

Appunti Sistemi Distribuiti

Domande e Risposte

Capitolo 5 - Web App, Servizi, Servlet

1) Cosa si intende per Applicazione Web?

Si definisce **applicazione web** indica tutte le **applicazioni** accessibili/fruibili via web per mezzo di un **network** che forniscono determinati **servizi** ad un **client Web**.

2) Quali sono le caratteristiche del protocollo HTTP?

Le caratteristiche del **protocollo HTTP** sono:

- formato a caratteri (lento): occorre tradurre e ritradurre i dati;
- **Header** (per **metadati**) e **Body** (corpo del messaggio);
- utilizzo del linguaggio **HTML** per input e output, ossia uso di **FORM** per l'acquisizione dati (invio dati al server), uso di documenti **HTML** in risposta (dal server verso il client) e possibilità di avere pagine dinamiche (**JSON**);
- utilizzo di payload di tipo **MIME** (**Multimedia Internet Mail Extensions**);
- conversazioni (**interazioni client - server**) prive di stato (memoria) e ogni richiesta è un messaggio autonomo, indipendente dagli altri.

3) Cookie HTTP: definizione

Un **cookie HTTP** (**web cookie, browser cookie**) è un piccolo blocco di dati che un server invia al browser web di un utente. Viene utilizzato per stabilire se due richieste provengono dallo stesso browser, ad esempio per mantenere un utente connesso alla **pagina Web**.

4) Caratteristiche della Tecnologia Server - Side

Le caratteristiche della **Tecnologia Server - Side** sono:

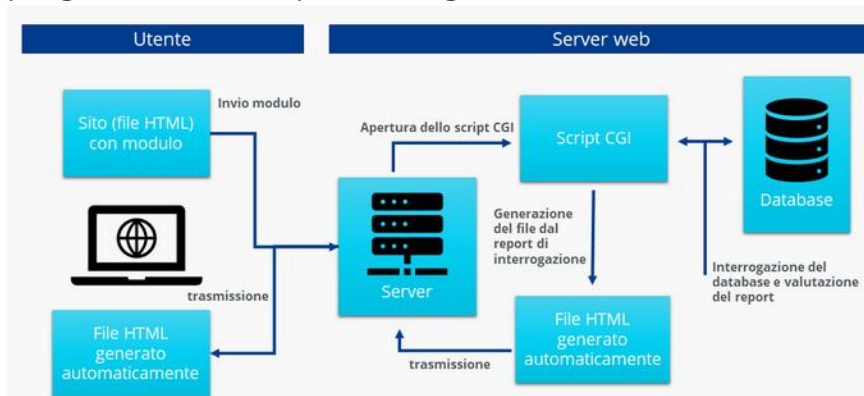
- la computazione avviene (in parte) lato server e può avvenire tramite l'**esecuzione programmi** (**compilati**) o **script** (**interpretati**);
- l'**application server** fornisce un ambiente per la gestione automatizzata del ciclo di vita di tali programmi;
- nel caso di programmi compilati il web server si limita ad invocare, su richiesta del client, un eseguibile (esempio **C++**);
- nel caso di esecuzione di **script**, il **web server** ha al suo interno un motore (engine) in grado di interpretare il linguaggio di scripting usato (**Java, Python, Perl e Node JS**).

5) Architettura di un'applicazione web compilata (CGI: Common Gateway Interface)

Il **Common Gateway Interface (CGI)** è un protocollo che permette al server di:

- attivare un programma (crea un processo);
- passargli le richieste e i parametri provenienti dal client;
- recuperare la risposta.

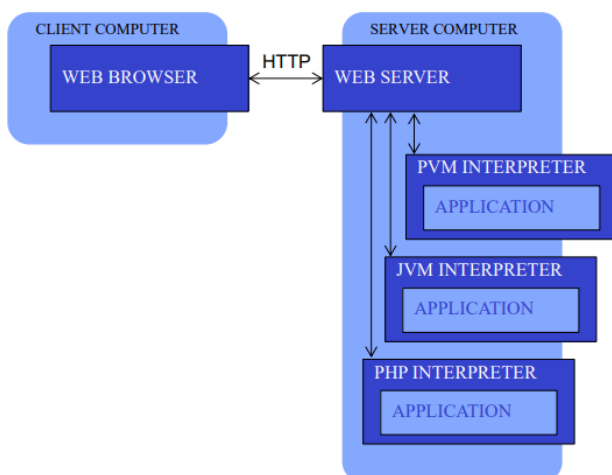
Ogni applicazione **CGI** deve quindi implementare l'interprete del protocollo e quando l'**URL** richiesto corrisponde a un'**applicazione CGI**, il **server** esegue il programma in tempo reale, generando dinamicamente la risposta per l'utente.



6) Architettura di un'applicazione web interpretata: vantaggi

I vantaggi di tale **architettura** sono:

- serve programmare solo le logiche delle applicazioni;
- modello delle applicazioni conformi al modello del linguaggio utilizzato;
- semplicità, portabilità, manutenibilità.



7) Client - Side: HTML (Richieste basate su link)

Un link in un **documento HTML** può essere usato per puntare ad una risorsa remota:

```
<html>
  <head>
    <title>Pagina Web Example</title>
  </head>
  <body>
    <p>Click the link to ask the servlet to send back an HTML document</p>
    <a href="http://localhost:8080/SlideServlet/GetHTTPServlet">GET HTML Document</a>
  </body>
</html>
```



Click the link to ask the servlet to send back an HTML document

[GET HTML Document](http://localhost:8080/SlideServlet/GetHTTPServlet)

Il **browser** invia richieste del tipo:

GET /SlideServlet/GetHTTPServlet HTTP/1.1

8) Client - Side: HTML (Richieste per mezzo di form)

Anche il parametro action di un **Form** può essere usato per puntare ad una risorsa remota (**applicazione**)

```
<html>
  <head>
    <title>Pagina Web Example</title>
  </head>
  <body>
    <p>Click the link to ask the servlet to send back an HTML document</p>
    <form action="http://localhost:8080/SlideServlet/GetHTTPServlet" method="GET">
      <p>Click the button to have the servlet send an HTML document</p>
      <input type="submit" value="GET HTML Document">
    </form>
  </body>
</html>
```



Click the link to ask the servlet to send back an HTML document

Click the button to have the servlet send an HTML document

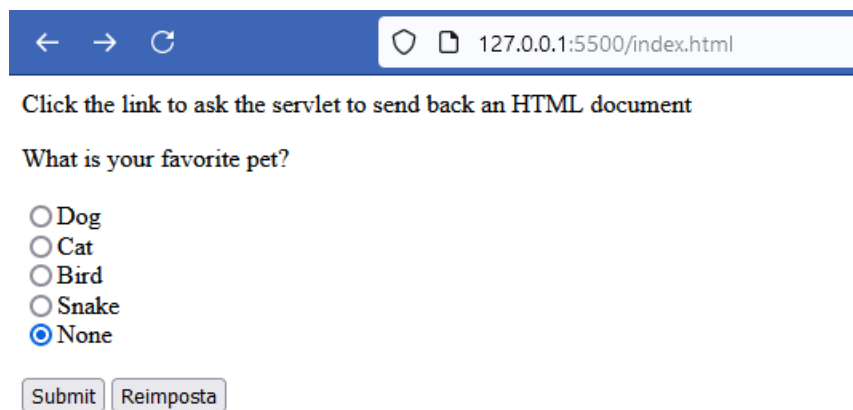
GET HTML Document

Viene inviata una richiesta del tipo

GET /SlideServlet/GetHTTPServlet HTTP/1.1

Un **Form** può essere utilizzato per mandare dei dati al **server**.

```
<html>
  <head>
    <title>Pagina Web Example</title>
  </head>
  <body>
    <p>Click the link to ask the servlet to send back an HTML document</p>
    <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>
      <input type="radio" name="animal" value="cat">Cat<br>
      <input type="radio" name="animal" value="bird">Bird<br>
      <input type="radio" name="animal" value="snake">Snake<br>
      <input type="radio" name="animal" value="none" checked="">None<br>
      <br>
      <input type="submit" value="Submit"> <input type="reset">
    </form>
  </body>
</html>
```



A screenshot of a web browser window. The address bar shows '127.0.0.1:5500/index.html'. Below the address bar, there is a text prompt 'Click the link to ask the servlet to send back an HTML document'. Below that, a question 'What is your favorite pet?' is followed by five radio button options: 'Dog', 'Cat', 'Bird', 'Snake', and 'None'. The 'None' option is selected. At the bottom, there are two buttons: 'Submit' and 'Reimposta'.

Il **browser** invia una richiesta del tipo:

```
POST /SlideServlet/PostHTTPServlet HTTP/1.1
Content-Length: 11
Content-Type: application/x-www-form-urlencoded
```

```
animal=none
```

Payload

9) Server - Side : Java Servlet - definizione e caratteristiche

Una **servlet** è un componente gestito in modo automatico da un **container** (detto anche **engine**). Il **container** controlla le **servlet** (le attiva/disattiva) in base alle richieste dei client e questo è possibile in maniera automatica perché le **servlet** implementano una interfaccia nota al **server**. Le **servlet** sono oggetti **Java** residenti in memoria dove:

- il codice dei metodi delle **servlet** viene eseguito da **thread** creati e gestiti dall'**application server**;
- possono interagire con altre **servlet**.

I **vantaggi** di tale componente sono la semplicità e standardizzazione, mentre gli **svantaggi** sono la rigidità del modello.

10) Come può essere implementata l'interfaccia Servlet?

Ogni **servlet** implementa l'interfaccia `jakarta.servlet.Servlet`, con 5 metodi:

- `void init(ServletConfig config)`: inizializza la **servlet**, viene invocato dopo la creazione della stessa;
- `void destroy()`: chiamata quando la **servlet** termina (es: per chiudere un file o una connessione con un database);
- `void service(ServletRequest request, ServletResponse response)`: invocato per gestire le richieste dei client;
- `ServletConfig getServletConfig()`: restituisce i parametri di inizializzazione e il

ServletContext che fornisce accesso all'ambiente;

→ `String getServletInfo()`: restituisce informazioni tipo autore e versione.

