Applicazioni Web e servizi

Definizione di Applicazione Web

Una applicazione web indica genericamente tutte le applicazioni distribuite ovvero applicazioni accessibili/fruibili via web per mezzo di un network, cioè in una architettura tipica di tipo client-server, offrendo determinati servizi all'utente client.

Perché Web? Ha molti lati positivi.

1. **Basato su Internet**
   - Ambiente standard (alla base c'è TCP/IP)
   - Ampia diffusione, conoscenza diffusa
2. **Semplicità e uniformità dell'interazione**
   - Interfaccia grafica (Browsers)
   - Interazione attraverso un protocollo standard (HTTP)
     Definizione di una API - Application Programming Interface
3. **Infrastruttura matura ed estesa**
   - Supporta sistemi aperti (connessione dinamica di nuovi componenti)
   - Strumenti sempre più completi e potenti: evoluzioni di HTML, Ajax, Web App etc.

5.1 Protocollo di comunicazione

Tutte le applicazioni Web adottano il protocollo HTTP.

5.1.1 Protocollo HTTP

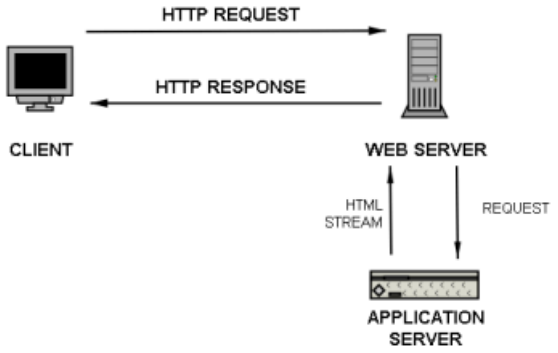
- Formato a caratteri (lento): occorre tradurre e ritradurre i dati
- Header (per metadati) e Body (corpo del messaggio)
- Utilizzo del linguaggio HTML per input e output:
  → Uso di FORM per l'acquisizione dati (invio dati al server)
  → Uso di documenti HTML in risposta (dal server verso il client)
  → Possibilità di pagine dinamiche (JavaScript con dati scambiati in JSON)
- Utilizzo di payload di tipo MIME (Multimedia Internet Mail Extensions)
  Per generalizzare oltre HTML (xml, json, zip, jpeg)
- Conversazioni (interazioni client-server) prive di stato (memoria): Ogni richiesta è un messaggio autonomo, indipendente dagli altri. Per creare sessioni di lavoro (legare più richieste tra loro) servono informazioni esplicite (Cookie, Campi nascosti)

Cookie HTTP

Un cookie HTTP (web cookie, browser cookie) è un piccolo blocco di dati che un server invia al browser web di un utente. Il browser può memorizzare il cookie e rinviarlo allo stesso server con richieste successive. In genere, un cookie viene utilizzato per stabilire se due richieste provengono dallo stesso browser, ad esempio per mantenere un utente loggato.

5.1.2 Esecuzione di un'applicazione

Il Web ha sempre supportato l'interazione tra client-server via HTTP. Con il tempo si è evoluto, andando al di là del trasferimento di pagine web ed è diventato uno strumento per fornire accesso ad applicazioni remote. Per eseguire una applicazione, il Web Server utilizza un ***Application Server,*** il quale si caratterizza dal protocollo di interazione con il Web Server, mentre l'interazione con il client avviene sempre tramite HTTP.



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. L'eseguibile può essere scritto in un qualsiasi linguaggio che supporti l'interazione con il web server (tra i linguaggi più diffusi 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. Si perde in velocità di esecuzione, ma si guadagna in facilità di scrittura dei programmi (tra i linguaggi più diffusi Java, PHP, Python e Perl, più recentemente Nodejs).

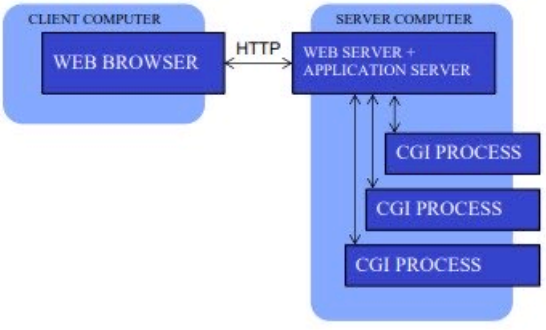
5.1.3 Architettura di un'applicazione web compilata

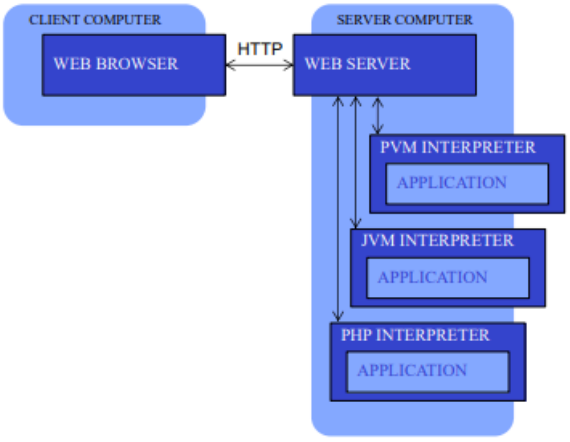
- **Uniform Resource Locator (URL)**: definisce un naming globale.
- **HyperText Transfer Protocol (HTTP)**: permette di invocare i programmi sul server come fossero risorse.

Questi due sono validi anche per applicazioni interpretate.

- **Common Gateway Interface (CGI)**: protocollo che permette al server di attivare un programma (crea un processo), passargli le richieste e i parametri provenienti dal client e recuperare la risposta.

Ogni applicazione CGI deve quindi implementare l'interprete del protocollo e verrà gestito solamente da esso e non più dall'applicazione.





Vantaggi

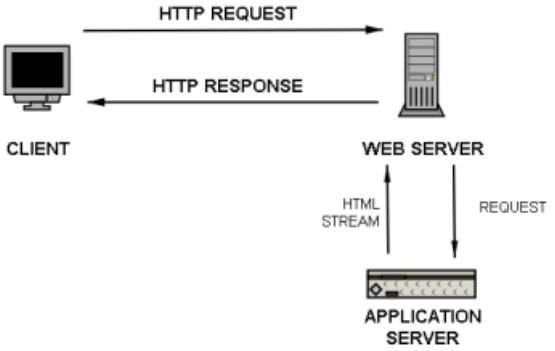
- Serve programmare solo le logiche delle applicazioni;
- Modello delle applicazioni conformi al modello del linguaggio utilizzato;
- Semplicità, portabilità, manutenibilità.

Esempi: Python, Java/Servlet, PHP, Nodejs.

> **Come funziona nella realtà**
>
> Quando un client richiede al server un URL corrispondente a un documento HTML, il server restituisce il documento stesso come un file di testo. Ciò significa che il documento viene generato una volta per tutte e poi viene inviato al client senza ulteriori elaborazioni. Quando l'URL richiesto corrisponde a un'applicazione CGI, il server esegue il programma in tempo reale, generando dinamicamente la risposta per l'utente.

5.2 Client side - HTML

Affrontiamo l'argomento "HTTP Request".



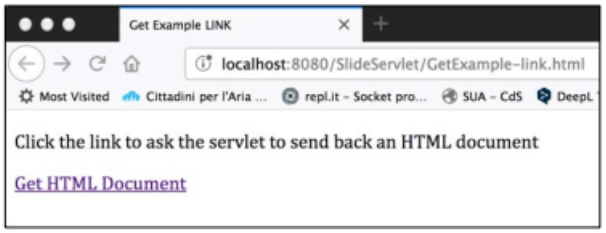
5.2.1 Richieste basate su link

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

```
<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>
```

In questo modo, il browser invia una richiesta come

```
GET /SlideServlet/GetHTTPServlet HTTP/1.1
```



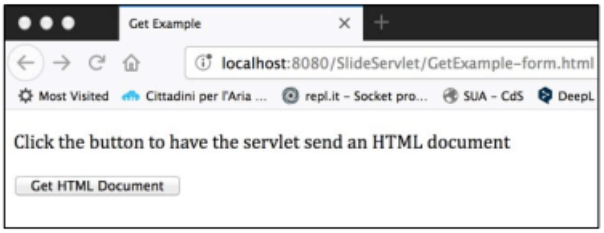
5.2.2 Richieste per mezzo di Form

1. Il parametro *action* di un Form può essere usato per puntare ad una risorsa remota.

```
<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>
```

In questo modo, il browser invia una richiesta come

```
GET /SlideServlet/GetHTTPServlet HTTP/1.1
```

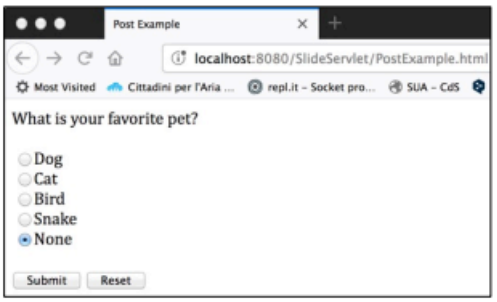


2. Un FORM può essere usato per mandare dati al server (chiedendoli all'utente).

```
# FORM con 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>
        <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>
```

In questo modo, il browser invia una richiesta come

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

Payload
animal=none
```




```
# FORM con GET
<form action="http://localhost:8080/SlideServlet/PostHTTPServlet" method="GET">
        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>
```

In questo modo, il browser invia una richiesta come

