    - **responseText** = restituisce la risposta come una stringa, si usa se la risposta del server non è XML
    - **responseXML** = la risposta dal server è XML e permette di fare il parsing come un oggetto XML

  NB: sono due proprietà dell'oggetto XMLHttpRequest

status:
- **200**: ricezione corretta
- **404**: page not found
- … altri codici di stato …

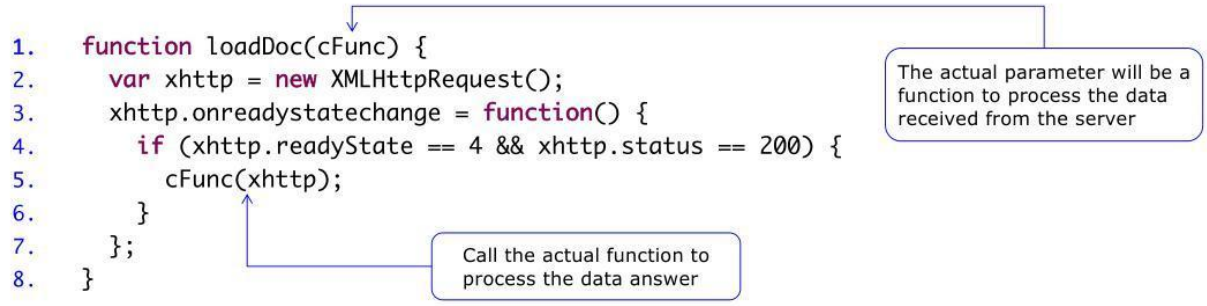
Mandare la richiesta consiste almeno di due operazioni:
- **open** che inizializza una richiesta e ha dei parametri:
    - **method** (GET/POST) = ottieni la risorsa
    - **url** (ajax_text.txt) = rappresenta l'URL della risorsa richiesta
    - **async**
        - **true** = asincrono → va specificata la funzione da eseguire quando la risposta del server è pronta nell'evento onreadystatechange, nel frattempo JS esegue altri script finché la risposta del server è pronta;
        - **false** = sincrono → JS non continuerà l'esecuzione finché la risposta del server non sarà pronta. Se il server è busy o lento, l'app si bloccherà. Non va dunque scritta la funzione onreadystatechange ma va semplicemente messo il codice dopo la send().
- **send** che manda il messaggio
  Se il metodo è POST, send() avrà un parametro string, quindi send(string), e sarà il body del messaggio

Se fosse stato inserito il gestore dell'evento (funzione) dopo la open e la send ciò non avrebbe avuto senso perché quando arriva il messaggio di risposta il browser non saprebbe cosa fare perché non ha ancora visto la funzione (che è successiva), la quale definisce l'associazione del gestore. Il sistema va configurato e predisposto prima di fare qualcosa con esso, quando viene inviata una richiesta al server noi vogliamo compiere delle azioni basate sulla risposta stessa.

Per fare la POST di dati come form HTML bisogna usare:
setRequestHeader(header, value) → aggiunge HTTP headers alla richiesta
header = nome dell'header
value = valore dell'header

Callback function
Una funzione di callback è una funzione passata come parametro a un'altra funzione.

Se si ha più di un'attività AJAX su un sito web, bisognerebbe creare una funzione standard per la creazione dell'oggetto XMLHttpRequest e chiamarla per ogni task AJAX.
La chiamata di funzione dovrebbe contenere l'URL e cosa fare nel onreadystatechange (che è probabilmente diverso per ogni chiamata):



Note sparse AJAX - Vizzari
Un primo meccanismo per app Ajax è la gestione del passaggio del tempo e ciò si fa con una funzione JS che è ***setInterval***. C'è una funzione più semplice che è ***setTimer*** che scatta una sola volta, mentre setInterval scatta più volte ma comunque ogni volta funziona come setTimer. SetInterval fa una append ogni 5000ms, fa una new date e costruisce una stringa (tutto con jQuery). Inoltre con Ajax è possibile interrogare servizi di terze parti.
In jQuery c'è la funzione ***$.getJSON()*** che permette di fare delle interrogazioni ad API.

JSON OBJECTS

JSON è un formato di scambio di dati (rappresentazione dei dati) ed è un sottoinsieme di JS, inoltre, può essere analizzato da un parser JS.

Esso è compatibile con altri linguaggi come C, C ++, C #, ColdFusion, Python ecc e supporta diversi tipi di dati e può rappresentare sia dati semplici che complessi.