11) Quali sono classi astratte implementano l'interfaccia Servlet?

Le due **classi astratte** sono `jakarta.servlet.GenericServlet`, che definisce metodi indipendenti dal protocollo, e `jakarta.servlet.http.HttpServlet`, che definisce i metodi per l'uso in ambiente **web**.

12) Che ruolo ha la classe HttpServlet?

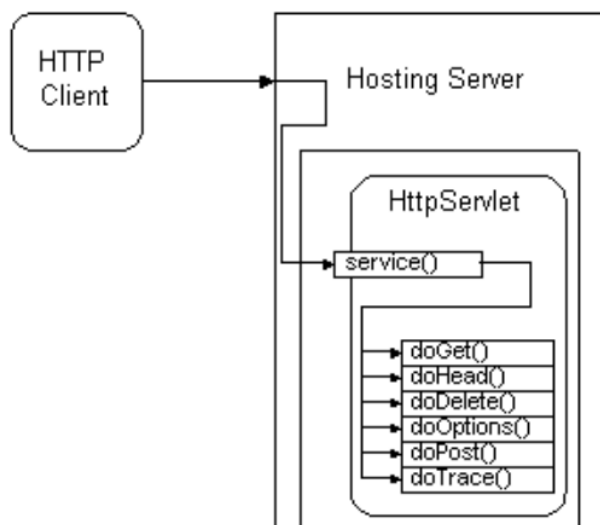
La classe **HttpServlet** implementa `service()` in modo da invocare i metodi per servire le richieste dal web.

Si hanno quindi:

→ **Metodi doX**: dove X è un metodo **HTTP** (`doGet`, `doPost`, ...) e doX è dedicato alle richieste di tipo X.

→ **Parametri**: (`HttpServletRequest`, `HttpServletResponse`)

→ **Eccezioni**: (`ServletException`, `IOException`)



13) Quali sono i metodi principali per gestire le richieste?

I metodi principali per manipolare le richieste:

→ `String getParameter(String name)`: Restituisce il valore del query parameter dato il nome (valore singolo)

→ `Enumeration getParameterNames()` - Restituisce l'elenco dei nomi degli argomenti

→ `String[] getParametersValues(String name)` - Restituisce i valori dell'argomento name (valore multiplo)

14) Quali sono i metodi principali per gestire le risposte?

I metodi principali per manipolare le risposte:

→ `void setContentType(String type)` - Specifica il tipo MIME della risposta per dire al browser come visualizzare la risposta

→ `ServletOutputStream getOutputStream()` - Restituisce lo stream di byte per scrivere la risposta

→ `PrintWriter getWriter()` - Restituisce lo stream di caratteri per scrivere la risposta

Altri metodi:

→ `Cookie[] getCookies()` - Restituisce i cookies del server sul client

→ `void addCookie(Cookie cookie)` - Aggiunge un cookie nell'intestazione (header) della risposta

→ `HttpSession getSession(boolean create)` - Una `HttpSession` identifica il client. Viene creata se `create = true`

15) Ciclo di vita di una servlet

Il ciclo di vita di una **servlet** è il seguente:

Una **servlet** viene creata dal **container/engine** quando:

→ quando viene effettuata la prima chiamata;

→ il **servlet** viene condivisa da tutti **client**;

→ ogni richiesta genera un **Thread** che esegue la `doXXX` appropriata.

Il container/engine invoca il metodo `init()` per inizializzazioni specifiche.

Una **servlet** viene distrutta dall'engine all'occorrenza di uno dei due eventi:

→ quando non ci sono thread in esecuzione su quella servlet;

→ quando è scaduto un timeout predefinito.

Viene invocato il metodo `destroy()` per terminare correttamente la **servlet**.

16) Terminazione della servlet

Per quanto riguarda la terminazione della **servlet** il container e le richieste dei client devono sincronizzarsi durante la terminazione. Alla scadenza del timeout potrebbe essere ancora dei **thread** in esecuzione in `service()`. Bisogna tener traccia dei **thread** in **esecuzione**, progettare il metodo `destroy()` in modo da notificare lo shutdown e attendere il completamento del metodo `service()` e progettare i metodi lunghi in modo che verifichino periodicamente se è in corso uno **shutdown** e comportarsi di conseguenza.

17) Java Server Pages (JSP): descrizione e caratteristiche

La **Java Server Pages (JSP)** è una tecnologia per la creazione di applicazioni web. Specifica l'interazione tra un contenitore/server ed un insieme di "pagine" che presentano informazioni all'utente (viste). Le caratteristiche di **JSP** sono:

- le pagine sono costituite da tag tradizionali (**HTML**, **XML**, **WML**, ...) e da tag applicativi che controllano la generazione del contenuto (generazione **server-side**);
- **JSP**, rispetto alle **servlet**, facilita la separazione tra **logica applicativa** e **presentazione**;
- analogo alla tecnologia **Microsoft Active Server Page (ASP)**;
- **Java Server Pages (JSP)** separano la parte dinamica delle pagine dal template **HTML** statico;
- la pagina viene convertita automaticamente in una **servlet Java** la prima volta che viene richiesta.

18) Elementi di una JSP

Java Server Page è costituita dalle seguenti parti:

- **Template text**: le parti statiche della **pagina HTML**.
- **Commenti**: `<!-- questo è un commento -->`
- **Direttive**: non influenzano la gestione di una singola richiesta **HTTP** ma influenzano le proprietà generali della **JSP** e come questa deve essere tradotta in una **servlet**.

In **JSP** si hanno le seguenti **direttive**:

a) **page**: rappresentata da una lista di attributi/valore e valgono pagina per pagina

```
<%@ page import="java.util.*" buffer="16k" %>
<%@ page import="java.math.*, java.util.*" %>
<%@ page session="false" %>
```

b) **include**: include in compilazione pagine **HTML** o **JSP**

```
<%@ include file="copyright.html" %>
```

c) **taglib**: dichiara tag definiti dall'utente implementando opportune classi

```
<%@ taglib uri="TableTagLibrary" prefix="table" %>
<table:loop> ... </table:loop>
```

→ **Azioni**: `<jsp:XXX attributes> body </jsp:XXX>`

In **JSP** si hanno le seguenti **azioni**:

a) **forward**: determina l'invio della richiesta corrente, eventualmente aggiornata con ulteriori parametri, alla JSP indicata

```
<jsp:forward page="login.jsp" >
  <jsp:param name="username" value="user" />
  <jsp:param name="password" value="pass" />
</jsp:forward>
```

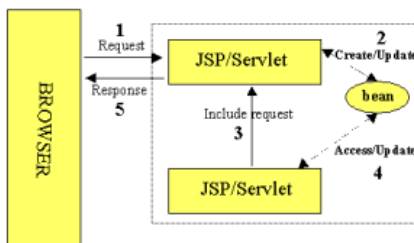
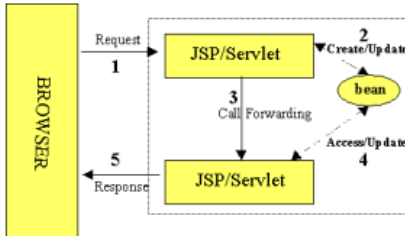
b) **include**: invia dinamicamente la richiesta ad una data URL e ne include il

risultato

```
<jsp:include page="shoppingCart.jsp" />
```

c) useBean: localizza ed istanzia (se necessario) un javaBean nel contesto specificato e il contesto può essere la pagina, la richiesta, la sessione e/o l'applicazione.

```
<jsp:useBean id="cart" scope="session" class="ShoppingCart" />
```



→ **Declaration:** `<%! declaration [declaration] ... %>`

Vi sono variabili o metodi usati nella pagina

```
<%! int[] v= new int[10]; %>
```

→ **Expression:** `<%= expression %>`

Una **espressione** nel linguaggio di scripting (**Java**) che viene valutata al momento della richiesta e sostituita al tag

```
<p> La radice di 2 vale <%= Math.sqrt(2.0) %> </p>
```

→ **Scriptlet:** `<% codice %>`

Frammenti di codice che controllano la generazione del codice HTML, valutati alla richiesta

```

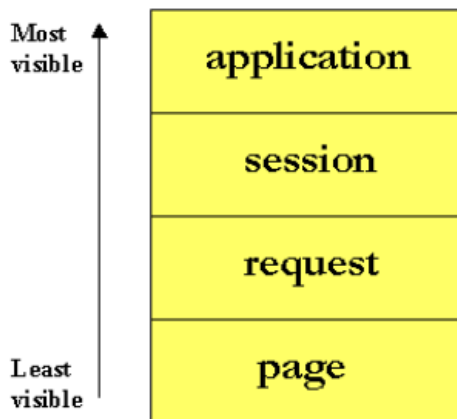
<table>
  <% for (int i=0; i< v.length; i++) { %>
    <tr><td> <%= v[i] %> </td></tr>
  <% } %>
</table>
  
```

19) Oggetti e loro scope

Gli **oggetti** possono essere creati in tre modi:

- implicitamente usando le direttive **JSP**;
- esplicitamente con le azioni;
- direttamente usando uno script (raro).

Gli oggetti hanno un attributo che ne definisce lo "scope"



20) Cosa si intende per JavaBean

Un **JavaBean** è una classe che segue regole precise (*specificata*):

- possiede un costruttore senza parametri;
- dovrebbe avere campi (*property*) private;
- i metodi di accesso ai campi (*property*) sono set/get/is
`setXxx`
`getXxx/isXxx`

con Xxx = property

```
1. class Book{
2.     private String title;
3.     private boolean available;
4.     void setTitle(String t) ...;
5.     String getTitle() ...;
6.     void setAvailable(boolean b) ...;
7.     boolean isAvailable () ...;
8. }
```

21) Accesso ad un JavaBean

Le azioni per utilizzare un *bean*:

- accedere ad un *bean* (inizializzazione)

```
<jsp:useBean id="user" class="com.jguru.Person" scope="session" />
```

```
<jsp:useBean id="user" class="com.jguru.Person" scope="session" >
    <% user.setDate(DateFormat.getDateInstance( ).format(new Date()));
</jsp:useBean>
```

- accedere alle proprietà

```
<jsp:getProperty name="user" property="name" />
<jsp:setProperty name="user" property="name" value="jGuru" />
<jsp:setProperty name="user" property="name" value="<%= expression %>" />
<jsp:useBean id="Attore" class="MyThread" scope="session" type="Thread"/>
```

→ Lo scope determina la vita e visibilità del bean:

a) **page**: è lo scope di default, viene messo in **pageContext** ed acceduto con `getAttribute`

b) **request**: viene messo in **ServletRequest** ed acceduto con `getAttribute`

c) **session e application**: se non esiste un bean con lo stesso id, ne viene creato uno nuovo.