```
GET /SlideServlet/GetPostHTTPServlet?animal=none HTTP/1.1
animal=none è una Query parameter
```

5.3 Server side - Java Servlet

> **Definizione di Java Servlet**
>
> Sono piccole applicazioni Java residenti sul server (application server). Una servlet è un **componente** gestito in modo automatico da un **container** (detto anche **engine**), il quale deve e implementare un'interfaccia prestabilita che definisce il set di metodi.

Vantaggi: semplicità e standardizzazione.
Svantaggi: rigidità del modello.

Il container controlla le servlet (le attiva/disattiva) in base alle richieste dei client. Questo è possibile in maniera automatica perché le servlet implementano una interfaccia nota al server.

Le servlet sono oggetti Java residenti in memoria. Il codice dei metodi delle servlet viene eseguito da thread creati e gestiti dalla application server e le servlet possono interagire con altre servlet.

Ricordiamoci che HTTP non prevede persistenza (stateless), ovvero non si possono mantenere informazioni tra un messaggio e i successivi e tantomeno si possono identificare i client.
Mantenere lo stato della conversazione è compito dell'applicazione (se vuole) mediante:

- **Cookies**
  - Informazioni memorizzate a livello di client;
  - Permettono di gestire sessioni di lavoro.
- **HTTPSession**
  - Oggetto gestito automaticamente dal container/engine (con cookie o riscrittura delle URL);
  - Le servlet possono accedervi per immagazzinare informazioni.

5.3.1 Interfaccia Servlet

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

1. `void init(ServletConfig config)`
   Inizializza la servlet, viene invocato dopo la creazione della stessa
2. `void destroy()`
   Chiamata quando la servlet termina (es: per chiudere un file o una connessione con un database)
3. `void service(ServletRequest request, ServletResponse response)`
   Invocato per gestire le richieste dei client
4. `ServletConfig getServletConfig()`
   Restituisce i parametri di inizializzazione e il ServletContext che da accesso all'ambiente
5. `String getServletInfo()`
   Restituisce informazioni tipo autore e versione

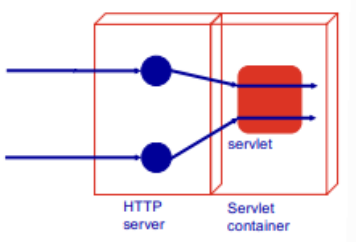
5.3.2 Richieste e risposte

Anche i parametri sono stati adattati al protocollo HTTP, cioè consentono di ricevere (inviare) messaggi HTTP leggendo (scrivendo) i dati nell'header e nel body di un messaggio.
Esistono due interfacce:

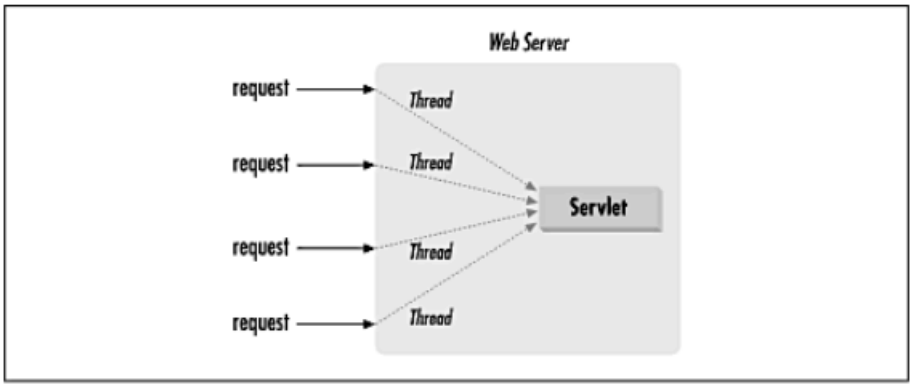
1. Interfaccia `HTTPServletRequest`: viene passato un oggetto da service e contiene la richiesta del client. Estende `ServletRequest`.
2. Interfaccia `HTTPServletResponse`: viene passato un oggetto da service e contiene la risposta destinata al client. Estende `ServletResponse`.

5.3.3 Ciclo di vita di una servlet

1. Una servlet viene creata dal **container/engine**:
   - Quando viene effettuata la prima chiamata;
   - La servlet viene condivisa da tutti client;
   - Ogni richiesta genera un Thread che esegue la doXXX appropriata;
2. Il container/engine invoca il metodo **init()** per inizializzazioni specifiche;
3. Una servlet viene distrutta dall'engine all'occorrenza di uno dei due eventi:
   1. Quando non ci sono thread in esecuzione su quella servlet;
   2. Quando è scaduto un timeout predefinito.
4. Viene invocato il metodo **destroy()** per terminare correttamente la servlet.



5.3.4 Servlet e Thread



- Molti thread -> una singola istanza della servlet.
- Più richieste vengono servite dalla stessa servlet, questo vuol dire che bisogna far attenzione a come la servlet implementa l'accesso alle sue risorse.
- In questo modello spesso le risorse sono condivise a livello di database.

Questa scelta ha diverse motivazioni:

- Utilizzare meno memoria (un solo oggetto);
- Ridurre il costo di gestione (per esempio tempi di inizializzazione inizializzazione) di molti oggetti che sarebbero spesso identici;
- Abilita la persistenza (in memoria) di risorse condivisibili tra diverse richieste (e.g. una connessione a DB, oggetto "carrello").

5.3.5 Terminazione di un servlet

Container e richieste dei client devono sincronizzarsi sulla terminazione. Alla scadenza del timeout potrebbe essere ancora dei thread in esecuzione in service().

Si deve:

1. Tener traccia dei thread in esecuzione;
2. Progettare il metodo destroy() in modo da notificare lo shutdown e attendere il completamento del metodo service();
3. Progettare i metodi lunghi in modo che verifichino periodicamente se è in corso uno shutdown e comportarsi di conseguenza.

5.4 Server Side - JSP

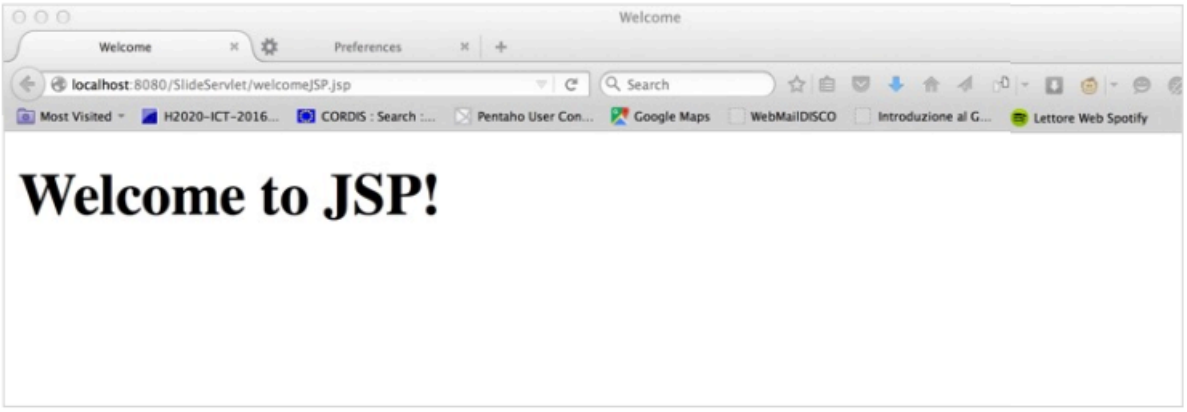
Java Server Pages, tecnologia per la creazione di applicazioni web. Specifica l'interazione tra un contenitore/server ed un insieme di "**pagine**" che presentano informazioni all'utente (viste).
Le **pagine** sono costituite da tag tradizionali (HTML, XML, WML, ...) e da *tag applicativi* che controllano la generazione del contenuto (generazione server-side).

JSP, rispetto alle servlet, facilita la separazione tra logica applicativa e presentazione.

Java Server Pages (JSP) separano la parte dinamica delle pagine dal template HTML statico. Il codice JSP va incluso in tag delimitati da "<%" e "%>".
La pagina viene convertita automaticamente in una servlet java la prima volta che viene richiesta.

Esempio di JSP

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Welcome</title>
</head>
<body>
        <h1> <%= "Welcome to JSP!" %> <h1>
</body>
</html>
```



5.4.1 Elementi di una JSP

Elemento	Caratteristiche
Template text	Le parti statiche della pagina HTML
Commento	
Direttiva	Le 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.
Azione	Le azioni permettono di supportare diversi comportamenti della pagina JSP. Vengono processati ad ogni invocazione della pagina JSP. Permettono di trasferire il controllo da una JSP all'altra, di interagire con i Java Data Beans, ecc.
Elemento di scripting	Istruzioni nel linguaggio specificato nelle direttive. Sono di tre tipi: scriplet, declaration, expression.

5.4.1.1 Direttive

page
Liste di attributi/valore;
Valgono per la pagina in cui sono inseriti

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

include
Include in compilazione pagine HTML o JSP

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

taglib
Dichiara tag definiti dall'utente implementando opportune classi

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

5.4.1.2 Azioni

forward
Determina l'invio della richiesta corrente, eventualmente aggiornata con ulteriori parametri, alla JSP indicata

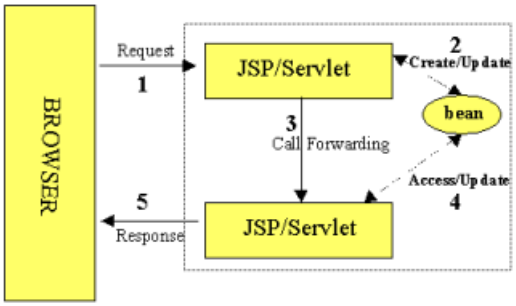
```
<jsp:forward page="login.jsp" >
         <jsp:param name="username" value="user" />
         <jsp:param name="password" value="pass" />
</jsp:forward>
```

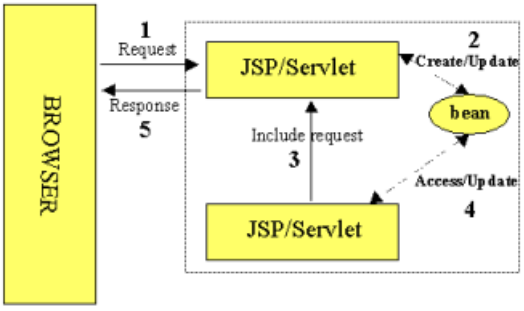
include
Invia dinamicamente la richiesta ad una data URL e ne include il risultato

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

useBean
Localizza ed istanzia (se necessario) un javaBean nel contesto specificato. Il contesto può essere la pagina, la richiesta, la sessione, l'applicazione.

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





5.4.1.3 Elementi di scripting

Declaration `<%! declaration [declaration] ... %>`
Variabili o metodi statici 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 § Tale codice diventerà parte dei metodi doGet (doPost) della servlet che viene associata la JSP

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

Linguaggio di script ha lo scopo di interagire con oggetti java e altre servlet e gestire le eccezioni java.

Può utilizzare anche oggetti impliciti (sono 9) che non devono essere creati

- request
- response
- out
- page
- pageContext
- session
- application
- config

- exception

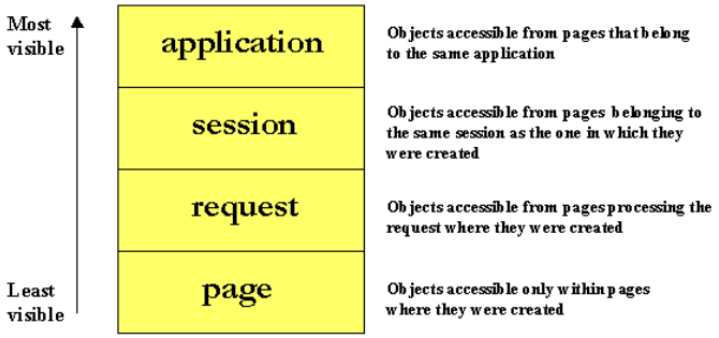
```
Grazie per la scelta di <i> <%= request.getParameter("title") %> </i>
```

5.4.1.4 Oggetti

Gli oggetti possono essere creati

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

Gli oggetti hanno un attributo che ne definisce lo “scope”.



5.4.1.5 JavaBean

Un javabean è una classe che segue regole precise (specifica).

1. Deve avere costruttore senza parametri;
2. Dovrebbe avere campi (property) private;
3. I metodi di accesso ai campi (property) sono set/get/is
   setXxx
   getXxx/isXxx
   con xxx = property

```
class Book{
        private String title;
        private boolean available;
        void setTitle(String t) ...;
        String getTitle() ...;
        void setAvailable(boolean b) ...;
        boolean isAvailable () ...;
}
```

Utilizzo di un bean

Accedere ad un bean (inizializzazione)

```
<jsp:useBean id="Attore" class="MyThread" scope="session" type="Thread"/
```

Scope: determina la vita e visibilità del bean.