Proprietà di JSON

- È un formato leggibile simultaneamente dall'uomo e dalla macchina.
- Supporta Unicode, consentendo a quasi tutte le informazioni in qualsiasi linguaggio di essere comunicate.
- Il formato auto-documentante descrive la struttura e i nomi dei campi, nonché valori specifici.
- La sintassi rigorosa e requisiti per il parsing permettono agli algoritmi necessari per il parsing di rimanere semplici, efficienti e consistenti
- La capacità di rappresentare le strutture dati informatiche più generali: records, liste e alberi.

Sintassi JSON

- Il dato minimo è una coppia nome / valore
- I dati sono separati da virgole
- Le parentesi graffe contengono gli oggetti
- Le parentesi quadre contengono gli array

I tipi di base sono:

- **Numero**: intero, reale o virgola mobile
- **Stringa**: Unicode tra virgolette e barre rovesciate
- **Booleano**
- **Array**: sequenza ordinata di valori separati da virgole racchiusi tra parentesi quadre
- **Oggetto**: raccolta di coppie chiave:valore separate da virgole e racchiuse tra parentesi graffe
- **Null**

Per passare da JSON a oggetti javascript devo togliere le “ ” dalle chiavi → si usa la funzione *JSON.parse(text)*, che ritorna un oggetto JS.

RESTFUL WEB SERVICES

Ad oggi i servizi sono fruiti da remoto.

SERVICE ORIENTED ARCHITECTURE – SOA

SOA è un modello architetturale per fornire servizi in internet.
È quindi basato sul concetto di servizio, che rappresenta quindi l‘elemento strutturale su cui le applicazioni vengono sviluppate.
SOA definisce una serie di proprietà che i servizi devono soddisfare per essere realmente riusabili e facilmente integrabili in ambiente eterogeneo:

- Fornire una **descrizione** e come fare per accedervi
- Fornire l'accesso tramite **protocolli** ben noti
- Supportare la **composizione** per fornire soluzioni complesse
- Indirizzati ai esigenze aziendali del client e ai requisiti di dominio

Architettura di base dei servizi Web

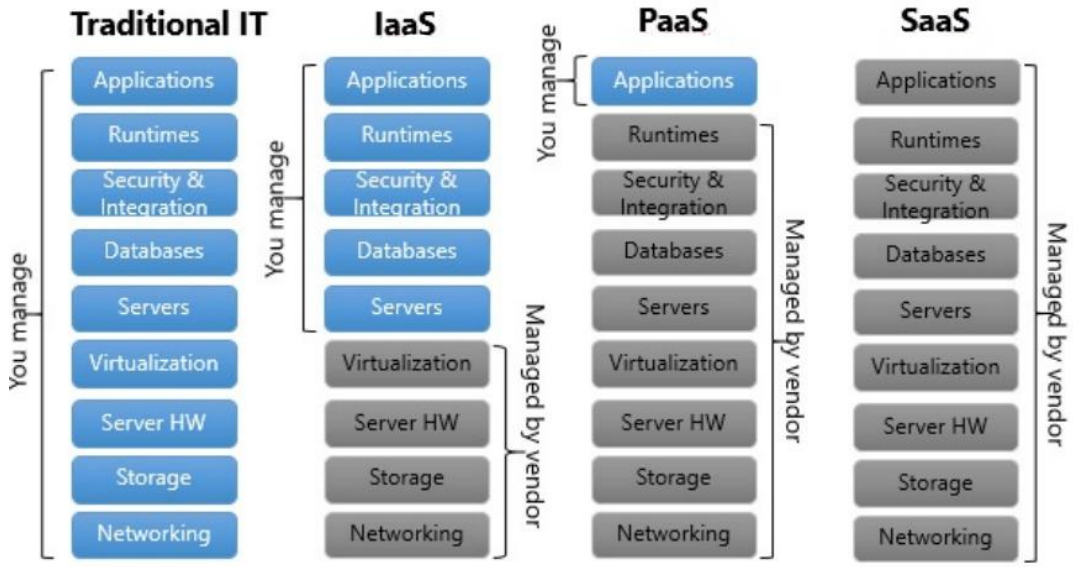
Un servizio deve quindi definire un‘interfaccia pubblicabile sulla rete, ricercabile e invocabile in maniera indipendente dal linguaggio e dalla piattaforma.

Per ottenere questi requisiti, le applicazioni SOA definiscono dei ruoli:

- **Service Provider** (fornitori di servizi)
- **Service Requestor** (richiedenti del servizio)
- **Service Brokers**: fornisce le informazioni riguardanti un servizio, definisce il formato per la richiesta di un servizio e della relativa risposta.