Il **type** assegna una **superclasse** o un'**interfaccia**.

22) Pattern MVC: definizione e caratteristiche

Il **Model-View-Controller (MVC)** è un pattern architetturale che separa data model, user interface, e control logic in tre componenti distinte:

→ **Model**: rappresentato dai dati (gli oggetti) trattati dall'applicazione, e le operazioni su di essi;

→ **View**: la struttura dei dati restituiti al richiedente (e.g., la pagina HTML/CSS)

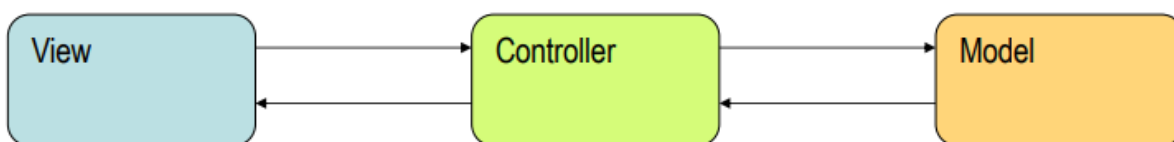
→ **Controller**: definisce le azioni da eseguire a fronte di una richiesta e interagisce con il Model per modificare i dati e con la View per generare la risposta.

Esso ha lo scopo di separare:

→ i dati e i metodi principali per la loro manipolazione (**Model**);

→ la presentazione, cioè l'interfaccia (**View**);

→ il coordinamento dell'interazione tra interfaccia (azioni degli utenti) e i dati (**Controller**).



23) Vantaggi e Svantaggi del pattern MVC

I **vantaggi** sono:

→ chiara separazione tra logica di business e logica di presentazione;

→ chiara separazione tra logica di business e modello dei dati;

→ ogni componente ha una responsabilità ben definita;

→ ogni parte può essere affidata a esperti.

Gli **svantaggi** sono:

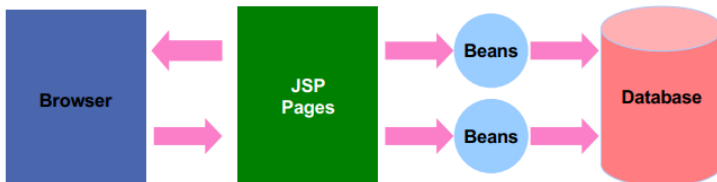
→ aumento della complessità dovuta alla concorrenza (è un sistema distribuito);

→ inefficienza nel passaggio dei dati alla view (un elemento in più tra cliente e controller).

24) Caratteristiche del Pattern Model 1

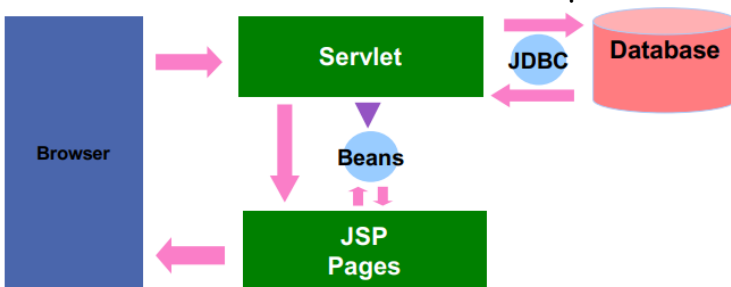
Lo scopo è la separazione tra dati, logica di business e visualizzazione. I passi sono:

- 1 - Il **browser** invia una richiesta per la pagina **JSP**;
- 2 - **JSP** accede a **Java Bean** e invoca la logica di business;
- 3 - **Java Bean** si connette al database e ottiene/salva i dati;
- 4 - La risposta, generata da **JSP**, viene inviata al **browser**.



25) Caratteristiche del Pattern Model 2

La richiesta viene inviata ad una **Java Servlet** che genera i dati dinamici richiesti dall'utente e li mette a disposizione della pagina **jsp** come **Java Beans**.



26) IoT: definizione e caratteristiche

L'Internet delle Cose (**IoT - Internet of Things**) si riferisce alla rete di dispositivi fisici connessi a **Internet**, che raccolgono e condividono dati tra loro. Questi dispositivi possono essere qualsiasi cosa, dagli oggetti di uso quotidiano come elettrodomestici, veicoli e dispositivi indossabili, a sistemi complessi come macchinari industriali e infrastrutture cittadine.

Le caratteristiche principali dell'**IoT**:

- **Connettività**: i dispositivi IoT devono essere connessi a Internet per comunicare e scambiare dati. La connettività può avvenire tramite Wi-Fi, Bluetooth, Zigbee, reti cellulari.
- **Sensori**: gli oggetti IoT sono dotati di sensori che raccolgono dati dall'ambiente circostante. Questi sensori possono misurare vari parametri come temperatura, umidità, movimento, luce, pressione, ecc.
- **Intelligenza**: i dati raccolti dai sensori vengono elaborati e analizzati per prendere decisioni intelligenti. Questo può avvenire direttamente sul dispositivo (edge computing) o su server remoti (cloud computing).

- **Automazione**: i sistemi IoT possono automatizzare compiti senza necessità di intervento umano. Ad esempio, un termostato intelligente può regolare automaticamente la temperatura in base alle preferenze dell'utente e ai dati ambientali.
- **Interoperabilità**: i dispositivi IoT devono essere in grado di comunicare tra loro indipendentemente dal produttore o dal protocollo utilizzato. Questo richiede standard di interoperabilità per garantire che i diversi dispositivi possano lavorare insieme senza problemi.
- **Scalabilità**: la rete IoT deve essere in grado di espandersi per includere un numero sempre maggiore di dispositivi. Questo significa che le infrastrutture sottostanti devono essere progettate per supportare una grande quantità di dati e connessioni.
- **Sicurezza**: la sicurezza è una delle principali preoccupazioni dell'IoT, dato che i dispositivi connessi possono essere vulnerabili a cyber-attacchi. Misure di sicurezza robuste, come la crittografia dei dati e l'autenticazione degli utenti, sono essenziali per proteggere i dispositivi e le informazioni.
- **Analisi dei Dati**: l'analisi dei dati è una componente cruciale dell'IoT. I dati raccolti dai dispositivi devono essere analizzati per ottenere insights utili, migliorare le prestazioni dei dispositivi e fornire valore agli utenti.
- **Manutenibilità**: i dispositivi IoT devono essere facili da mantenere e aggiornare. La possibilità di aggiornare il software da remoto (OTA - Over The Air) è importante per correggere bug, migliorare le funzionalità e risolvere problemi di sicurezza.
- **Personalizzazione**: i dispositivi IoT spesso offrono livelli di personalizzazione elevati, consentendo agli utenti di configurare i dispositivi in base alle loro preferenze e necessità specifiche.

27) IoT e servizi

I dispositivi elettronici sono in grado di raccogliere e scambiare dati, ed eseguire azioni per realizzare le piattaforme del futuro. Si ha quindi un elevato numero di componenti coinvolti in singole applicazioni, la riconfigurazione dinamica delle applicazioni e le cose consumano/espongono funzionalità (servizi).



28) Services Computing: definizione

Service Computing è un paradigma che integra vari servizi attraverso piattaforme di calcolo per creare soluzioni flessibili, scalabili e riutilizzabili. Si basa su architetture orientate ai servizi (SOA) e utilizza tecnologie come servizi web, cloud computing, e microservizi per fornire servizi informatici che possono essere scoperti, invocati e combinati in modo dinamico. In altre parole si riferisce all'insieme delle metodologie, delle tecnologie e delle pratiche che permettono la creazione, l'implementazione, la gestione e l'integrazione di servizi software che possono essere offerti attraverso una rete (**Internet**).

29) Architettura orientata ai servizi - SOA: definizioni e caratteristiche

SOA è uno stile architettonico che si concentra su elementi discreti riutilizzabili (chiamati **servizi**), invece che su un design monolitico, per costruire le applicazioni. Un **servizio** fornisce funzionalità ai richiedenti (che possono essere altri servizi). L'accesso al servizio è fornito attraverso la rete (anche **Internet**).

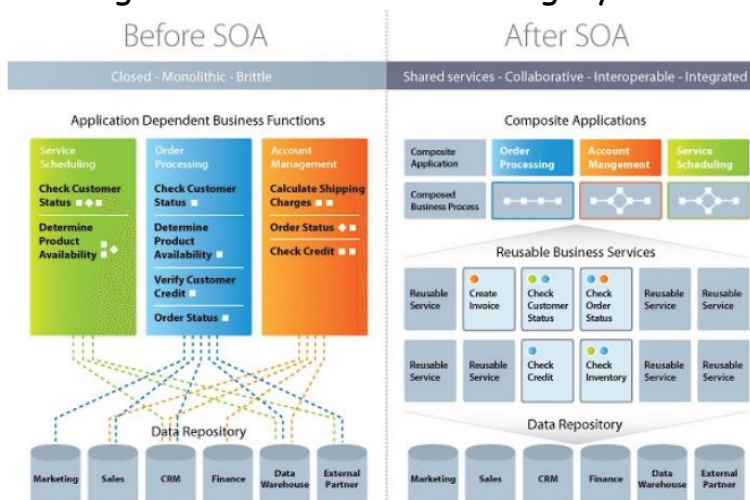
30) Architettura orientata ai servizi - SOA: vantaggi e svantaggi

I **vantaggi** sono:

- Riutilizzabilità
- Processo di sviluppo agile e orientato al business
- Economia dei servizi
- Scalabilità
- Ottimizzazione e riduzione dei costi

Gli **svantaggi** sono:

- Gestione del ciclo di vita complesso
- Dependency Hell
- Integrazione con le soluzioni legacy



31) Cosa si intende per servizio web?