- *page*: è lo scope di default, viene messo in pageContext ed acceduto con getAttribute;
- *request*: viene messo in ServletRequest ed acceduto con getAttribute;

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

Type: permette di assegnargli una superclasse o un'interfaccia.

Al posto della classe si può usare il nome del bean § beanName="nome" dove nome è la classe o un file serializzato.

```
<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())); //etc.. %>
</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 %>" />
```

Esempio

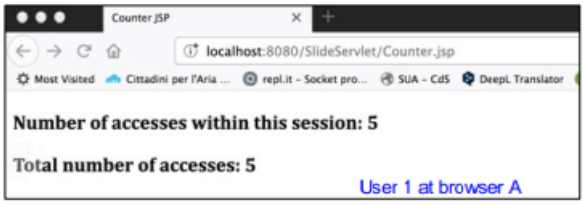
CounterBean.java

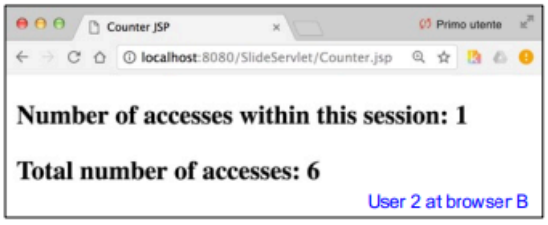
```
public class CounterBean {
        //declare a integer for the counter
        private int count;

        public int getCount() {
                //return count
                return count;
        }

        public void increaseCount() {
                //increment count
                count++;
        }
}
```

Counter.jsp

```
<%@ page import="itis.mvc.CounterBean"%>
        <jsp:useBean id="session_counter" class="itis.mvc.CounterBean" scope="session" />
        <jsp:useBean id="app_counter" class="itis.mvc.CounterBean" scope="application" />
        <%
        session_counter.increaseCount();
        synchronized (page) {
                app_counter.increaseCount();
        }
        %>
        <h3>
                Number of accesses within this session:
                <jsp:getProperty name="session_counter" property="count" />
        </h3>
        <p></p>
        <h3>
                Total number of accesses:
                <% synchronized (page) { %>
                <jsp:getProperty name="app_counter" property="count" />
```

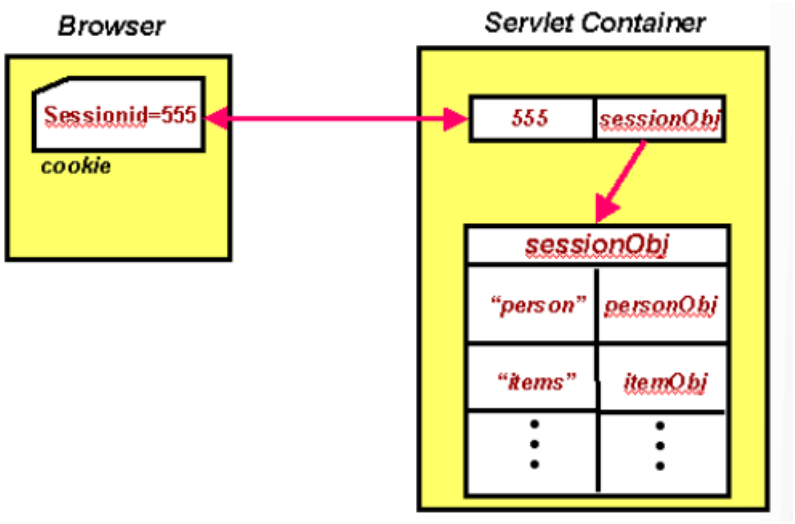
```
                <% } %>
            </h3>
```





5.4.1.6 Sessioni

Accede ad un oggetto HTTPSession (session). Funziona come mappa chiave-valore.



Esempi

- *Memorizzazione*
  `<% Foo foo = new Foo(); session.putValue("foo",foo); %>`
- *Recupero*
  `<% Foo myFoo = (Foo) session.getValue("foo"); %>`
- *Esclusione di una pagina dalla sessione*
  `<%@ page session="false" %>`

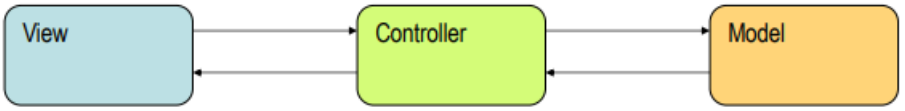
5.5 Pattern MVC - Model View Control

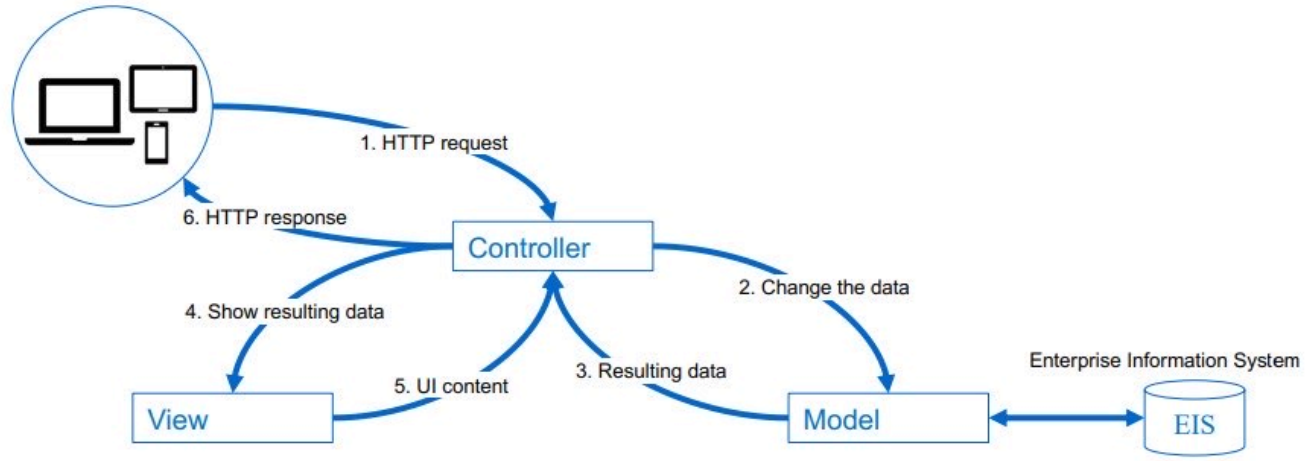
Il **Model-View-Controller (MVC)** è un pattern architetturale che separa data model, user interface, e control logic in tre componenti distinte.

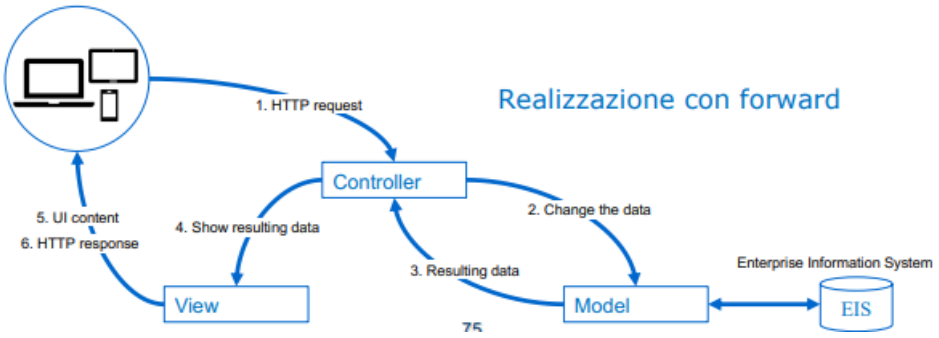
Il pattern **Model View Controller (MVC)** ha lo scopo di separare

- I dati (gli oggetti) trattati dall'applicazione e i metodi principali per la loro manipolazione (*Model*);

- La struttura dei dati restituiti al richiedente (Esempio: pagina HTML/CSS) (*View*);
- Il coordinamento dell'interazione tra interfaccia (*azioni degli utenti*) e i dati (*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.







5.5.1 Vantaggi e svantaggi

Vantaggi

- ***Chiara separazione tra logica di business e logica di presentazione***
  - Poter cambiare la view senza modificare il control e viceversa
  - Poter arricchire la view senza appesantire il codice del controller
  - Poter definire e scegliere a run-time la view da usare a seconda dell'interazione, dei device utilizzati, dello stato dei dati o delle preferenze dell'utente
- ***Chiara separazione tra logica di business e modello dei dati***
  Poter definire diversi mapping tra le azioni degli utenti sul controller e le funzioni sul model
- ***Ogni componente ha una responsabilità ben definita***
  - Poter sviluppare in parallelo
  - Poter manutenere e far evolvere ogni componente in modo indipendente, quindi semplificando la gestione
- ***Ogni parte può essere affidata a esperti***
  - Poter assegnare lo sviluppo della view ad esperti di interfaccia e interazione (e.g., pagine jsp o asp)
  - Uso tecnologie adatte allo sviluppo delle singole componenti

Svantaggi

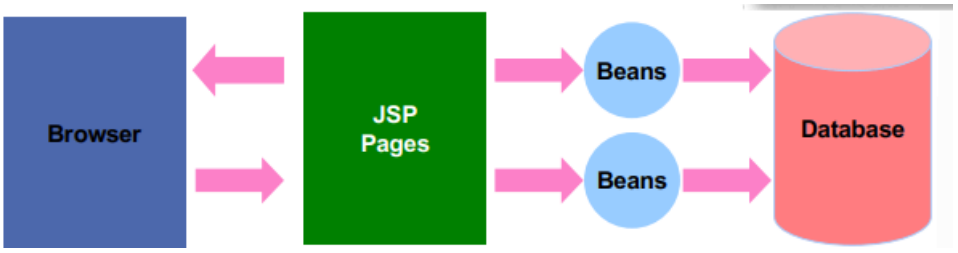
- Aumento della complessità dovuta alla concorrenza (è un sistema distribuito)

- Inefficienza nel passaggio dei dati alla view (un elemento in più tra cliente e controller)

5.5.2 Pattern Model 1

Scopo: separazione tra dati, logica di business e visualizzazione.