Le discovery agency fungono da intermediari.
I componenti della Web services sono: servizio e descrizione del servizio.
Le operazioni, invece sono: trova, pubblica e interagisci.

I servizi cloud possono avere diversi livello di messa in opera dei servizi stessi:



- **Traditional IT**: più sulla nostra macchina (es. i nostri esercizi).
- **IaaS**: infrastructure delivery, l'utente può gestire l'istanza della macchina virtuale e del virtual storage, mentre il provider gestisce l'infrastruttura (ovvero l'hardware)
- **PaaS**: platform delivery, c'è la possibilità di sviluppare e impiegare le app attraverso sistemi di sviluppo forniti dal fornitore del servizio cloud (es. Github)
- **SaaS**: software delivery tutto in cloud, la gestione è demandata a chi gestisce sia app che cloud per cui la gestione fisica del server è dislocata ed eventualmente sconosciuta. CI sono più utenti che possono usare la stessa app con una visione personale del software (es. Gmail)

La maggior parte dei servizi esistenti fornisce **Web APIs**, su cui si basano tutti i sw.

REST (REpresentational State Transfer)

REST è uno **stile architetturale** per sistemi distribuiti, ovvero:

- un set di constraints che implicano che il sistema abbia certe proprietà
- un insieme coordinato di vincoli architettonici che limita i ruoli/caratteristiche di elementi e le relazioni consentite tra essi all'interno di qualsiasi architettura conforme a quello stile
- un'astrazione, un modello di progettazione, un modo di discutere un'architettura senza preoccupazione per la sua implementazione

Obiettivo: minimizzare la latenza e la comunicazione attraverso la rete e allo stesso tempo massimizzare l'indipendenza e la scalabilità (= la possibilità di essere modificato) dei diversi componenti in un sistema.
REST abilita la memorizzazione nella cache e riutilizzo delle interazioni, sostituibilità dinamica dei componenti e elaborazione di azioni da parte di intermediari, soddisfacendo così le esigenze di un sistema basato su hypermedia e distribuito su scala d i Internet.

I costraint definiti da REST permettono di raggiungere:

- Semplicità/Familiarità
- Scalabilità
- Modificabilità
- Performance
- Visibilità (per monitorare)

- Portabilità
- Affidabilità

Principi e vincoli di Rest:

1. **Architettura client-server**: il server offre uno o più servizi richiesti da un client. Un client invoca il servizio messo a disposizione dal server inviando il corrispondente messaggio di richiesta e il servizio lato server respinge la richiesta o esegue l'attività richiesta prima di inviare un messaggio di risposta al client. La gestione delle eccezioni è delegata al client. Alla richiesta di una informazione la risorsa può trasferire anche la rappresentazione del proprio stato. Le rappresentazioni ritornate dal server possono linkare a uno stato aggiuntivo, quindi i client possono seguire il link per assumere un nuovo stato.
2. **Senza stato (stateless)**: la comunicazione tra client e server deve essere senza stato, ciò significa che ogni richiesta da parte di un client dovrebbe contenere tutte le informazioni necessarie affinchè il server sia in grado di comprendere la richiesta. Ovvero che ogni richiesta non è correlata ad una precedente, per cui i messaggi devono essere auto esplicativi. Avere un minor numero di stati di comunicazione permette un load balancing tra i server.
3. **Messaggi self-descriptive** → standard media types + meta-data e control-data
4. **Alcune risposte date dal server sono etichettate come cacheable**, in modo tale da essere riutilizzate in richieste successive. Le richieste del client vengono quindi passate attraverso un componente cache, che può riutilizzare le risposte precedenti per eliminare parzialmente o completamente alcune interazioni sulla rete. In questo modo si riduce il tempo di risposta e il carico del server.
5. **Interfaccia uniforme**: i componenti devono essere in grado di comunicare tramite un'interfaccia uniforme, che è esposta dalle risorse
6. **Il sistema deve essere stratificato** (layered system): una soluzione basata su REST può essere composta da più livelli architettonici e questi ultimi sono indipendenti l'uno dall'altro, ciò è necessario per garantire che sia cacheable.
7. **Identificazione di una risorsa**: una risorsa deve essere identificabile univocamente (tramite URI). Una risorsa è qualsiasi informazione che ha un nome, esse sono dotate di stato (che si modifica nel tempo) e di identificatori (una risorsa è qualsiasi cosa che sia importante da essere referenziata). L'identificazione viene effettuata utilizzando meccanismi standard, non sono necessari nomi aggiuntivi
8. **Manipolazione delle risorse attraverso rappresentazioni**: Una risorsa può essere rappresentata in molti modi diversi, ad esempio come HTML, XML, JSON o anche come file JPEG. Questa regola significa che i client interagiscono con le risorse tramite le loro rappresentazioni.
9. **Hypermedia è il modo per controllare il comportamento (stato) dell'app**.
10. **Lo stato dell'applicazione è guidato dalla manipolazione delle risorse**