Un **servizio** è un'entità software indipendente che può essere scoperta e invocata da altri sistemi software attraverso una rete. In altre parole un servizio Web è un'applicazione software che viene identificata da un **URI** (**Uniform Resource Identifier**), le **interfacce** [...] sono in grado di essere definite, descritte e scoperte [...] e supporta interazioni dirette con altri software per mezzo di [...] messaggi e protocolli basati su **Internet**.



32) Quali sono gli elementi fondamentali di SOA?

Gli elementi fondamentali di **SOA** sono:

a) **Componenti**: servizio, descrizione del servizio;

b) **Ruoli**:

→ **Service Providers - Fornitori di servizi**: offrono servizi/funzionalità.

→ **Service Broker**: cataloghi di servizi di menage.

→ **Service Requestors - Richiedenti di servizi**: trovano un servizio e interagiscono con i provider.

c) **Operazioni**:

→ **Publish** (un servizio);

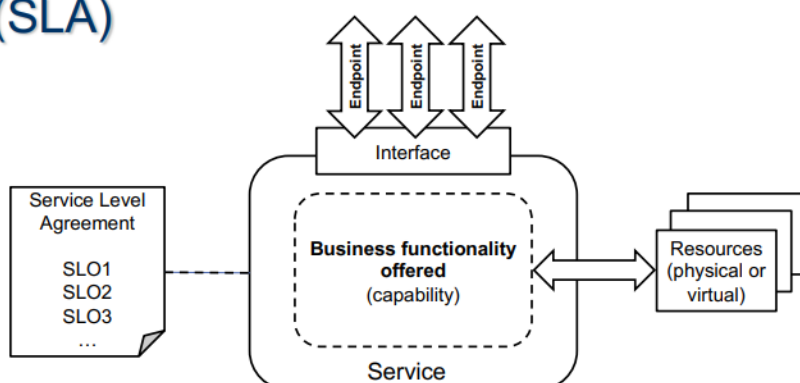
→ **Find** (servizio/endpoint);

→ **Interact** (ad esempio, richiesta-risposta).

33) Accordo sul livello di servizio (SLA): definizione e caratteristiche

Lo **SLA** è un contratto tra il provider e l'utente che assicura che la funzionalità sia fornita correttamente e garantisce proprietà non funzionali. Esso comprende diversi **SLO (Service Level Objectives)** che definiscono la qualità del servizio da garantire attraverso metriche specifiche.

(SLA)



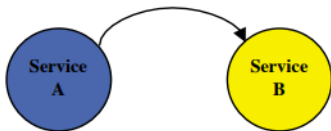
34) Quali sono le peculiarità del WS?

Le peculiarità del Web Service sono:

- **Componenti pubblici** (scopribili con interfacce pubbliche).
- **Componibilità**: servizi composti e coordinamento.
- **Descrizioni semantiche** (scoperta, composizione, recommending systems).
- **QoS**: base, context awareness;
- **Organizzazione del sistema**: Peer to Peer, ESB, Grid.

35) Cosa si intende per servizio di composizione?

Una **composizione** consiste in un insieme di servizi interconnessi, che possono essere utilizzati come nuovi servizi in altre composizioni. Due **servizi sono interconnessi** se almeno uno dei due richiede la funzionalità esposta (alias endpoint, alias API) dell'altro.



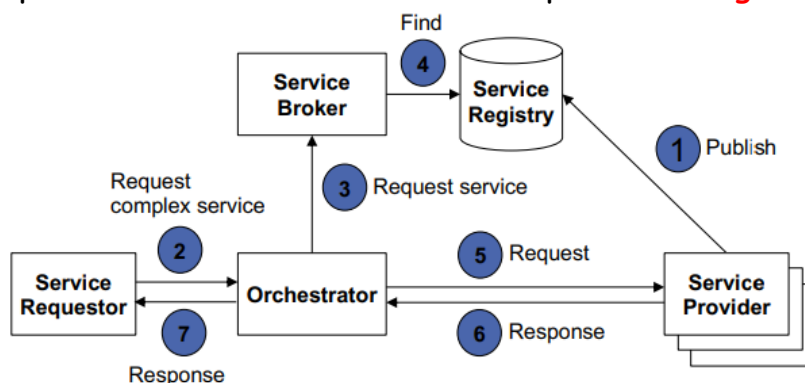
36) Cosa si intende per Business Processes?

Il **Business Process** è un insieme di attività correlate (workflow) eseguite da persone e applicazioni per ottenere un risultato ben definito (servizio o prodotto). I **BP software** sono creati dalla composizione (integrazione) di servizi.

37) Differenza tra l'orchestrazione e la coreografia

L'**orchestrazione** descrive come i servizi interagiscono tra loro, compresa la logica di business e l'ordine di esecuzione delle interazioni dal punto di vista e sotto il controllo di un singolo attore (servizio). La **coreografia** descrive la sequenza di interazioni tra più parti coinvolte nel processo dal punto di vista di tutte le parti. Definisce lo stato condiviso delle interazioni tra le entità.

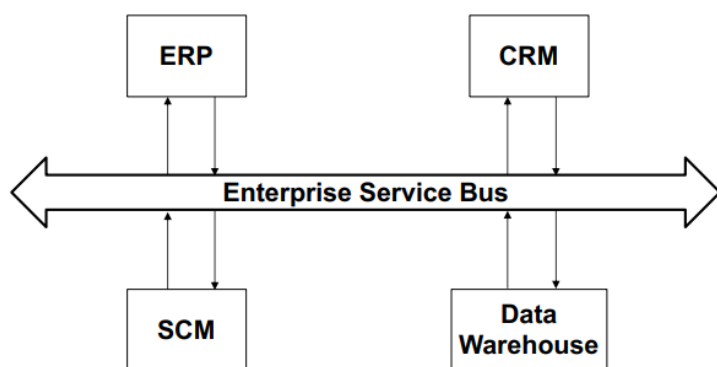
Un'**organizzazione** assume il ruolo di orchestratore e si fa carico di implementare il servizio controller per altre **organizzazioni**.



38) Definizione di Enterprise Service Bus

L'**Enterprise Service Bus (ESB)** è un sistema di comunicazione per supportare l'interazione e la comunicazione tra i componenti di un sistema informativo. Le caratteristiche principali sono:

- instradamento dei messaggi tra applicazioni e servizi;
- trasformazione del messaggio;
- comunicazione sicura;
- architettura estensibile.



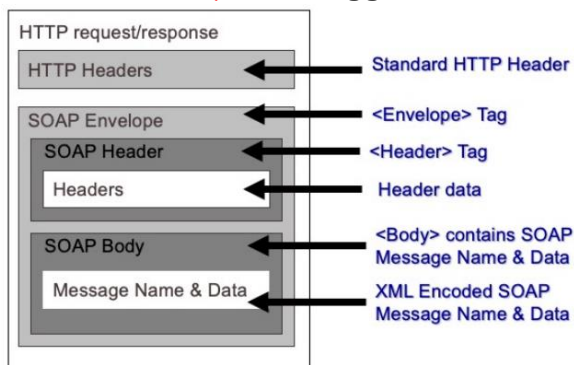
39) SOAP: definizione e caratteristiche

Si definisce **SOAP (Simple Object Access Protocol)** un protocollo basato su **XML** per lo scambio di informazioni strutturate e tipizzate. È stato uno dei primi protocolli utilizzati per i servizi web. **SOAP** è stato uno dei primi standard per i servizi web e supporta operazioni complesse e contratti di servizio formali tramite **WSDL**.

40) Componenti di un messaggio SOAP

I componenti di un messaggio SOAP sono:

- **SOAP Envelope**: ingloba il contenuto del messaggio;
- **SOAP Header**: maggiore (opzionale): Maggiore flessibilità, può essere elaborato dai nodi tra l'origine e la destinazione e contiene blocchi di informazioni su come elaborare il messaggio;
- **SOAP Body**: messaggio effettivo da consegnare ed elaborare.



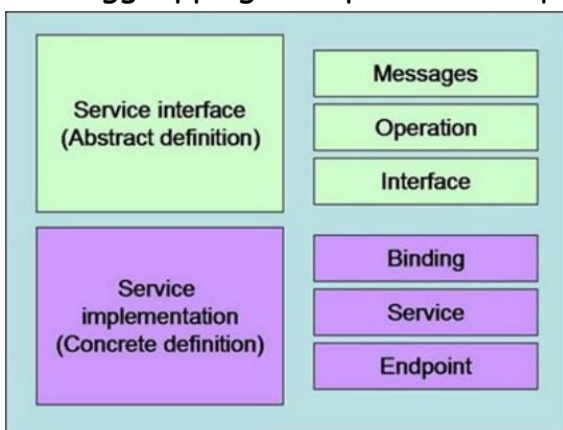
41) Che cos'è WSDL?

Si definisce **WSDL (Web Service Description Language)**, la descrizione formale del servizio web, che specifica le operazioni disponibili, i parametri richiesti, i tipi di dati utilizzati e i protocolli di comunicazione supportati.

42) WSDL 2.0: modello concettuale

Il modello concettuale di WSDL 2.0 è costituito da:

- **Parte astratta**: riferito alla descrizione di un servizio in termini di messaggi che invia e riceve attraverso un sistema di tipi, tipicamente **W3C XML Schema**. I componenti sono l'operazione, che associa modelli di scambio di messaggi a uno o più messaggi, modelli di scambio di messaggi, che definiscono la sequenza e la cardinalità dei messaggi scambiati tra i nodi (servizi) e un'interfaccia che raggruppa queste operazioni in modo indipendente dal canale di comunicazione.
- **Parte concreta**: si hanno i **binding**, che specificano il protocollo di trasporto per le interfacce, un endpoint, che associa un URI a un **binding**, e un servizio che raggruppa gli endpoint che implementano un'interfaccia comune.