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



Svantaggio:

1. Il controllo della navigazione è decentralizzato, poiché ogni pagina contiene la logica per determinare la pagina successiva
2. Difficile da mantenere e da estendere.
3. Per evitare che la JSP sia piena di codice java bisogna creare molti tag ad hoc

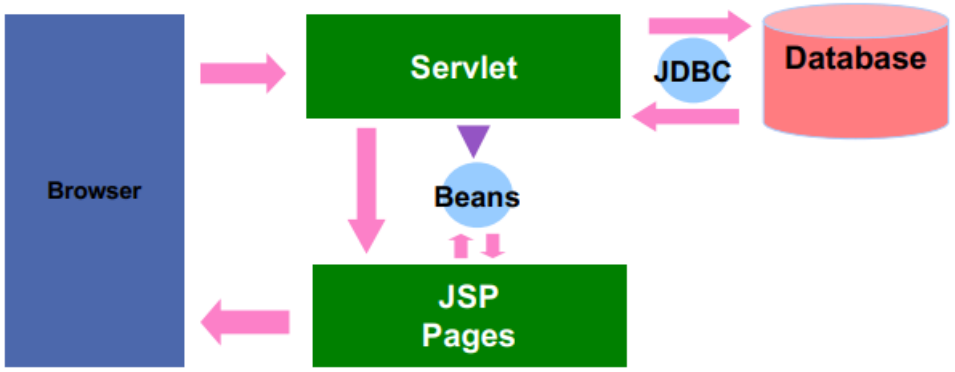
5.5.3 Pattern Model 2

La richiesta viene inviata ad una **Java Servlet** che:

1. genera i dati dinamici richiesti dall'utente
2. li mette a disposizione della pagina jsp come Java Beans

La **Servlet** richiama una **pagina .jsp**, che:

1. legge i dati dai beans
2. organizza la presentazione in HTML che invia all'utente

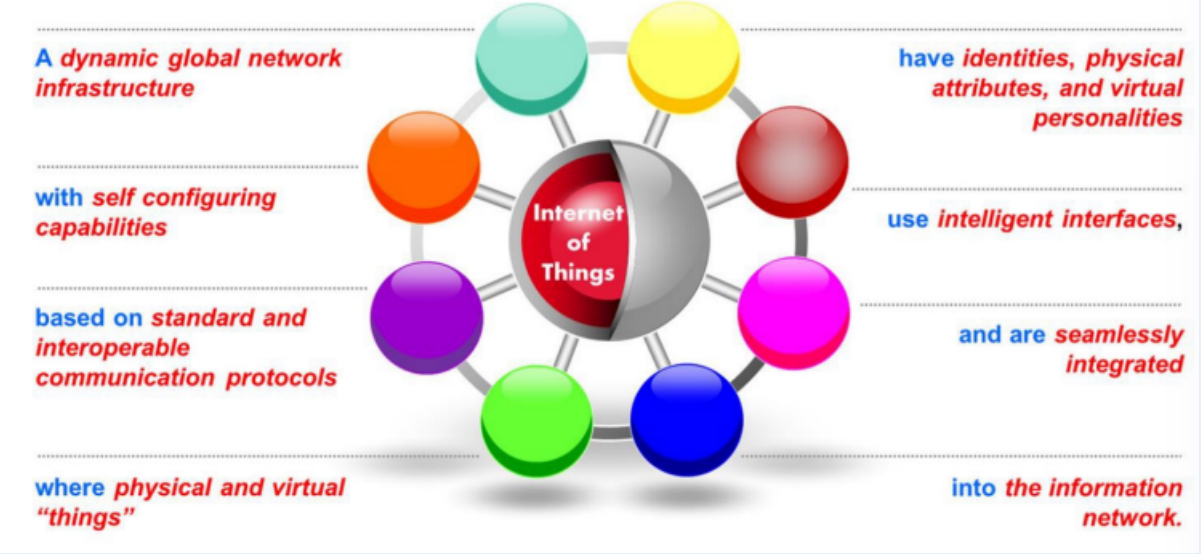


5.6 Service computing

IoT

L'Internet of Things (Internet delle cose) è un sistema di oggetti fisici che possono essere scoperti, monitorati, controllati o interagire con altri dispositivi elettronici, che comunicano attraverso varie interfacce di rete e possono essere collegati ad Internet.

IoT significa **comunicazione** e **connettività**.



I dispositivi elettronici sono in grado di raccogliere e scambiare dati, ed eseguire azioni per realizzare le piattaforme del futuro. Ogni elemento consuma/espone delle funzionalità (servizi), costituito da un numero elevato di componenti coinvolti in singole applicazioni.

Con il Services Computing, è avvenuto anche un **cambio di prospettiva**, in quanto si è reso necessario un cambiamento nella progettazione, nell'implementazione e nella manutenzione dei sistemi rispetto alle architetture distribuite classiche (applicazioni basate sul web).

Sistema distribuito tradizionale	Sistema distribuito contemporaneo
Sistema singolo sviluppato in modo ***top-down*** mediante decomposizione	Sistema tradizionale + include ***sottosistemi di terze parti***
Quasi sempre implementa il ***modello client-server***	Ogni componente software deve essere in grado di servire sistemi diversi in modi diversi
Forte accoppiamento: un'unica organizzazione mantiene e/o possiede il sistema	***Debole accoppiamento***: ogni componente viene mantenuto indipendentemente dagli altri

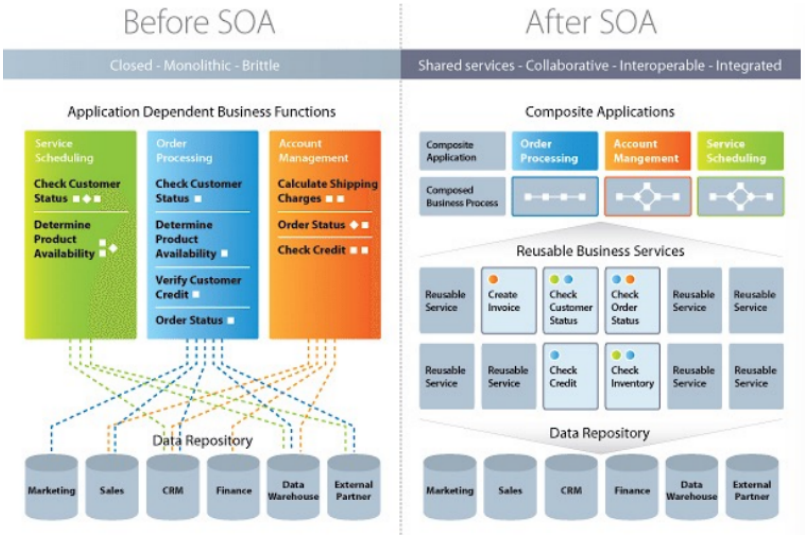
Esempio: un sistema contemporaneo come un sistema bancario non offre solo prodotti bancari ai clienti, ma offre anche contratti di assicurazione ecc.

5.6.1 Architettura orientata ai servizi - SOA

Definizione di SOA

Stile architettonico che si concentra su elementi discreti riutilizzabili (chiamati servizi), invece che su un design monolitico, per costruire le applicazioni.

È un approccio al design del software in cui i servizi software sono forniti, utilizzati e coordinati tra loro in modo indipendente. In un'architettura SOA, i servizi sono i blocchi fondamentali, i quali sono componenti autonomi che forniscono funzioni specifiche ai richiedenti (che possono essere altri servizi) e possono essere eseguiti su sistemi distribuiti su una rete. L'accesso al servizio è fornito attraverso la rete (anche Internet).



5.6.2 Elementi fondamentali di SOA

Componenti

- Servizio
- Descrizione del servizio

Ruoli

- ***Service Providers***
  Fornitori di servizi: offrono servizi/funzionalità;
- ***Service Broker***
  Cataloghi di servizi di menage: servizi o funzionalità aggiuntive fornite per supportare o facilitare la gestione e l'organizzazione dei servizi stessi all'interno di un ambiente di brokeraggio dei servizi.
  Un "service broker" è un intermediario che facilita la transazione tra fornitori di servizi e consumatori di servizi.
- ***Service Requestors***
  Richiedenti di servizi: trovano un servizio e interagiscono con i provider.

Operazioni

- ***Publish*** (un servizio);
  La pubblicazione dei servizi è il processo mediante il quale i servizi vengono resi disponibili all'interno dell'ambiente SOA in modo che possano essere scoperti e utilizzati da altri componenti o applicazioni.
- ***Find*** (servizio/endpoint);
  La ricerca dei servizi è il processo mediante il quale i client o altri componenti all'interno dell'ambiente SOA individuano i servizi disponibili per l'utilizzo.
- ***Interact*** (ad esempio, richiesta-risposta).
  L'interazione con i servizi è il processo mediante il quale i client utilizzano effettivamente i servizi trovati per eseguire operazioni specifiche.
  Cosa fa: Una volta individuati i servizi desiderati, i client possono interagire con essi inviando richieste e ricevendo risposte in base alle operazioni supportate dal servizio.

5.6.3 Servizio

Servizio Web

Definizione informale: Un servizio è un'entità software indipendente che può essere scoperta e invocata da altri sistemi software attraverso una rete.

Definizione formale: Un servizio Web è un'applicazione software

1. Identificata da un URI (Uniform Resource Identifier);
2. Le cui interfacce sono in grado di essere definite, descritte e scoperte;
3. Supporta interazioni dirette con altri software per mezzo di messaggi e protocolli basati su Internet.

Un servizio web utilizza le tecnologie web (in particolare l'HTTP) per fornire funzionalità.

Altre definizioni:

1. I servizi web sono autonomi, autodescrittivi, applicazioni modulari che possono essere pubblicate, localizzate e invocate attraverso il Web.
2. I servizi web sono incapsulati, "componenti" Web liberamente accoppiati che possono legarsi dinamicamente l'uno all'altro.

Caratteristiche dei servizi
I servizi sono "componenti" indipendenti

1. **Interfaccia nota**
   Viene utilizzato un linguaggio di descrizione **standard** (ad esempio, WSDL). Inoltre, è possibile averne una gestione automatica da parte del **middleware**.
   Attenzione: Il termine middleware viene utilizzato soprattutto per il software che consente la comunicazione e la gestione dei dati nelle applicazioni distribuite.
2. **Punto di accesso unico**
   Uso di ***URI*** (URL/URN), scopribile tramite servizi di nomi (ad esempio, directory UDDI).
   Attenzione: UDDI è uno standard basato su XML per descrivere, pubblicare e trovare servizi web. UDDI è l'acronimo di Universal Description, Discovery, and Integration.
3. **Scambio di dati basato su documenti**
   Utilizzo di un formato di rappresentazione standard (ad esempio, XML, JSON).

Ogni servizio deve:

1. Fornire una descrizione delle sue funzionalità da scoprire e selezionare;
2. Fornire l'accesso alle sue funzionalità attraverso protocolli di rete noti;
3. Supporto alla composizione con altri servizi per fornire soluzioni complesse;
4. Rispondere alle esigenze aziendali dei clienti e ai requisiti del dominio (funzionali e non funzionali);
5. Garantire un livello di qualità di servizio (QoS).

5.6.3.1 Accordo sul livello di servizio (SLA)

Lo SLA, Service Level Agreement, è un contratto tra il provider e l'utente.

1. Assicura che la funzionalità sia fornita correttamente;
2. Definisce le caratteristiche non funzionali garantite dal servizio.

Un SLA comprende diversi SLO (Service Level Objectives) che definiscono la qualità del servizio da garantire attraverso metriche specifiche.

Service Level Objectives

Un SLO è un obiettivo specifico e misurabile che definisce il livello di servizio che un fornitore si impegna a offrire ai suoi clienti. Esso delinea le metriche di qualità e performance che devono essere mantenute per soddisfare le aspettative del destinatario del servizio.

