

Appunti di

Tecniche di programmazione concorrente e distribuita

Rosario Terranova

v 1.0.3

Sommario

Introduzione.....	2
Sistema distribuito	2
Caratteristiche dei sistemi distribuiti	3
Openess.....	3
Scalabilità	4
Socket in Java	5
Indirizzi IP e porte	5
Tipi di socket	6
Datagram Socket e classi correlate	7
Stream socket	11
Server e socket paralleli	13
JSP E SERVLET.....	19
WEB SERVICES.....	25

Introduzione

Le architetture dei sistemi informativi si sono sviluppate e evolute nel corso degli anni passando da schemi centralizzati (i primi computer erano stati realizzati per svolgere calcoli solo localmente) a modelli distribuiti e diffusi, maggiormente rispondenti alle necessità di decentralizzazione e di cooperazione delle moderne organizzazioni. Tale distribuzione è dovuta dalla possibilità dei computer di comunicare, caratteristica che è venuta solo molto dopo.

I computer oggi sono essenzialmente distribuiti ed eseguono elaborazioni distribuite. I sistemi distribuiti sono alla base di tutte le più importanti applicazioni informatiche.

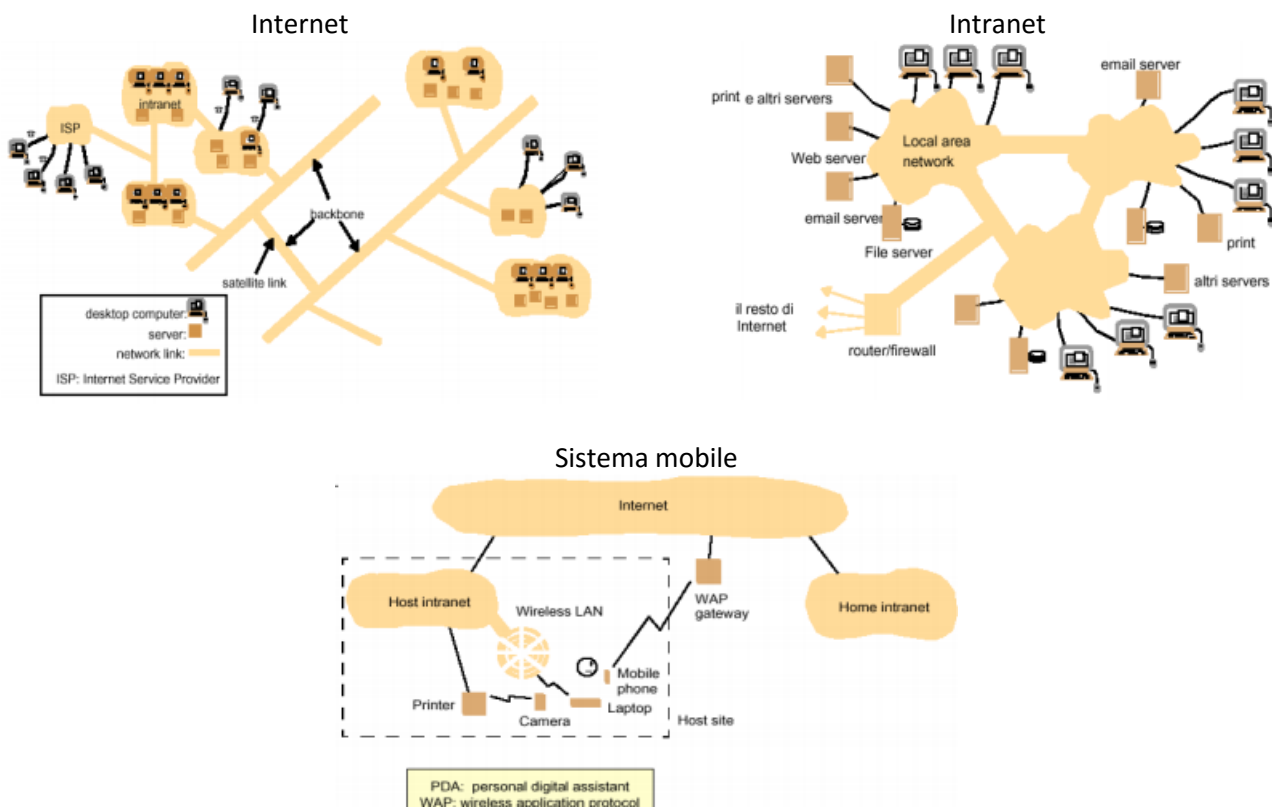
Sistema distribuito

Un sistema distribuito è: una collezione di computer indipendenti (sia software che hardware autonomi) spazialmente separati che comunicano e coordinano tra loro le loro azioni attraverso scambio di messaggi, apparendo agli utenti come un sistema singolo coerente.

Caratteristiche di base

- **Distribuzione e località:** ogni entità gode di una visione non accurata e non completa dell'intero sistema
- **Concorrenza:** le entità eseguono le loro azioni in modo concorrente
- **Guasti parziali:** alcune entità potrebbero guastarsi mentre altre continuano la loro attività, il sistema nel suo complesso può rimanere funzionante.

Esempi di sistemi distribuiti



Ma un sistema distribuito è molto di più: si possono considerare parte del sistema tutti i componenti software che girano sui vari dispositivi, quindi gli algoritmi che vengono implementati per risolvere problematiche di base in questi sistemi (comunicazione molti a molti, problemi di coordinamento).

Altri esempi di sistemi distribuiti:

- **Sistema di controllo del traffico aereo:** dati che devono essere trasferiti dalla sorgente alla destinazione per controllo/elaborazione/memorizzazione