43) Definizione e caratteristiche di REST

Si definisce **REST**, un'architettura più leggera rispetto a SOAP, che utilizza i metodi standard **HTTP (GET, POST, PUT, DELETE)** per operazioni **CRUD (Create, Read, Update, Delete)**. È molto utilizzato grazie alla sua semplicità e flessibilità.

44) Principi di REST

I principi **REST** sono:

- **Architettura Client-Server**: Il client e il server devono essere separati e indipendenti l'uno dall'altro. Il client invia richieste al server, che elabora le richieste e invia indietro le risposte.
- **Statelessness (Senza stato)**: Ogni richiesta del client al server deve contenere tutte le informazioni necessarie per comprendere e gestire la

richiesta. Il server non deve mantenere lo stato delle sessioni del client tra le richieste.

→ **Cache**: Le risposte alle richieste devono essere esplicitamente o implicitamente contrassegnate come cacheable (memorizzabili) o non-cacheable. La cache può essere utilizzata per migliorare le prestazioni e ridurre la latenza.

→ **Interfaccia uniforme**: Le risorse (dati) devono essere identificate in richieste individuali tramite un URI (Uniform Resource Identifier). I client manipolano le risorse attraverso le rappresentazioni (ad esempio JSON, XML) che vengono trasferite tra client e server e ogni messaggio deve contenere tutte le informazioni necessarie per comprendere la richiesta. Le risposte devono contenere metadati sufficienti per descrivere come trattare il payload.

→ **Sistema a livelli**: L'architettura può essere composta da più livelli di componenti, ad esempio proxy, cache o gateway, che semplificano e migliorano la scalabilità dell'architettura.

→ **Manipolazione delle risorse tramite metodi HTTP: REST** utilizza i metodi standard **HTTP** (**GET**, **POST**, **PUT**, **DELETE**) per manipolare le risorse. Questi metodi corrispondono alle operazioni **CRUD** (**Create**, **Read**, **Update**, **Delete**) che si possono eseguire su una risorsa.

45) Caratteristiche della cache

La **cache** riduce la **latenza** e il **traffico di rete** e vengono memorizzati nella cache solo le risposte ai **GET** (non **SSL**), non i **POST** ecc. Vi sono due tipi di cache:

→ **Lato client (ad esempio nel browser)**

→ **Proxy/server**.

46) Ricetta di REST

La ricetta REST è la seguente:

→ definire i sostantivi

→ definire i formati

→ scegliere le operazioni

→ evidenziare i codici di ritorno

47) Codici di Stato della Risposta

Il codice di stato della risposta viene generato dal server per indicare l'esito di una richiesta e il codice di stato è un numero di tre cifre:

→ **1xx (informativo)**: richiesta ricevuta; il server sta continuando il processo.

- **2xx (successo)**: richiesta ricevuta, compresa, accettata e servita.
- **3xx (reindirizzamento)**: è necessario intraprendere ulteriori azioni per completare la richiesta.
- **4xx (errore del cliente)**: La richiesta contiene una sintassi errata o non può essere compresa.
- **5xx (errore del server)**: Il server non è riuscito a soddisfare una richiesta apparentemente valida.

48) Cosa si intende per controllo ipermediale?

Il concetto di **controllo ipermediale** si riferisce a una delle caratteristiche fondamentali dell'architettura **REST (Representational State Transfer)**. È strettamente legato al principio di **interfaccia uniforme di REST** e rappresenta un modo per gestire e navigare tra le risorse tramite ipermedia. Il **controllo ipermediale** implica che l'applicazione client, che interagisce con un servizio web **RESTful**, riceve dai **server** una rappresentazione dello stato attuale delle risorse, insieme a informazioni ipermediali (solitamente sotto forma di link o metadati) che guidano l'utente su come navigare e interagire con il sistema.

Le caratteristiche sono:

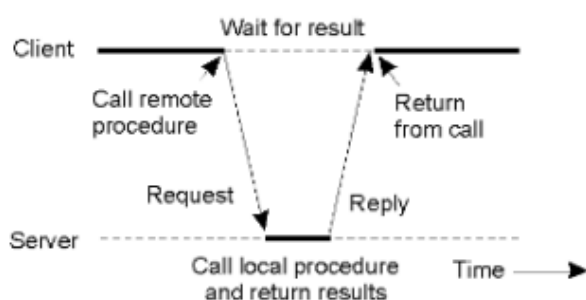
- Risorse e Link
- Navigazione basata su HATEOAS
- Indipendenza dell'implementazione
- Flessibilità e scalabilità.

Capitolo 6 - Remote Procedure Call

49) Remote Procedure Call: definizione, vantaggi e svantaggi

Le **procedure remote** sono una estensione distribuita del normale protocollo di chiamata di procedura. Hanno una semantica nota e sono facili da implementare, ma sono statiche e non c'è concorrenza, cioè sono bloccanti.

Modello RPC



Modello RPC asincrona

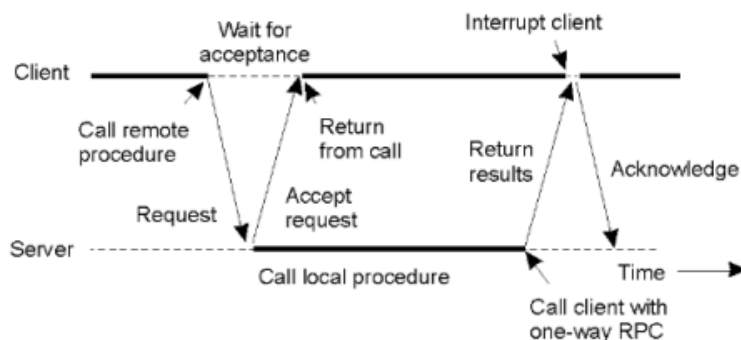
Interazione tra **client** e **server** utilizzando una **RPC asincrona**, senza una

risposta: il server restituisce il controllo prima di eseguire la richiesta.



Modello RPC sincrona

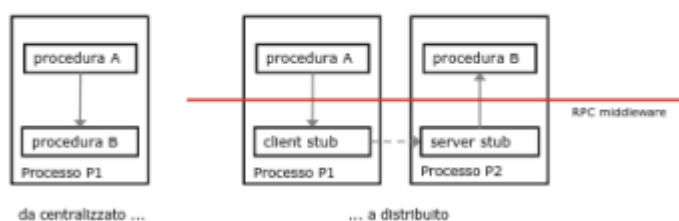
Interazione tra **client** e **server** utilizzando una **RPC asincrona**, attraverso una call back che non è altro che una risposta posticipata.



50) Architettura RPC

L'**architettura RPC** introduce l'utilizzo degli stub (rappresentazione delle controparti) per simulare comportamenti locali per chiamante e chiamato e per realizzare la comunicazione tra processi remoti.

Figura 6.4: implementazione degli stub

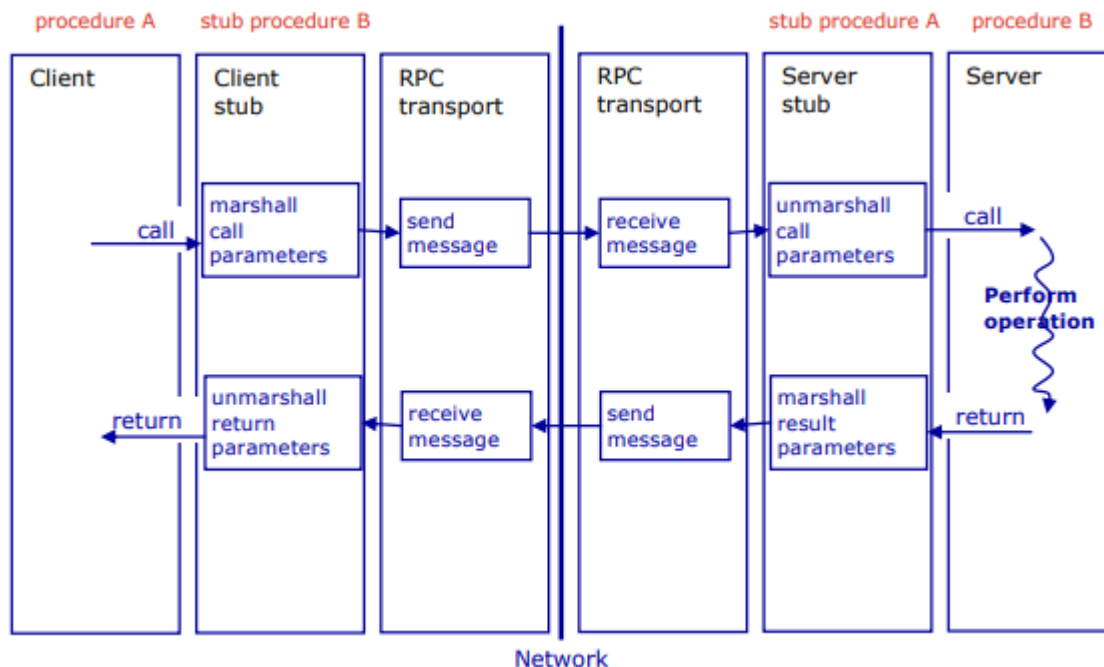


Quando si implementa una computazione remota tramite RPC, la computazione stessa segue una serie di step:

- a - il **client** richiama la procedura remota
- b - lo **stub** costruisce il messaggio
- c - il **messaggio** viene inviato, attraverso la rete, al server
- d - il **sistema operativo del server** gestisce il messaggio e lo trasferisce allo stub del server
- e - lo **stub** spacchetta il messaggio
- f - lo **stub** gestisce localmente la richiesta

I parametri localmente vengono passati attraverso i vari step attraverso dei puntatori, invece quando si arriva alla call remota si replica lo stack nel processo remoto. Questa operazione si chiama **marshalling** e **unmarshalling**.