Peculiarità dei Web Service

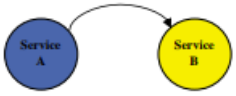
Peculiarità	Descrizione
Componenti pubblici	- Sono "scopribili" (naming, registri, accesso); - Interfacce pubbliche (protocolli standard, gestiti dalla macchina)
Componibilità	- Servizi compositi: **orchestrazione** - Coordinamento: **coreografia**
Quality of Service	- Base: sicurezza, disponibilità, prestazioni... - Context awareness: capacità di un sistema o di un'applicazione di comprendere e reagire in base al contesto circostante.
Organizzazione del sistema	- **Peer-to-Peer (P2P)**: i partecipanti possono offrire e utilizzare servizi direttamente tra loro, senza dover passare attraverso un intermediario centrale come un broker o un registro dei servizi. (decentralizzazione, scalabilità, autonomia del partecipanti, etc.). - **Enterprise Service Bus (ESB)**: piattaforma software che facilita l'integrazione di sistemi e applicazioni eterogenee all'interno di un'organizzazione. (intermediario tra diversi sistemi e applicazioni, conversione dei dati tra formati, monitoraggio, etc.). - **Grid**: architettura di rete distribuita che consente la condivisione e l'accesso a risorse computazionali, di archiviazione e di elaborazione distribuite in modo trasparente (condivisione delle risorse, interoperabilità, applicazioni distribuite etc.).

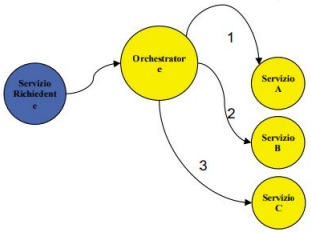
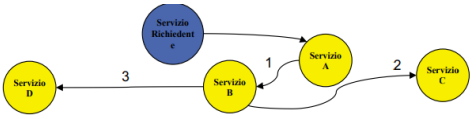
5.6.3.2 Composizione di servizi

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.

Affinché la composizione dei servizi abbia successo, è necessaria la ***compatibilità*** tra i servizi:

1. Devono parlare la **stessa lingua** (a livello sintattico e semantico);
2. Un fornitore di servizi deve presentare un'**interfaccia pubblica**.

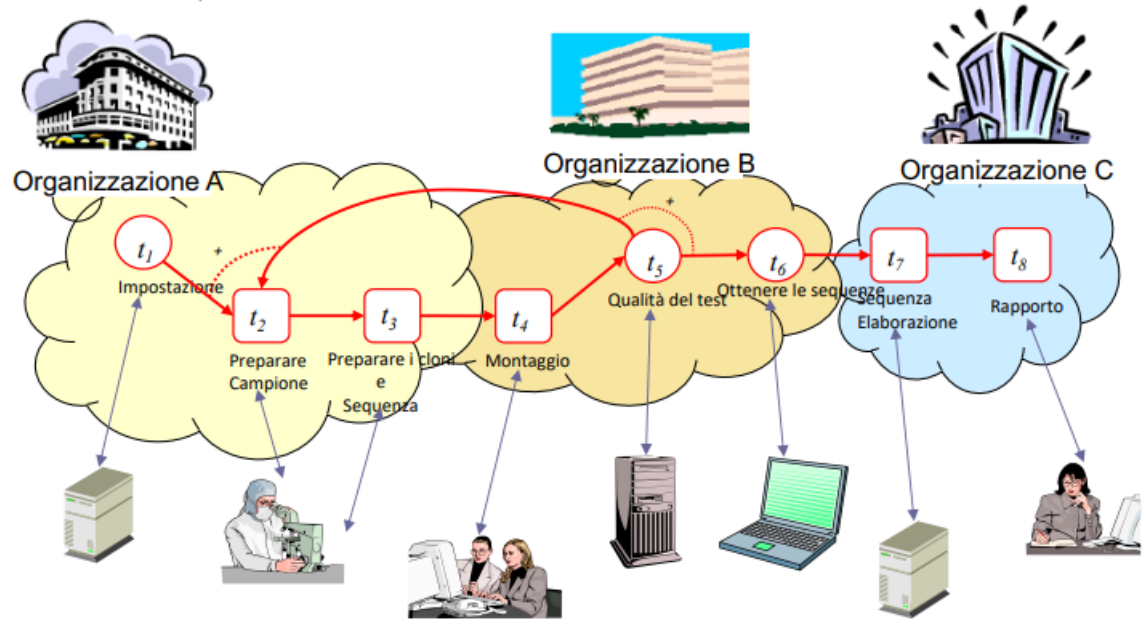


Orchestrazione	Coreografia
Descrive come i servizi interagiscono tra loro, compresa la logica di business e l'ordine di esecuzione delle interazioni dal punto di vista e sotto il controllo di un singolo attore (servizio).	Descrive la sequenza di interazioni tra più parti coinvolte nel processo dal punto di vista di tutte le parti. Definisce lo stato condiviso delle interazioni tra le entità.
Richiede un controllo attivo. È più complicato, ma più facile da monitorare.	Ogni servizio è responsabile di portare a termine il proprio compito, invocando anche altri servizi. Di solito viene implementato utilizzando eventi.
	

Business Processes

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.

Flusso di lavoro del sequenziamento del DNA - Cardoso 2004



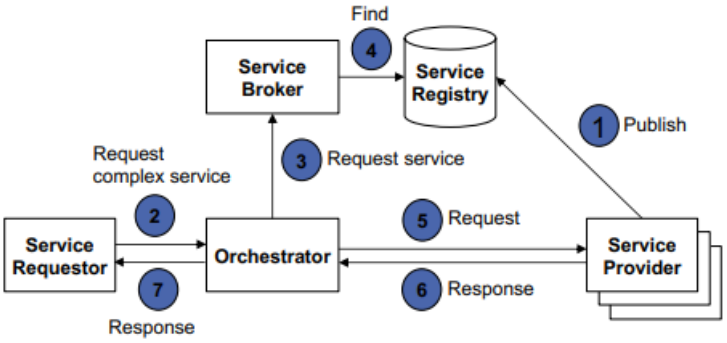
Caratteristiche

1. Può contenere ***condizioni definite*** che ne determinano l'avvio e ***uscite definite*** al suo completamento.
2. Può comportare interazioni ***formali*** o relativamente ***informali*** tra i partecipanti (umani e software).
3. Può contenere una serie di attività ***automatizzate*** e/o ***manuali***.
4. È ***distribuito*** e personalizzato al di là dei confini all'interno delle ***organizzazioni*** e ***tra di esse***, e spesso si estende a più applicazioni con piattaforme tecnologiche diverse.
5. Ha una ***durata*** che può ***variare notevolmente***.
   Di solito è di lunga durata. Una singola istanza può durare mesi o addirittura anni.

Architettura orchestrata

Struttura che si basa sul creare una rete dinamica di servizi, in cui i singoli servizi possono essere offerti da organizzazioni diverse. Un'organizzazione assume il ruolo di orchestratore e si fa carico di implementare il servizio controller per altre organizzazioni.

Broker di servizi: organizzazioni che trovano il miglior allineamento (intermediario) tra i servizi richiesti dai Service Requestors e quelli offerti dai Service providers.

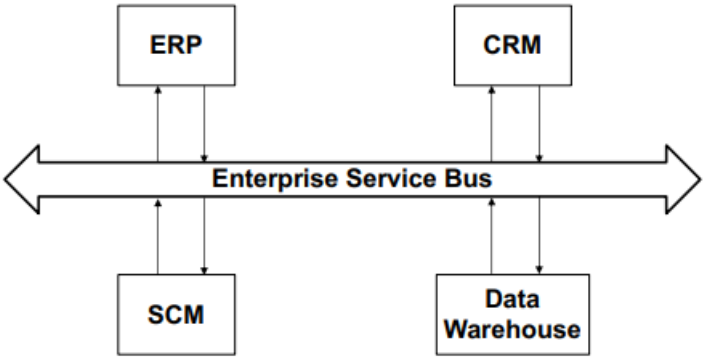


Enterprise Service Bus (ESP)

L'architettura SOA ha permesso di semplificare l'integrazione tra i diversi componenti di un sistema informativo. Ha portato alla definizione dell'Enterprise Service Bus (ESB).

> **ESP**

Sistema di comunicazione per supportare l'interazione e la comunicazione tra i componenti di un sistema informativo. È un esempio di *approccio coreografico*.
Di solito implementa il **modello publish/subscribe**.

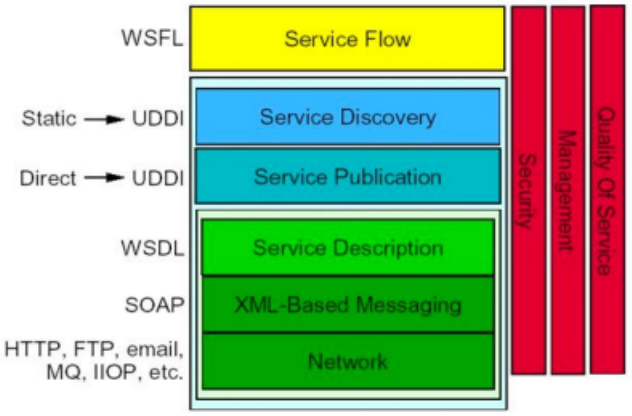


Caratteristiche principali

- **Instradamento dei messaggi tra applicazioni e servizi**
  L'ESB gestisce il routing dei messaggi, indirizzandoli ai sistemi o ai servizi appropriati in base ai contenuti o ai metadati del messaggio stesso. Questo permette una distribuzione efficiente dei messaggi tra i vari componenti del sistema.
- **Trasformazione del messaggio**
  Spesso i sistemi eterogenei utilizzano formati di dati diversi. L'ESB offre funzionalità per la trasformazione dei dati, consentendo di convertire i dati da un formato all'altro in modo che possano essere interpretati correttamente dai sistemi destinatari.
- **Comunicazione sicura**
  L'ESB offre funzionalità di sicurezza per proteggere i dati e garantire l'accesso autorizzato alle risorse. Ciò può includere autenticazione, autorizzazione, crittografia e altre misure di protezione dei dati sensibili durante il trasferimento.
- **Architettura estensibile**
  Un'architettura estensibile consente di adattare l'ESP alle esigenze specifiche dell'azienda, consentendo l'aggiunta o la rimozione di funzionalità in base alle necessità. Le aziende possono avere una varietà di sistemi esistenti e applicazioni legacy che devono essere integrati con l'ESP. Un'architettura estensibile consente di integrare facilmente questi sistemi esistenti aggiungendo nuovi componenti o adattando quelli esistenti per soddisfare i requisiti di integrazione. Inoltre, possono desiderare di integrare servizi o funzionalità forniti da terze parti nell'ESP. Un'architettura estensibile consente di integrare facilmente questi servizi esterni attraverso interfacce standard o API, consentendo una maggiore flessibilità e capacità di collaborazione.
- **Modello publish/subscribe**
  Il modello "publish/subscribe" in un ESP offre un modo efficiente e scalabile per distribuire eventi, notifiche o dati tra i vari componenti di un sistema distribuito, consentendo una comunicazione efficace e una gestione flessibile degli eventi all'interno dell'ambiente aziendale.
  È un paradigma di comunicazione in cui i servizi o le risorse vengono pubblicati da un mittente e i destinatari interessati a tali informazioni si iscrivono per riceverle.