REST segue i principi architetturali del web, ovvero URI e HTTP

Con REST il software è meno specializzato perché le tecnologie sottostanti sono semplici, inoltre esso permette di usare correttamente e completamente HTTP, fornendo un meccanismo a strati e leggero per i dati e l'integrazione dei servizi → REST con HTTP fornisce una piattaforma distribuita e basata su applicazioni basate su hypermedia

sistema RESTful: un sistema che rispetta tutti i vincoli precedentemente definiti, anche se in generale i sistemi possono aggiungere costraints o rilassarne alcuni e quindi sono più o meno RESTful. Il termine è in generale usato per fare riferimento a servizi Web che implementano l'architettura REST.

Http è perfetto per implementare un'architettura REST

L'insieme dei principi di progettazione, conosciuti come REST, sostiene il protocollo HTTP.

- HTTP si serve di un'architettura client-server ed è un protocollo a livello applicativo non a livello di trasporto
- Una richiesta a una risorsa avviene attraverso l'impiego di URI (**addressability**)
- I client fanno richieste ai server attraverso operazioni ben definite (**uniform interface**) → get, post, put, delete ecc
- In HTTP request/response vengono inserite le header lines e le eventuali risorse (**self-descriptive**)

Ripasso su formato di messaggi HTTP

HTTP utilizzato per implementare un'architettura REST:

- Trovare tutti i nomi: tutto in un sistema RESTful è una risorsa (un nome), il cui URI dovrebbe essere descrittivo, deve essere opaco (ovvero che client non umani non dovrebbero poterne dedurre il significato) e dovrebbe essere interessante (non cambiano). È necessario:
  - usare variabili path per codificare una gerarchia (es. /spese/123)
  - utilizzare altra punteggiatura per evitare di suggerire la gerarchia (es. /spese/Q107;Q307)
  - utilizzare variabili di query per passare input in un algoritmo (es. /search?approved=false) → bisogna evitare query complicate e semplificare (es. /spese/20070101-20071231 al posto di start e end).
  - Le cache tendono a (erroneamente) ignorare gli URI con variabili di query
  - Lo spazio dell'URI è infinito (ma la lunghezza dell'URI non lo è ~ 4K)
  - Non far trapelare informazioni sulla piattaforma (es. /spese.php/123 andrebbe tolto il .php)
- Definire i formati: HTTP e REST non impongono una singola rappresentazione per i dati, quindi una risorsa può avere più rappresentazioni (es. XML, JSON, ecc), fatte con i tipi MIME.
  Gli schema languages non sono richiesti (se possibile)
- Scegliere le operazioni: HTTP ha un'interfaccia utente vincolata (insieme di verbi/operazioni/metodi):
  - **OPTIONS**: rappresenta una richiesta di informazioni sulle opzioni di comunicazione disponibili sulla catena di request/response identificata dalla RequestURI
  - **GET**: significa recuperare qualsiasi informazione (sotto forma di un'entità) identificata dal RequestURI → tutte le nostre risorse sosterranno GET
    Conditional GET: è possibile richiedere la rappresentazione di una risorsa se non è cambiata rispetto all'ultima volta in cui è stata richiesta → riduce larghezza di banda, elaborazione del client e (possibilmente) elaborazione del server
  - **HEAD**: identico a GET tranne per il fatto che il server NON DEVE restituire un corpo del messaggio nella risposta, ma solo l'intestazione
  - **POST**: viene richiesto al server di creare e aggiornare una risorsa corrispondente a quella indicata nella RequestURI che viene assegnato dal server stesso
  - **PUT**: viene richiesto al server di creare e aggiornare una risorsa corrispondente a quella indicata nella RequestURI che viene assegnato dal client

- **DELETE**: richiede che il server di origine elimini la risorsa identificata dalla RequestURI
- **TRACE**: dà ai client un modo per scoprire il percorso di rete che i messaggi seguono per arrivare fino ai server.