Esecuzione di una RPC



51) Oggetti distribuiti

Si hanno i seguenti tipi di oggetti:

- **oggetti**: incapsulano i dati e definiscono le modalità di accesso e le operazioni che si effettuano su di loro;
- **oggetti a compile-time**: definiti attraverso interfacce e classi;
- **oggetti a run-time**: accessibili attraverso adapters, detti anche wrapper
- **oggetti persistenti e transienti**;
- **riferimenti**: a degli oggetti remoti

52) Differenza tra puntatori e riferimento

Un **puntatore** è una variabile che contiene un indirizzo di memoria che può contenere qualsiasi cosa. Può essere modificato in ogni momento e non è tipizzato. Un **riferimento** è una variabile che contiene informazioni logiche (alias) per accedere ad un oggetto. È immutabile e può essere inizializzato all'atto della creazione di un oggetto. È **fortemente tipizzato**.

Sono **distribuibili** e possono essere referenze a:

- indirizzo della macchina;
- indirizzo del server;
- identificatore dell'oggetto.

53) Java RMI: definizioni e caratteristiche

La **Java RMI (Remote Method Interface)** è un middleware che estende l'approccio **object - oriented** al distribuito e che supporta l'invocazione di metodi tra oggetti su macchine virtuali distinte. Si basa sulla portabilità del **bytecode** e sulla macchina virtuale.

54) Java RMI: tipi di invocazioni

La **Java RMI** è simile all'approccio **RPC** per la gestione dei parametri per valore, consente anche il passaggio parametri per reference e definisce stub specifici per ogni oggetto (mentre in RPC sono generici). Si hanno due tipologie di invocazioni:

→ **Invocazioni statiche**: Interfaccia nota in compilazione.

→ **Invocazioni dinamiche**: L'invocazione include informazioni logiche sull'identità dell'oggetto e del metodo.

55) Java RMI: trasferimento dei parametri

Il trasferimento può avvenire per:

→ **valore**: utilizzato con i tipi primitivi e con gli oggetti se serializzabili;

→ **reference**: e i riferimenti ad oggetti remoti vengono passati per valore per permettere invocazioni remote, implementando le interfacce Remote e Serializable.

56) Java RMI: serializzazione degli oggetti

La **serializzazione** rappresenta lo stato di un oggetto come **stream di byte**. È essenziale per poter memorizzare e ricostruire lo stato degli oggetti, in modo da poterli trasferire via rete e renderli persistenti. Il **meccanismo di loading** dinamico di **Java** permette di passare solo le informazioni essenziali sullo stato, mentre la descrizione della classe può essere caricata a parte. La serializzazione usa il metodo `writeObject(Object obj)` della classe `Object` che attraversa tutti i riferimenti contenuti in `obj` per costruire una rappresentazione completa del grafo.



57) Java RMI: identificazione degli oggetti

Utilizza nomi assegnati dall'utente e una directory (o naming) service per

convertirli in **reference** operativi. Le **directory service** devono essere disponibili ad un **host** e porta noti. **RMI** definisce un **rmiregistry** che sta su ogni macchina che ospita oggetti remoti, convenzionalmente alla porta **1099**, con un servizio di ascolto collocato attivato dalla stessa **RMI**.

58) Java RMI: classe Naming

La **classe naming** fornisce accesso diretto alle funzionalità del **RMI registry**. I parametri sono stringhe in formato **URL** riferiti al **registry** e all'oggetto remoto considerato, mentre fornisce una serie di **metodi statici** quali:

- **lookup**: restituisce un **riferimento**, uno **stub**, all'oggetto associato al nome specificato;
- **bind**: collega il nome specificato all'oggetto remoto;
- **list**: restituisce i nomi, in formato **URL**, degli oggetti del **registry**;
- **unbind**: distrugge il collegamento al nome specificato;
- **rebind**: collega il nome specificato all'oggetto remoto, cancellando i collegamenti esistenti. Si potrebbe pensare di usare un'altra volta il metodo **bind**, ma potrebbero esserci problemi di concorrenza e nomi duplicati.

59) Architettura di RMI

Il **server** pubblica il **reference** e il nome dell'oggetto remoto nel **registry** invocando il metodo **bind**, mentre il **client** ottiene il **reference** all'oggetto invocando il metodo **lookup** e accede all'oggetto remoto.

60) Come crea un oggetto remoto Java RMI?

Java definisce un'interfaccia per implementare oggetti remoti, la `java.rmi.Remote`. Per creare una **classe remota** serve per:

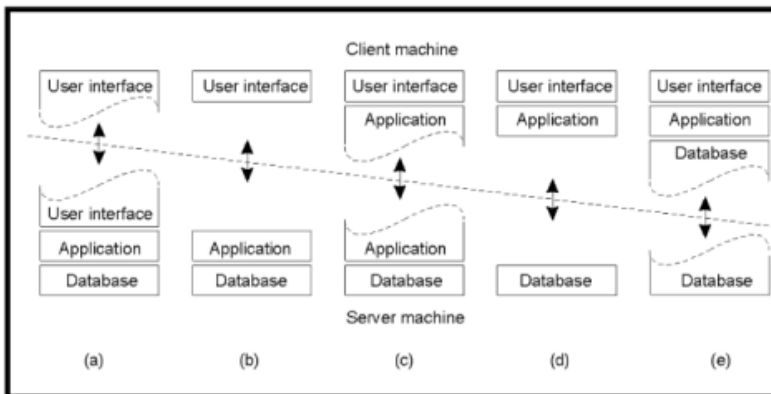
1. definire l'interfaccia della classe remota;
2. implementare la nuova interfaccia;
3. implementare un server che crei e registri l'oggetto al **Registry**.

Capitolo 7 - JavaScript

61) Architettura Multitier

Ci sono differenze logiche e tecniche tra le diverse architetture con cui il modello a 3-tier può presentarsi, in particolare ci possono essere 5 varianti, a

seconda di quanto carico si vuole affidare al client e quanto al server.



62) Cos'è AJAX e quali sono le sue caratteristiche?

Si dice **AJAX** un insieme di tecniche di sviluppo di applicazioni web dinamiche con un modello asincrono e tale approccio permette di:

- alleggerire il lavoro lato server nella validazione dei dati nei form;
- auto - completare i **form**;
- aggiornare parte di pagina, senza dover riaggiornare la pagina intera;
- introduce controlli **UI** più sofisticati.

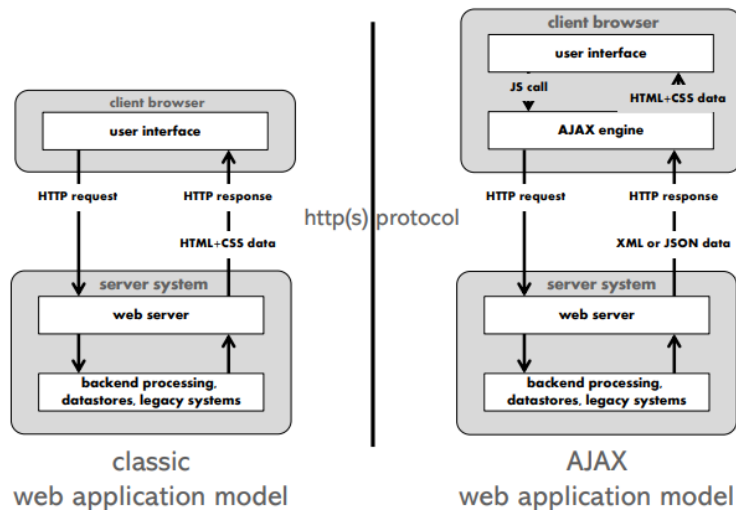
63) JavaScript: definizione e caratteristiche

JavaScript è un linguaggio di scripting ad oggetti, non tipizzato e interpretato da un **engine**, che in pratica è il **browser**. **Node.js** permette l'esecuzione di **JavaScript** lato server. Consente di rendere le pagine html dinamiche, cioè di inserire dei programmi che modificano il comportamento e le visualizzazioni, cioè di inserire dei programmi che modificano il comportamento e le visualizzazioni. Esso è importante perché:

- ha la capacità di effettuare richieste **HTTP** al **server**, in maniera trasparente all'utente;
- la funzione di rendere asincrona la comunicazione tra **browser** e **web browser**. Può richiedere dati in formato **XML** o in formato **JSON**.

64) AJAX: differenza tra architettura classica vs dinamica

La differenza principale è che si interpone, tra **client** e **server web**, un **AJAX engine** che elabora la risposta fornita dal **server** e la presenta al **client**.



65) Problemi delle RIA

Ci possono essere alcuni problemi alle tecniche **RIA**, conseguenti soprattutto dalle caratteristiche asincrone e dall'elaborazione lato client:

- interferenza con la funzionalità del tasto "indietro" del browser, che potrebbe presentare pagine diverse da quelle precedentemente viste;
- alcune parti della pagina potrebbero cambiare inaspettatamente;
- aumenta il carico da parte del browser, bisogna infatti usare una **balancer**: capire quanto è conveniente usare il codice, e quanto i dati ;
- debug complesso (possono esserci errori sul server o sul client o in entrambe le parti);
- completa visibilità della sorgente, quindi si introducono problemi di sicurezza e affidabilità.

66) Come avviene la trasmissione dei dati?

La **trasmissione dei dati** tra **server** e **applicazioni RIA** avviene in diversi formati ed è un aspetto critico, la scelta del formato influenza la struttura dell'applicazione e le sue performance, in particolare un formato richiede del tempo per predisporre i dati, trasferirli, fare il **parsing** dei dati ricevuti e infine farne il rendering per renderli disponibili all'interfaccia.

67) Struttura pagina AJAX

La struttura di una pagina **AJAX** si attiene al formato **HTML**, quindi si ha una **struttura HTML** che include i contenuti da presentare, l'interfaccia utente e le modalità di interazione input e output. Gli **script AJAX** sono inclusi nei tag `<script></script>`, devono essere posti nella head, oppure in coda al resto del codice **HTML**.

68) Output dei dati

JavaScript non fornisce alcuna funzione per l'output ma risulta possibile utilizzare le seguenti tecniche:

- `innerHTML`: **sostituisce** il contenuto di un elemento html con un identificatore;
- `window.alert()`: apre una nuova finestra, **alert box**, con il contenuto indicato;
- `console.log()`: scrive sulla console messaggi di debugging;
- `document.write()`: sostituisce l'intera pagina.