5.6.2 Servizi SOAP

Un servizio web (SOAP) può essere descritto con il seguente stack



Strato	Linguaggio/Protocollo	Informazione
Business Process (Service Flow)	BPEL/WSFL	Linguaggio di esecuzione dei processi aziendali
Search and Find (Discovery & Publication)	UDDI	Universal Discovery Description and Integration, per i registri dei servizi Web
Description	WSDL	Web Services Description Language, per descrivere servizi in rete basati su XML
Messaggi (XML-Based Messaging)	SOAP	Simple Object Access Protocol, per definire un modo uniforme di passare dati codificati in XML
Network	Protocolli Internet (HTTP, STMP, etc.)	

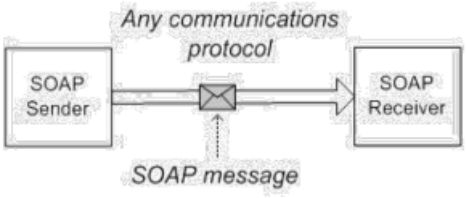
5.6.2.1 SOAP

Simple Object Access Protocol
SOAP è un protocollo basato su XML che consente ad applicazioni software di comunicare utilizzando messaggi XML.

- Gli spazi dei nomi XML sono utilizzati per fornire una semantica ai dati (non solo ai tipi di dati).
  Gli spazi dei nomi XML sono un meccanismo utilizzato per evitare ambiguità nelle etichette degli elementi e negli attributi nei documenti XML. Questo è particolarmente utile quando si combinano più documenti XML da diverse fonti o quando si lavora con vocabolari di dati complessi.
- È agnostico rispetto al sottosistema di trasporto.
  Non dipende da un particolare protocollo di rete o di trasporto per inviare i suoi messaggi. In altre parole, SOAP è progettato per essere indipendente dal mezzo attraverso il quale viene inviato un messaggio.
- Permette di creare messaggi di richiesta e risposta in diverse configurazioni

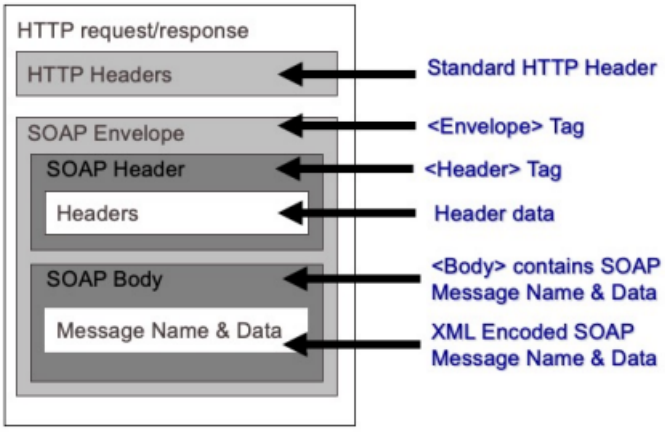
SOAP è un modo standard per strutturare i messaggi XML (dati).

- Un'applicazione delle specifiche XML
- Si affida a XML Schema, spazi dei nomi XML
- Non è legato ad alcun linguaggio o piattaforma di programmazione (nemmeno ai servizi web)
- Estensibile
- Utilizzato nei servizi orientati ai documenti



5.6.2.2 Componenti di un messaggio SOAP

1. **SOAP Envelope**
   Ingloba il contenuto del messaggio
2. **SOAP Header** (opzionale)
   - Maggiore flessibilità, può essere elaborato dai nodi tra l'origine e la destinazione.
   - Contiene blocchi di informazioni su come elaborare il messaggio:
     - Impostazioni di instradamento e consegna
     - Asserzioni di autenticazione/autorizzazione
     - Contesti di transazione
3. **SOAP Body**
   - Messaggio effettivo da consegnare ed elaborare
   - Sia per le informazioni di richiesta che per quelle di risposta



5.6.2.3 SOAP HTTP

Richiesta HTTP GET

```
GET /travel.example.org/reservations?code=F3T HTTP/1.1
Host: travelcompany.example.org
Accept: text/html;q=0.5, application/soap+xml
```

Richiesta HTTP POST

```
POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 230
<env:Envelope
        xmlns:env="http://www.w3.org/2003/05/soap-envelope">
        <env:Header>...</env:Header>
        <env:Body>...</env:Body>
</env:Envelope>
```

Risposta HTTP GET/POST

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 200
<?xml version='1.0' ?>
<env:Envelope
        xmlns:env="http://www.w3.org/2003/05/soap-envelope">
        <env:Header>...</env:Header>
        <env:Body>...</env:Body>
</env:Envelope>
```

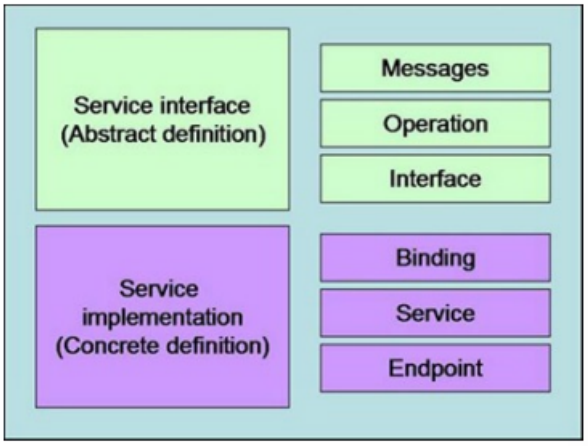
5.6.3 WSDL

WSDL, acronimo di "Web Services Description Language" (Linguaggio di Descrizione dei Servizi Web), è un linguaggio basato su XML utilizzato per descrivere i servizi web, i messaggi e le modalità di invocazione. Un file WSDL fornisce una descrizione formale dell'interfaccia pubblica di un servizio web, specificando i dettagli tecnici necessari per invocare le operazioni offerte dal servizio. Utilizzato per definire *"Come e dove accedere al servizio"*.

Cosa consente di descrivere WSDL

- Informazioni sull'**interfaccia** che descrivono tutte le operazioni pubblicamente disponibili di un servizio.
- Dichiarazioni di **tipi di dati** per tutti i messaggi. I tipi complessi possono essere dichiarati (utilizzando SOAP) e utilizzati
- Informazioni di **binding** sul protocollo di trasporto (HTTP, SMTP, UDP)
- Informazioni sull'**indirizzo** per la localizzazione del servizio (URI)

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



5.6.3.1 Binding

- L'attributo ***name*** definisce il nome del binding. Con questo nome si può fare riferimento ad esso quando si definisce un endpoint di servizio. Ogni nome di binding deve essere univoco all'interno dello spazio dei nomi del target WSDL 2.0;

- L'attributo ***interface*** contiene il nome di una delle interfacce definite;
- L'attributo ***type*** definisce il formato del messaggio da utilizzare. In questo caso è SOAP (sono possibili altri binding);
- ***wsoap:protocol*** definisce il protocollo di "trasporto". Nell'esempio, i messaggi soap verranno trasportati utilizzando HTTP.

5.6.3.2 Operation

- L'attributo ***ref*** dell'elemento operation fa riferimento a una specifica operazione (già definita nella sezione interface);
- L'attributo ***wsoap:mep*** definisce lo schema di scambio dei messaggi per SOAP (richiesta GET).

5.6.3.3 Fault

- L'attributo ***ref*** definisce quale errore (già definito nella sezione dell'interfaccia) ci si riferisce per questo binding;
- L'attributo ***wsoap:code*** dell'elemento fault definisce il codice di errore che determina l'invio di questo messaggio.

```
<binding name = "myServiceInterfaceSOAPBinding"
                interface = "tns:myServiceInterface"
                type = "http://www.w3.org/ns/wsdl/soap"
                wsoap:protocol = "http://www.w3.org/2003/05/soap/bindings/HTTP/">
        <operation ref = "tns:checkServiceStatusOp"
                                wsoap:mep = "http://www.w3.org/2003/05/soap/mep/soap-response"/>
        <fault ref = "tns:dataFault"
                        wsoap:code = "soap:Sender"/>
</binding>
```

5.6.3.4 Service

- L'attributo ***name*** definisce il nome del servizio. Ogni nome di servizio deve essere unico all'interno dello spazio dei nomi di destinazione;
- L'attributo ***interface*** definisce il nome dell'interfaccia (precedentemente definita nella sezione interface) implementata dal servizio.

5.6.3.5 Endpoint

- L'attributo ***name*** definisce il nome dell'endpoint, che deve essere unico all'interno dei servizi;
- L'attributo ***binding*** specifica quale binding già definito verrà utilizzato da questo endpoint;
- L'attributo ***address*** specifica l'indirizzo fisico dove il servizio sarà disponibile.