Operazione	***Cache***	***Safe***	***Idempotent***
Definizione	È possibile memorizzare risorse nella cache che permette di ridurre la latenza ed il traffico di rete.	Operazioni solo in lettura (non modifica né lo stato del sistema né la risorsa)	Se effettuo la stessa operazione più volte ottengo lo stesso risultato
Options			X
Get	X	X	
Head	X	X	
Post			
Put			X
Delete			X
Trace			X

Evidenzia codici di stato eccezionali → **Ripasso su Codici di stato**

La differenza tra POST e PUT sta nell'interpretazione dell'URI da parte di un server. Con il metodo POST, l'URI identifica un oggetto sul server che può processare i dati inclusi nel corpo dell'entità. Con PUT, invece, l'URI indica un oggetto in cui il server dovrebbe immettere i dati. Mentre un URI POST indica generalmente un programma o uno script, l'URI PUT è di solito il percorso o il nome di un file.

POST significa "crea nuovo" come in "Ecco l'input per creare un utente, crealo per me". PUT significa "inserisci, sostituisci se già esiste" come in "Ecco i dati per l'utente 5".

PUT vs POST

- Quando si crea una nuova risorsa
  - Usare POST se il server sceglie l'URI (id)
  - Usare PUT se il client sceglie l'URI (id)
- Processa questo
  - POST fa qualcosa e ritorna qualcosa
  - non usare casualmente

Content Negotiation: dato che una risorsa può avere più rappresentazioni, il client può chiederne una specifica tramite request headers, es. Accept: text/xml,application/xml ecc, Accept-Language: en-us,en;q=0.5 dove q è una preferenza relativa (tra 0 e 1). Nella pratica la content negotiation non funziona perchè molti client non espongono un mezzo per settare gli accept headers oppure non li supportano → best practice: usare accept headers e esporre il content type nell'URI (es. http://expenses.example.com/123.xml)

Caching

Esistono 2 tipi di cache: browser (user agent) e proxy
I siti più grandi dovrebbero utilizzare un server proxy di memorizzazione nella cache. L'applicazione deve sapere come controllare come il suo contenuto viene memorizzato nella cache, in caso contrario possono essere utilizzate le impostazioni predefinite, inoltre non vanno memorizzate nella cache informazioni sensibili. Per gestire la cache sono presenti le seguenti operazioni:

Header	*Property*
Expires	data HTTP tenere fino alla scadenza
Cache-Control: max-age	Mantiene per questa quantità di tempo (secondi)
Cache-Control: s-maxage	Come sopra, ma solo per proxy
Cache-Control: public	Cacheable anche se è richiesto authN (autenticazione per accedere al servizio)
Cache-Control: no-cache	Cacheable, ma la cache deve convalidarne l'aggiornamento
Cache-Control: no-store	Non memorizzare nella cache
Cache-Control: must-revalidate	Non permettere le rappresentazioni di stato
Cache-Control: proxy-revalidate	Come sopra, ma solo proxy

NB: è possibile fare richieste di tipo asincrono

REST vs RESTful

HTTP	Method	CRUD	Desc.
POST	CREATE	Create	-
GET	RETRIEVE	Retrieve	Safe,Idempotent,Cacheable
PUT	UPDATE	Update	Idempotent
DELETE	DELETE	Delete	Idempotent

Molte app REST si basano su **CRUD** (create,read,update,delete) e il controllo degli hypermedia manca (la combinazione di metodi e URI che dice al client quali risorse sono disponibili e come gestirle) → **HATEOAS**: Hypermedia as the engine of application state

- application state: lo stato corrente è mantenuto sul client, mentre i server mantengono lo stato della risorsa
- gli hypermedia sono link e form, seguire un link significa avere una transizione di stato (il server fornisce una nuova rappresentazione e il client assume quello stato)

- il server guida il client a un nuovo stato fornendo i link all'interno delle rappresentazioni degli ipertesti (tutte le rappresentazioni che possono contenere link dovrebbero farlo)
- i client dovrebbero solo seguire i link forniti dal server o costruiti via form → i link dovrebbero anche essere classificati in base al loro significato (semantica) così un client smart potrebbe sapere a cosa punta un determinato link, ciò aiuta anche a rendere l'accoppiamento tra client e server meno stretto
- Gli hypermedia permettono:
    - scoprire nuove capacità
    - ordinare le interazioni
    - indipendenza di localizzazione e confini di sicurezza
- il consumatore richiede una singola URI per navigare tra i link
  NB: la rappresentazione degli hypermedia è l'interfaccia

Come capire se un'app è RESTful:

- cambio stato nella GET (indizio: ho un verbo nell'URI) → è RESTful ma può essere sbagliato
- espongo solo alcune URL e metto verbi nel message body o nell'URI → overloading post è RESTful ma è più RESTful se si mantengono le risorse disponibili
- Possiamo rilassare i constraints ma bisogna farlo in modo consapevole, se servono delle sessioni è meglio usare i cookie. È meglio avere risorse collegate tra loro.