69) Flusso di controllo di JavaScript

La chiamata ad una funzione avviene come una **callback**, si invoca quando:

- avviene un certo evento;
- **JavaScript** invoca la funzione;
- automaticamente (**self invoked**).

70) Eventi di JavaScript

Un evento **JavaScript** è un qualcosa che avviene ad un elemento **HTML**. Per aggiungere un evento ad un elemento, si utilizza la sintassi:

```
<element event = "JavaScriptCode()">
```

71) Paradigma ad eventi

L'applicazione deve essere puramente reattiva, ovvero non è possibile identificare staticamente un flusso di controllo unitario. Il programma principale inizializza l'applicazione ed istanzia gli osservatori e associando gli opportuni **handler**.

72) AddEventListener

Si può associare uno o più gestori ad ogni elemento del DOM HTML che genera eventi, utilizzando la funzione `element.addEventListener(event, function)`

Codici di esempio

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript Example</title>
  </head>
  <body>
    <p>Click the button to display the date.</p>
    <input type = "button" value = "What time is it? " onclick = "displayDate()">
    <p id = "demo"></p>
    <script>
      function displayDate() {
        document.getElementById("demo").innerHTML = Date();
      }
    </script>
  </body>
</html>
```




Click the button to display the date.

What time is it? =

Thu Jul 04 2024 15:50:08 GMT+0200 (Ora legale dell'Europa centrale)

```
<html>
  <body>
    Text input: <input type="text" id="txt1" onkeyup="echo1(this.value)">
    <p>Echo: <span id="demo"></span> </p>
    <script>
      function echo1(str) {
        document.getElementById("demo").innerHTML = str;
      }
    </script>
  </body>
</html>
```



Text input: ciao come stai?

Echo: ciao come stai?

```
<html>
  <title> Echo 1 </title>
  <body>
    <form action="">
      Text input: <input type="text" id="txt1" onkeyup="echo()">
    </form>
    <p>Echo: <span id="demo"></span></p>
    <script>
      function echo() {
        document.getElementById("demo").innerHTML = document.getElementById("txt1").value;
      }
    </script>
  </body>
</html>
```

73) Programmazione in Javascript: Variabili e costanti

Le **variabili** sono dichiarate `var` e sono **global scope** o solo all'interno di funzioni (**function scope**), mentre le variabili `let` sono visibili solo all'interno del blocco in cui sono dichiarate (**block scope**). Infine le `const` permettono di definire variabili `let` con un valore costante, cioè che non può essere modificato.

74) Programmazione in Javascript: Forme sintattiche per le funzioni

Vi sono tre modi per definire delle funzioni:

→

```
function f(x, y) {  
  return x * y;  
}
```

→

```
var f = function f(x, y) {  
  return x * y;  
}
```

→ Arrow function

```
const f = function f(x, y) {  
  return x * y;  
}
```

75) Programmazione in Javascript: Tipo String

Sono usate per memorizzare e modificare del testo, e corrisponde a zero o più caratteri scritti tra apici, doppi o singoli indifferentemente. Esistono dei metodi per manipolarle:

- `indexOf()`: indice della prima occorrenza di una stringa in un'altra;
- `lastIndexOf()` come il precedente, ma trova l'ultima occorrenza;
- `startsWith()` e `endsWith()`;
- `slice()` estrae un pezzo di stringa

76) Programmazione in Javascript: come è possibile manipolare gli array

Gli array, dichiarati come `const`, vengono manipolati con diverse funzioni standard:

- `push()`: inserisce un elemento in ultima posizione
- `pop()`: rimuove l'ultimo elemento
- `shift()`: rimuove il primo elemento e effettua lo shift degli altri
- `unshift()`: inserisce un elemento in prima posizione
- `splice()`: inserisce un elemento ad un indice dato

77) Programmazione in Javascript: le classi

Servono per definire dei prototipi di **oggetti**, per dichiarare più entità dello stesso tipo.

78) Programmazione in Javascript: elementi di programmazione aggiuntivi

Ci sono alcuni elementi di programmazione caratteristici e con funzionalità che aiutano la programmazione, come:

- `forEach`: itera gli elementi di un array;
- `map`: mappa gli elementi di un array, data una funzione;
- `filter`: crea un nuovo array con gli elementi di un array che passano un

certo test filtro, programmato come una funzione booleana;

→ reduce: produce un singolo valore da un array.

79) Interazione con il server: XMLHttpRequest

Tale oggetto viene utilizzato per lo scambio dei dati con il server dietro le quinte e ciò avviene mediante la creazione della variabile `var variable = new XMLHttpRequest();` e poi si usano i metodi:

→ `open(method, url, async)`: specifica il tipo di richiesta;

→ `send()`: invia la richiesta di tipo GET;

→ `send(string)`: invia la richiesta di tipo POST.

Codice

```
<script>
  function loadDoc() {
    const xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
      if (xhttp.readyState == 4 && xhttp.status == 200)
        document.getElementById("demo").innerHTML = xhttp.responseText;
    };
    xhttp.open("GET", "ajax_text.txt", true);
    xhttp.send();
  }
</script>
```

80) Interazione con il server: Evento onreadystatechange

L'evento `onreadystatechange` è lanciato ogni volta che la `readyState` cambia.

Gli stati possibili sono:

→ UNSENT creata ma non chiamata

→ OPENED chiamata ma non inviata

→ HEADERS_RECEIVED ricevuti gli headers

→ LOADING scaricamento dei dati, conservati in `responseText`

→ DONE operazione completata

81) Interazione con il server: risposte del server

Le risposte del server possono essere:

→ `ResponseText`: se la risposta non è XML, si utilizza questo metodo, che restituisce una stringa.

→ `ResponseXML`: se la risposta è in formato XML, possiamo ottenere direttamente un oggetto XML, senza passare per il formato testuale.

82) Oggetti JSON: Formato JSON

JSON (JavaScript Object Notation) è un formato leggero di scambio dei dati,

facile da leggere e scrivere per l'uomo e per la macchina. È costruito su due strutture universali:

- **coppie chiave/valore**: una collezione di coppie che associano ad ogni chiave un valore
- **list**: una sequenza di valori
- **oggetto**: è una collezione, non ordinata, di coppie chiave valore
- **array**: collezione ordinata di valori
- **valore**: una stringa (sequenza di caratteri UNICODE), un numero, un booleano, un valore nullo un oggetto o una stringa.

83) Oggetti JSON: Interoperabilità con JavaScript

Un oggetto **JSON** può essere semplicemente convertito in un oggetto **JavaScript** e viceversa.

84) Node.js: descriverne le caratteristiche

Node.js è una piattaforma realizzata con il motore **JavaScript**, che permette di realizzare applicazioni web veloci e scalabili, che usa un modello di **I/O** non bloccante e ad eventi e consente la realizzazione di **applicazioni lato server** scritte in **JavaScript**.

85) Node.js: descrivere brevemente modello ad eventi

Il **modello ad eventi** è sviluppato su un singolo **thread**, è un loop che esegue e verifica periodicamente l'avvenire di un certo evento. Il lato negativo di questo modello è che le esecuzioni lunghe comportano il blocco **UI**.

86) Node.js: approccio asincrono

Node.js può avere un approccio sincrono o asincrono. Nell'approccio asincrono, largamente più utilizzato, il server **Node.js** non si aspetta che l'effetto di una funzione sia completato prima di eseguire la prossima funzione, per cui crea una funzione di **callback** che completa l'effetto desiderato mentre l'esecuzione prosegue.

87) Node.js: modello di esecuzione

L'esecuzione dei programmi in **Node.js** si basa su un single event loop, che preleva un evento da una singola coda e lo serve eseguendo le operazioni previste. Il **loop** è ad un livello logico, le operazioni sono eseguite logicamente in sequenza: a livello fisico ogni operazione è eseguita da un **thread** autonomo che permette l'esecuzione parallela e concorrente. Questo modello si basa

sull'esecuzione di operazioni **stateless** e disaccoppia la programmazione dall'esecuzione favorendo la **scalabilità**.

88) Node.js: come avviene la gestione HTTP?

Node.js permette di realizzare un **Server Web** che può gestire pagine html e applicazioni scritte in **JavaScript**.

89) Node.js: modulo del file system

Il modulo **filesystem** `fs` fornisce funzioni standard di gestione dei file.

90) Realizzare una Web app in un Application Server: il framework **express**

Il modulo `express.js` è un framework web che fornisce gli strumenti per realizzare un server che può ospitare sia risorse statiche, sia applicazioni che generano rappresentazioni dinamiche.

91) Realizzare una Web app in un Application Server: installazione

L'installazione richiede la sequenza di 3 passi:

```
gianl@DESKTOP-BU3868V MINGW64
t
$ mkdir server-web

gianl@DESKTOP-BU3868V MINGW64
t
$ cd server-web/

gianl@DESKTOP-BU3868V MINGW64
t/server-web
$ npm install express
```

92) Realizzare una Web app in un Application Server: esecuzione

Nella cartella dove sono presenti i sorgenti dell'applicativo, eseguire `node example.js`.

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

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(3000, localhost,
  function () {
    var host = server.address().address
    var port = server.address().port
  })
```

Capitolo 8 - Semantica dei dati

93) Dati semantici: perché si è definito il metadata?

Perché il metadata permette sia alla macchina che all'uomo di comprendere un oggetto **JSON** complicato ed esteso qualificandone il contenuto mediante l'inserimento di ulteriori informazioni.

94) Dati semantici: descrivere l'approccio lightweight collaborative respositories

L'approccio **lightweight collaborative respositories** offre un semplice schema per specifiche descrizioni di semantica, definendo di fatto dei modi per descrivere concetti semplici come persone, cose o luoghi.

95) Dati semantici: schema.org

Lo **schema.org** è una community collaborativa con la missione di creare, mantenere e promuovere schemi di **strutture dati** su **Internet**. Un **vocabolario condiviso** rende più semplici le decisioni sullo schema.

96) JSON - LD

Il **JSON - LD** offre una via semplice per aggiungere significato semantico ai documenti **JSON** aggiungendo delle informazioni id contesto e **hyperlinks** per descrivere la semantica dei diversi elementi degli oggetti **JSON**.