```
<service name = "myService" interface = "tns: myServiceInterface">
        <endpoint name = "myServiceEndpoint"
                        binding = "tns:myServiceInterfaceSOAPBinding"
                        address = "http://yoursite.com/MyService"/>
</service>
```

5.6.4 Problema di SOAP e WSDL

WSDL e SOAP sono troppo prolissi, difficili da capire, complessi.
Si è ritenuto necessario un approccio più semplice e leggibile.

- Semantica concordata per il metodo e JSON per il payload
- DNS, nomi leggibili per gli endpoint

Il nuovo approccio è proprio **REST**.

5.6.5 REST

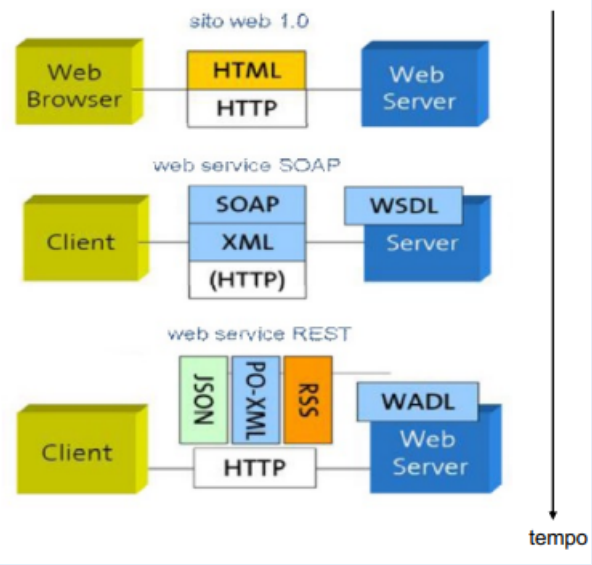
REST = REpresentational State Transfer
REST è uno stile architettonico per i sistemi distribuiti.

Evoluzione dei servizi web

Inizialmente solo HTML e HTTP e i servizi erano limitati alla fornitura di pagine web.

Con SOAP i servizi risultano fortemente caratterizzati e descritti con elevata precisione. Possono persino essere "compilati". Tuttavia, i servizi sono complessi e difficili da descrivere e comprendere.

Con REST si torna ai principi del protocollo HTTP eliminando le ridondanze e assegnando una semantica ai verbi (GET, POST, PUT...) e agli URI.



5.6.5.1 REST vs SOAP

Caratteristiche	REST	SOAP
Protocollo di trasporto	HTTP	Vari protocolli (ad esempio, HTTP, TCP, SMTP)
Formato del messaggio	Vari formati (ad esempio, XML, JSON)	XML-SOAP (envelope, header, body)
Identificatori di servizio	URI	URI e indirizzamento WS
Descrizione del servizio	- Documentazione testuale - Linguaggio di descrizione delle applicazioni web (WADL) – **OpenAPI-Swagger**	Web Services Description Language (WSDL)
Service Discovery	Nessun supporto standard	UDDI

5.6.5.2 Da composizione a un modello comune

I sistemi distribuiti per funzionare devono essere basati su un modello condiviso. Precedentemente, con WSDL/SOAP, i sistemi SOA devono concordare un API comune e messaggi, il che porta ad avere una struttura ben definita ma eccessivamente complessa.

REST è costruito intorno all'idea di ***semplificare l'accordo*** (attraverso defaults). I nomi, i verbi e i tipi di contenuto sono concetti fondamentali utilizzati per definire e manipolare le risorse attraverso l'interfaccia RESTful.

- **nouns**
  Rappresentano le risorse all'interno del sistema. Sono necessari per nominare le risorse di cui si può parlare (scambiate, cancellate, create);
- **verbs**
  Rappresentano le azioni che possono essere eseguite sulle risorse;
- **content types** (ad esempio, `application/json`)
  Definiscono quali rappresentazioni delle informazioni sono disponibili. Rappresentano il formato dei dati scambiati tra client e server.
  La scelta del tipo di contenuto dipende dalle esigenze dell'applicazione e dalle preferenze degli sviluppatori. ***JSON è diventato il formato preferito*** per i servizi web RESTful per la sua leggibilità e facilità di utilizzo da parte di applicazioni basate su JavaScript e non solo.

5.6.5.3 Principi REST

I principi di REST adottano i componenti architettonici del Web: URI E **HTTP**

1. Le **risorse** sono *identificate da URI*. (es. `/baseURL/resourceType/{id}`).
2. Le **risorse** vengono manipolate attraverso le loro *rappresentazioni*.
   Possono esistere *più rappresentazioni* di una risorsa (ad esempio, XML, JSON, HTML).

   ```
   Accetta: text/xml, application/xml, application/xhtml+xml,
   text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
   Accept-Language: en-us,en;q=0.5
   Codifica di accettazione: gzip,deflate
   Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7 "q" = relative preference(0-1)
   ```

   Il cliente può richiedere una particolare rappresentazione tramite le intestazioni della richiesta.
3. **Comunicazione** tramite **messaggi**, che sono *autodescrittivi* e *senza stato*.
4. **Lo stato dell'applicazione** evolve per mezzo della *manipolazione* delle risorse da parte del client.
5. **Hypermedia** è il modo per che ha il server per guidare il comportamento del client
   - Links

Vantaggi:

- Architettura del sistema semplificata;
- Visibilità migliorata;
- Semplifica l'evolvibilità delle implementazioni.

Risorsa

Una risorsa è una qualsiasi informazione che può essere nominata: documenti, immagini, servizi, collezioni, ecc. ***Le risorse sono lo stato dell'applicazione***. Di conseguenza possono cambiare nel tempo, solitamente con l'interazione con il client.

Le risorse hanno **identificatori** (vincolo).
Esempio: `(unimib.it/{matricola}/pianoStudio)`

- `{matricola}` è chiamato *path parameter*

Le risorse espongono un'**interfaccia uniforme** (vincolo).
→ **GET, POST, PUT, DELETE + Identificatori nell'URI**

Principio dell'interfaccia uniforme
Il principio dell'interfaccia uniforme in REST è uno dei pilastri fondamentali dell'architettura RESTful. Questo principio implica che tutte le risorse all'interno di un sistema REST siano accessibili e manipolabili attraverso un insieme comune e uniforme di interfacce. Ciò significa che indipendentemente dalla risorsa a cui si sta accedendo o dalla specifica azione che si sta cercando di eseguire su di essa, il modo in cui ci si interfaccia con la risorsa deve essere coerente e prevedibile.

Architettura client-server
Su richiesta, un servizio può fornire una rappresentazione di una risorsa su richiesta da parte di un client. Questo avviene all'interno di un'**architettura client-server**, dove il client fa una richiesta al server per ottenere una rappresentazione specifica della risorsa desiderata.

Esempio: immagina un'applicazione che gestisce elenchi di esami per gli studenti. Se un client, come un'applicazione web o un'applicazione mobile, vuole ottenere l'elenco degli esami registrati per un determinato studente, può fare una richiesta al server indicando l'ID dello studente di interesse. Il server, seguendo i principi di REST, risponderà con una rappresentazione dei dati degli esami registrati per lo studente richiesto. Questa rappresentazione potrebbe essere in formato JSON, XML o altro, a seconda delle convenzioni dell'applicazione.

Manipolazione di risorse attraverso rappresentazioni
Un client può modificare una risorsa inviando una rappresentazione modificata della risorsa stessa al server, senza dover invocare azioni specifiche. In altre parole, il client può effettuare operazioni sulla risorsa attraverso la manipolazione delle rappresentazioni della risorsa stessa.
Esempio: se hai un'applicazione web che gestisce elenchi di esami per gli studenti e segui il principio REST, uno studente può aggiungere un nuovo esame all'elenco semplicemente inviando al server una rappresentazione dei dati dell'esame da aggiungere, senza bisogno di chiamate API esplicite per "aggiungere" l'esame. Il server, seguendo il principio REST, dovrebbe essere in grado di interpretare e gestire correttamente la richiesta, aggiornando l'elenco degli esami di conseguenza.

Hypermedia come motore dello stato dell'applicazione
Esiste un principio, noto come "Hypermedia as the Engine of Application State" o "Hypermedia come motore dello stato dell'applicazione", il quale sottolinea che le rappresentazioni delle risorse restituite dal server devono contenere collegamenti ipertestuali (link) a risorse correlate o azioni che possono essere intraprese dal client. In altre parole, il client non deve dipendere da conoscenze pregresse o hardcoded sulle interfacce del server, ma deve poter esplorare e interagire con le risorse seguendo i collegamenti forniti dal server stesso.

- Manipolazione dello stato dell'applicazione = lo stato corrente manipolato dal client.
- Hypermedia = link.

Seguire un collegamento può essere visto come una transizione di stato dell'applicazione dove il server fornisce una nuova rappresentazione; il client assume tale stato. Il server "guida" il client verso nuovi stati fornendo collegamenti all'interno di rappresentazioni di risorse. Tutte le rappresentazioni devono contenere dei link -> Niente link, niente Web.
I client devono seguire solo i link forniti dal server.

Link
I link dovrebbero essere "classificati" con un significato semantico, in modo che un cliente intelligente sappia a cosa punta il link. Aiuta anche ad allentare l'accoppiamento tra client e server.
I link permettono

- Scoperta delle funzionalità
- L'ordine delle interazioni (ad esempio, come presentato)
- Indipendenza dalla posizione (nell’albero delle API) e sicurezza

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.account+json
Content-Length: ...
{
"account": {
"numero_conto": 12345,
"saldo":
        {
        "valuta": "usd",
        "valore": 100,00
        },
"_links": {
        "deposito": "/accounts/12345/deposit",
        "prelievo": "/accounts/12345/withdraw",
        "trasferimento": "/accounts/12345/transfer",
        "chiusura": "/accounts/12345/close"
            }
      }
}
```

Esempio: immagina di ottenere un elenco degli esami registrati per uno studente dal server. Oltre ai dati degli esami, la rappresentazione restituita potrebbe includere anche collegamenti ipertestuali che consentono al client di eseguire azioni come aggiungere o eliminare una registrazione. Il client può quindi seguire questi link per intraprendere le azioni desiderate, senza dover memorizzare o dedurre a priori gli endpoint specifici per eseguire tali azioni. Questo approccio rende il sistema più flessibile, scalabile e facile da mantenere nel tempo.

Interazioni prive di stato

Esiste un principio delle "Interazioni prive di stato" in REST e indica che ogni richiesta inviata dal client al server deve includere tutte le informazioni necessarie per il server per comprendere e soddisfare la richiesta. Il server non deve mantenere alcun contesto o stato relativo alla sessione del client tra una richiesta e l'altra.
In altre parole, ogni richiesta inviata dal client deve essere autocontenuta e non deve dipendere da alcuna informazione di stato precedentemente memorizzata sul server. Questo approccio rende le interazioni tra client e server più semplici e scalabili, in quanto ogni richiesta può essere elaborata in modo indipendente dalle altre e non richiede la memorizzazione di informazioni di stato sul server.

Esempio: se un client desidera ottenere i dettagli di un esame da un server RESTful, la richiesta deve includere tutte le informazioni necessarie per identificare e recuperare l'esame desiderato. Non ci si può aspettare che il server ricordi informazioni precedenti sul client o sul contesto della sessione per soddisfare la richiesta.

Messaggi autodescrittivi

Esiste un principio della "mancanza di stato", il quale richiede che i messaggi scambiati tra client e server in un'architettura RESTful siano autodescrittivi. Ciò significa che ogni messaggio deve contenere tutte le informazioni necessarie per essere compreso e interpretato, senza dipendere da uno stato di sessione mantenuto sul server.
Inoltre, per garantire che i messaggi siano autodescrittivi, è comune utilizzare tipi di media standard per rappresentare i dati, come ad esempio `application/json` per i dati JSON o `application/xml` per i dati XML. Questi tipi di media standard permettono ai client e ai server di comprendere il formato dei dati scambiati, senza richiedere convenzioni personalizzate o ambiguità sul loro significato.
Oltre ai dati stessi, i messaggi possono contenere anche meta-dati e dati di controllo che forniscono ulteriori informazioni sull'intenzione della richiesta o sulla sua elaborazione. Questi metadati possono essere utilizzati per fornire indicazioni aggiuntive al server o al client su come gestire la richiesta o la risposta.

Cacheability

il concetto di "cacheability" si riferisce alla capacità di una risorsa di essere memorizzata temporaneamente (cache) da un client (ad esempio nel browser) o da un intermediario (come un server proxy) per migliorare le prestazioni e ridurre il carico sul server di origine. Quando una risorsa è "cacheable", significa che è possibile memorizzarla localmente su un client o su un server proxy per un certo periodo di tempo, anziché richiederla nuovamente al server di origine ad ogni richiesta.

Come già detto, esistono due tipi di cache:

- Lato client
- Proxy/server
  Le architetture SOA di grandi dimensioni utilizzano un server proxy di caching (memcache, et similia). Ricordate che possono esserci cache lungo tutto il percorso tra il client e il server.

La cacheability richiede un sistema a livelli

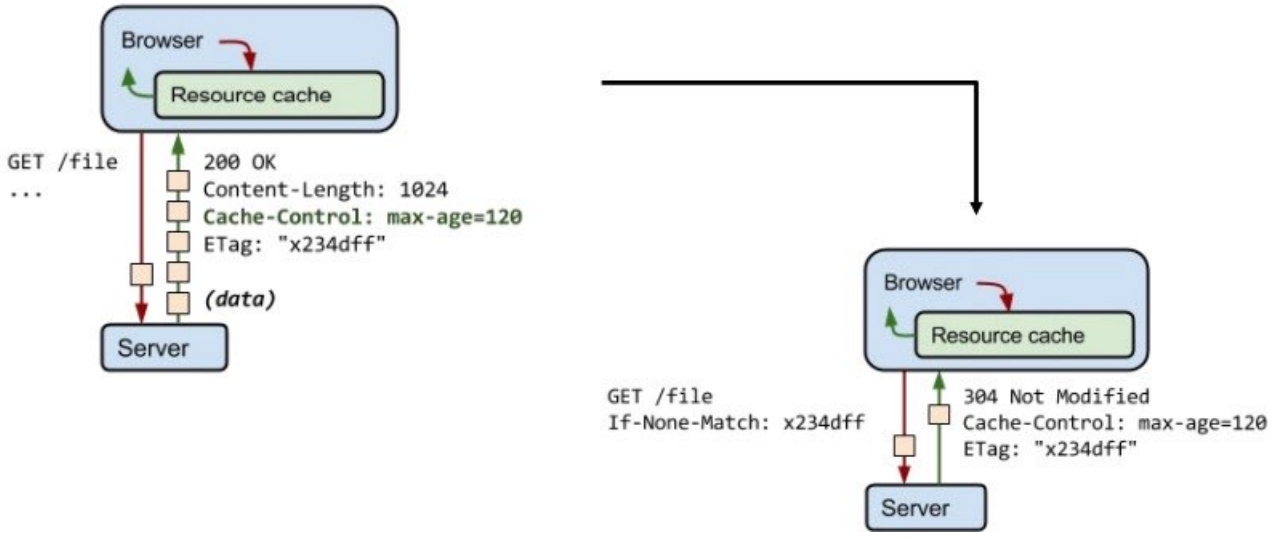
- Almeno un livello di cache
- Nella cache è memorizzata l'ultima rappresentazione di una risorsa
- Vengono memorizzati nella cache solo le risposte ai **GET** (non SSL), non i POST ecc.
- Una cache viene invalidata quando viene modificato lo stato della risorsa (PUT, DELETE).

La cache riduce la latenza e il traffico di rete.

Esempio di caching locale
Un GET condizionale è una richiesta HTTP GET che può restituire una risposta HTTP 304 (invece di HTTP 200). Una risposta HTTP 304 indica che la risorsa non è stata modificata dalla precedente GET e quindi la risorsa non viene restituita al client in tale risposta.

Casi:

1. La risorsa non è presente nella cache
   Il browser deve richiedere la risorsa al server mediante una GET condizionale (GET /file)
2. La risorsa è presente nella cache e non è stato modificato
   Il browser riceve dalla GET condizionale HTTP 304 e utilizza la risorsa presente nella cache.
3. La risorsa è presente nella cache ma è stato modificato
   Il browser effettua GET /file per aggiornare la risorsa. (GEt /file If-None-Match: ...)



5.6.5.4 Utilizzo di HTTP per applicazioni REST

La ricetta REST:

1. Definire i sostantivi
2. Definire i formati
3. Scegliere le operazioni
4. Evidenziare i codici di ritorno

Intestazione	Proprietà
Expires	Data HTTP. Mantenere in cache fino alla scadenza

Intestazione	Proprietà
Cache-Control: max-age	Un secondo. Mantenere in cache per il tempo indicato
Cache-Control: s-maxage	Secondi. Come sopra, ma solo per i proxy
Cache-Control: public	La risorsa può essere salvata in una cache condivisa
Cache-Control: no-cache	La risorsa può essere memorizzata nella cache, ma deve essere convalidata dal server di origine prima di ogni riutilizzo
Cache-Control: no-store	Non memorizzare nella cache
Cache-Control: must-revalidate	La risorsa può essere memorizzata in cache e riutilizzata quando è “fresca". Se diventa stantia (stale), deve essere convalidata con il server di origine prima di essere riutilizzata.
Cache-Control: proxy-revalidate	Come sopra, ma solo per i proxy

Definizione dei nomi (URI)

Tutto in un sistema REST è una risorsa, un nome.

- **Ogni risorsa ha un URI**. Idealmente solo uno.
- Gli URI devono essere descrittivi
  Se vi ritrovate a creare dei verbi, modificateli in nomi (ApproveExpense -> ExpenseApprovals).
  Esempio: `http://example.com/expenses`
- Gli URI dovrebbero essere opachi
  Esempio: `https://www.w3.org/DesignIssues/Axioms.html#opaque)`
  I client automatizzati (non umani) non dovrebbero dedurre i metadati da un URI. Importante non divulgare le informazioni sulla piattaforma
  Esempio: `/spese.php/123`
- Gli URI dovrebbero essere mantenuti invariati - persistenti
  Esempio: `http://www.w3.org/Provider/Style/URI`
  Gli URI non cambiano (semplicità, stabilità, gestibilità). Requisito fondamentale del web semantico.
- Utilizzare le path variables per codificare la gerarchia
  Esempio: `/spese/pendenti/123`
  `(/spese/pendenti/{id})
- Usare altri segni di punteggiatura per evitare di sottintendere una gerarchia.
  Esempio: `/spese/Q107;Q307 /spese/lacey,peter`
- Usare le query variables per implicare le condizioni di filtraggio
  Le query variables non sono così necessarie come si pensa
  Esempio: `/expenses?start=20070101&end=20071231` dovrebbe essere `/expenses/20070101-20071231`
  Le cache tendono a ignorare gli URI con variabili di query.
- Lo spazio URI è infinito

Definire i formati

Né HTTP né REST impongono un'unica rappresentazione dei dati.

- Una risorsa può avere diverse rappresentazioni
  Esempio: XML, JSON, binario (ad esempio, jpeg), coppie nome/valore
- Evitare di creare rappresentazioni personalizzate
  Utilizzare tipi di media ben noti (tipi MIME registrati dalla IANA)
  Questo perchè rende i contenuti più accessibili ed è anche parte del vincolo di messaggistica autodescrittiva.

- Le rappresentazioni in entrata devono essere uguali a quelle in uscita.
  Un client deve essere in grado di accedere (GET) e modificare (PUT) un documento mentre il server deve scartare tutti i dati estranei.

Riassunto REST

REST mette al centro la semplicità

- La cache migliora i tempi di risposta e riduce il carico del server
- L'assenza di stato e la riduzione delle comunicazioni facilitano il bilanciamento del carico tra i server.
- Software meno specializzato perché le tecnologie sottostanti sono ben note e semplici
- L'identificazione delle risorse avviene tramite meccanismi standard, non sono necessari nomi aggiuntivi.

REST è (basato su) standard

- I principi REST enfatizzano l'uso corretto e completo del protocollo HTTP per pubblicare servizi sul Web.
- REST+HTTP fornisce un meccanismo leggero e stratificato per l'integrazione di dati e servizi.
- REST+HTTP fornisce una piattaforma applicativa distribuita e guidata dagli ipermedia.

Scegliere le operazioni

Obiettivo: Scegliere le operazioni che possono essere applicate alle risorse.

L'HTTP ha un'interfaccia utente limitata (insieme di verbi/operazioni/metodi). Tuttavia, per la maggior parte delle applicazioni, i metodi di base di HTTP sono sufficienti.

Operazione	Descrizione	Cache	Safe	Idempotente
OPTIONS	Rappresenta una richiesta di informazioni sulle opzioni di comunicazione disponibili sulla catena richiesta/risposta identificata dal Request-URI	✓	✓	✓
GET	Significa recuperare qualsiasi risorsa identificata dall'indirizzo Request-URI	✓	✓	✓
HEAD	Identico a GET, tranne per il fatto che il server NON DEVE restituire un message-body nella risposta.	✓	✓	✓
POST	Utilizzato per richiedere che il server di origine accetti l'entità contenuta nella richiesta come nuovo subordinato della risorsa identificata dal Request-URI nella Request-Line			
PUT	Richiede che la risorsa allegata sia memorizzata nell'ambito del Request-URI fornito			✓
DELETE	Richiede che il server di origine cancelli la risorsa identificata dal Request-URI			✓
TRACE	Usato per invocare un loop-back remoto, a livello di applicazione, del messaggio di richiesta		✓	✓
PATCH	Viene utilizzato per eseguire aggiornamenti parziali su una risorsa.			Può essere

Reminder:

- **Safe** = Un metodo HTTP è safe (sicuro) se non altera lo stato del server. In altre parole, un metodo è sicuro se porta a un'operazione di sola lettura.
- **Idempotente** = gli effetti collaterali di N > 0 richieste identiche è lo stesso di una singola richiesta.

Situazione	PUT	POST
Creazione nuove risorse	Da utilizzare se il client sceglie l'URI (l'id)	Da utilizzare se il server sceglie l'URI (l'id)

Quando si tratta di creare nuove risorse in un'applicazione web, è essenziale adottare una strategia coerente per gestire le richieste dei clienti e garantire un corretto funzionamento del server. In questo contesto, l'uso dei metodi HTTP come POST e PUT riveste un'importanza cruciale.

Il metodo POST viene utilizzato quando il server è responsabile di scegliere l'identificatore univoco (URI) per la nuova risorsa. Questo approccio è appropriato quando si desidera che il server mantenga il controllo sul processo di creazione della risorsa. D'altra parte, il metodo PUT viene impiegato quando il cliente ha il compito di fornire l'URI per la risorsa. Questo consente al cliente di specificare in modo esplicito l'ubicazione della risorsa nel sistema.

È fondamentale comprendere il significato di POST come "Process this". Questo metodo non solo esegue un'azione sulla risorsa, ma restituisce anche un risultato. Spesso, viene utilizzato per mascherare chiamate di procedura remota (RPC), come l'invocazione di funzioni, ma è importante non abusarne e utilizzarlo con parsimonia.

Tuttavia, il "Process this" POST può risultare prezioso in determinati contesti, come nella creazione di risorse o quando è necessario introdurre un nuovo verbo nell'applicazione. È importante riflettere sulle decisioni progettuali, specialmente se si creano risorse che non possono essere recuperate tramite GET. In tali casi, è consigliabile rivedere il design dell'API per garantire coerenza e facilità d'uso.

Codice di stato della risposta

Il codice di stato della risposta viene generato dal server per indicare l'esito di una richiesta.

Codice	Descrizione
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.

La corretta identificazione dei codici di stato associati a una richiesta è una decisione cruciale nello sviluppo REST.