Le tre principali keyword sono:

- @content: URL che si riferisce ad un particolare schema in rete;
- @id: identificatore univoco, solitamente un URI;
- @type: un URL che si riferisce al tipo di un valore.

97) Grafi di conoscenza: cos'è la semantic web?

La **semantic web** è un'estensione del web che promuove formati di dati comuni per facilitare lo scambio di dati significativi tra delle macchine.

98) Grafi di conoscenza: cos'è il linked data?

Sono un set di best practice per pubblicare e connettere dati strutturati sul web, in modo tale che le risorse web siano interconnesse in modo da consentire alle macchine di comprendere automaticamente i tipi e i dati di ogni risorsa.

99) RDF: definizione e caratteristiche

Si definisce **RDF** un modello dati per rappresentare dati sul web, basato su tre elementi:

- triples unità base di organizzazione delle informazioni 1 item directed (labeled) graphs set di triple;

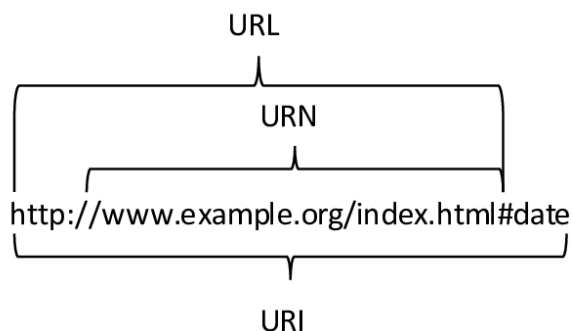
→ URI.

RDF è un linguaggio general-purpose per rappresentare fatti nel web. Ha una sintassi **XML** e tutto viene rappresentato come una **risorsa**.

100) URI

Un **URI** è un **uniform resource identifier**, una stringa di caratteri usati per identificare un nome o una risorsa.

Si distinguono in **URL**, **uniform resource locator**, per identificatori che sono locatori (come le risorse web) e in **URN**, **uniform resource names**, usati per identificare risorse indipendenti da locazioni e persistenti.



101) Caratteristiche RDF

L'**RDF** ha le seguenti caratteristiche:

- **indipendenza**: dal momento che i predicati sono risorse, qualsiasi organizzazione indipendente può definirle;
- **interscambiabilità**: le risorse RDF possono essere convertite in XML, permettendone facilmente lo scambio;
- **scalabilità**: essendo formato da semplici unità base, sono facilmente componibili e scalabili;
- **manipolabilità**: le risorse hanno le proprie caratteristiche e possono essere manipolate;
- **risorse**: tutto può essere una risorsa, anche soggetti e oggetti.

Capitolo 9 - Cloud Computing

102) Cloud Computing: definizione e caratteristiche

Il **cloud computing** è uno stile di computazione che fornisce un insieme di capacità "**as a service**" scalabili ed elastiche a degli utilizzatori esterni, attraverso le tecnologie internet. In altre parole, le applicazioni, i dati e le risorse sono fornite come servizi all'utilizzatore finale.

103) Cloud: definizione e caratteristiche

Il **cloud** è un modello che fornisce un "**network access**" a un pool condiviso di

risorse che possono essere rapidamente strutturate e rilasciate con un dispendio minimo in termini di gestione e interazione con dei provider di servizi.

104) Cloud: caratteristiche essenziali

Le 5 caratteristiche sono (non sono tradotte per mantenerne il significato originale):

- **on-demand self-service**: un utente può utilizzare la potenza di calcolo senza la necessità di interagire (umanamente) con un cloud provider;
- **broad network access**: sono disponibili su internet;
- **resource pooling**: possono essere serviti più client contemporaneamente su diversi servizi;
- **rapid elasticity**: sono scalabili e, spesso, all'utilizzatore finale sembra di poter utilizzare il sistema senza limitazioni;
- **measured service**: le risorse sono monitorate e controllate, con un lavoro di ottimizzazione da parte del cloud.

105) Cloud: quali sono i modelli di servizi?

I modelli di servizi sono:

- **SAAS**: software as a service;
- **PAAS**: platform as a service;
- **IAAS**: infrastructure as a service.

106) Differenza tra Cloud pubblico e privato

Nel **cloud pubblico** l'infrastruttura cloud è resa disponibile ad un pubblico generico o a un largo gruppo di utenti ed è di proprietà di un'organizzazione che vende servizi in cloud, mentre nel **cloud privato** l'infrastruttura cloud è dedicata ad un'organizzazione privata.

107) Virtualizzazione: caratteristiche, funzioni principali e cosa consente di fare?

La funzione principale della **virtualizzazione** è l'abilità di eseguire diversi sistemi operativi e applicazioni concorrenti, indipendentemente dalla piattaforma hardware e software.

La **virtualizzazione** consente di:

- isolare i fallimenti o i problemi di sicurezza;
- introduzione di nuove capacità, senza aggiungere complessità a sistemi già complessi;

→ può eventualmente migliorare le performance grazie al bilanciamento delle risorse, utilizzando solo ciò che è necessario.

108) Quali sono i livelli di interfaccia

I livelli sono, dal più basso al più alto:

- 1 - **hardware istruzioni macchina privilegiate**: un'interfaccia per l'hardware che è disponibile per qualsiasi programma;
- 2 - **operating system istruzioni privilegiate**: fornisce un'interfaccia all'hardware per il sistema operativo;
- 3 - **system calls**: fornisce un'interfaccia al sistema operativo per le applicazioni;
- 4 - **API**: un'interfaccia OS implementata con function calls.

109) Due tipi di virtualizzazione

Vi sono due tipologie di virtualizzazione:

- **Process Virtual Machine**: Virtualizzazione attraverso interpretazione ed emulazione. Implementato eventualmente per un solo processo.
- **Virtual Machine Monitor**: Capacità di fornire una virtual machine to differenti programmi contemporaneamente, come se ci fossero molteplici **CPU** che lavorano sulla stessa piattaforma.

110) Microservizio: cosa si intende?

I **microservizi** sono un'architettura software che suddivide un'applicazione monolitica in una serie di servizi piccoli, indipendenti e distribuiti, che comunicano tra loro attraverso **API** ben definite. Ogni **microservizio** è focalizzato su una singola funzionalità o su un piccolo insieme di funzionalità correlate, ed è sviluppato, distribuito e scalato in modo indipendente.

111) Applicazione monolitica del microservizio: perché è definita monolitica?

Si dice **applicazione monolitica** perché tale applicazione risulta essere impacchettata e distribuita come un **monolite**.

112) Architettura dei microservizi: descrivere brevemente l'architettura

L'**architettura del microservizio** è fondata sull'idea della suddivisione dell'applicazione in una collezione di piccoli servizi interconnessi. Un **servizio** è tipicamente implementato come una collezione di singole e distinte funzionalità: ogni microservizio è in tutto e per tutto una mini - applicazione. La comunicazione è affidata ad un **API Gateway** che fa da intermediario e bilancia il carico, gestisce la cache, controlla gli accessi e monitora il tutto.

113) Benefici dei microservizi

I benefici dei **microservizi** sono:

- contrastano il problema della complessità, in quanto i singoli servizi sono più veloci da sviluppare e sono più facili da comprendere e mantenere;
- ogni **servizio** può essere sviluppato da un team indipendente che si incentra su un singolo servizio e che può effettuare scelte indipendenti senza invalidare l'intero software;
- ogni **microservizio** può essere distribuito indipendentemente dagli altri, ovvero in base alle richieste possono essere scalati in maniera diversa.

114) Dimensioni e scalabilità

Si hanno le seguenti dimensioni di **scalabilità**:

- **X - axis**: esegue molteplici copie di una stessa applicazione con un **load balancer**
 - **Y - axis**: suddivide l'applicazione in differenti servizi multipli
 - **Z - axis**: ogni server è responsabile per un solo subset di dati
- L'architettura a microservizi opera sull'asse **Y**.

115) Container: descrivere le caratteristiche

I **container** sono unità più leggere e non legate ad un'infrastruttura specifica in cui viene impacchettata un'applicazione e tutte le sue dipendenze in modo che possano essere spostati di ambiente e eseguire senza cambiamenti. Sono potenzialmente più robuste, sicure e performanti.

116) Docker: definizione e tipologie

Docker è una piattaforma aperta per costruire applicazioni distribuite.

Alla base di **docker** ci sono:

- **image**: un read-only snapshot di un container che può essere usato come template per costruire un container;
- **container**: l'unità standard in cui le applicazioni risiedono e sono "trasportate";
- **docker hub/registry**: immagazzina, distribuisce e condivide immagini per container
- **docker engine**: un programma che crea e esegue container, può eseguire su qualsiasi piattaforma e la comunicazione avviene attraverso l'esecuzione di comandi.

117) Docker: per cosa vengono utilizzati

Vengono utilizzati per:

- rendere l'ambiente di sviluppo più veloce e leggero;
- eseguire servizi stand-alone;
- creare istanze isolate per eseguire dei test;
- costruire e testare applicazioni complesse e architetture sulla localhost prima di addentrarsi in un ambiente di produzione;
- fornire un'ambiente sandbox stand-alone e leggero per sviluppare e testare.

118) Docker Engine

Il **Docker Engine** è un'applicazione **client - server** con un **server** che è un tipo di long-running program chiamato processo demone, una REST API che specifica interfacce che il programma usa per dialogare con il demone e l'infrastruttura e dei comandi **CLI client**.

119) Docker Image

Il **Docker Image** è costituito da dei **filesystems** impilati uno sopra l'altro. Alla base c'è un **boot filesystem**, che riproduce il tipico boot filesystem di **Linux/Unix**. Utilizza la tecnica dello **union mount** per permettere a diversi filesystems di essere montati insieme ed apparire come un unico filesystem.

120) Architettura Docker

Docker utilizza un'architettura **client - server**. Il client Docker e il demone possono eseguire sullo stesso sistema, oppure puoi connettere n client ad un demone Docker remoto. Il cliente Docker e il demone comunicano usando delle **REST API**, costruiti su **socket UNIX** o un'interfaccia di rete. Un registro docker gestisce le immagini docker, **Docker Hub** e **Docker Cloud** sono registri pubblici.