- **Web:** attraverso un web browser, gli utenti possono accedere a documenti di vari tipi, ascoltare musica, vedere video e interagire con un illimitato numero di servizi.

Caratteristiche dei sistemi distribuiti

Sfide dei sistemi distribuiti

Ipotizziamo che una azienda vuole condividere i propri dati conservati in diversi database di diversi computer; se ne vogliono rendere pubblici (accessibili solo alcuni tipi), e l'accesso avviene attraverso un'applicazione client in grado di accedere ai dati. Bisogna stare attenti alle seguenti caratteristiche:

- **Sicurezza:** solo i client autorizzati possono accedere e solo ai dati pubblici.
- **Interoperabilità (o openness):** tecnologie e metodologie usate per lo storage dei dati potrebbero essere diverse, il client deve accedere ai dati in modo trasparente rispetto a queste diversità. Tecnologie e metodologie potrebbero cambiare nel tempo, il sistema deve essere in grado di integrarle facilmente.
- **Scalabilità:** tutti i dati messi a disposizione dai diversi computer dell'azienda potrebbero essere spostati in un solo database accessibile a tutti, ma se il numero di client aumenta? Il sistema nel suo complesso non deve perdere prestazioni.
- **Affidabilità e Disponibilità:** durante l'update di un dato o un errore del client che lo richiede, il database deve rimanere in uno stato consistente e comunque accessibile ad altri client.

Caratteristiche di un buon sistema

In base alle sfide sovraesposte, dunque, un buon sistema distribuito è sicuro, aperto, scalabile, affidabile e disponibile.

Openess

Il numero a volte elevatissimo (a volte ordine di decine di migliaia) di sviluppatori di software indipendenti rende lo sviluppo di una piattaforma distribuita un lavoro molto complesso e difficile da gestire. Per questo si utilizza una convenzione standar chiamata openness, caratterizzata da:

- **Interoperabilità:** La capacità di due implementazioni di sistemi diversi a cooperare usando servizi/componenti specificati da uno standard comune.
 - **Es. Mobile code and Java Virtual Machine:** manda il codice (java applet), il codice è interpretabile da ogni hardware perché viene generato per una macchina virtuale; il codice (chiamato bytecode) è interpretabile dalla virtual machine che lo traduce in linguaggio macchina e lo esegue. In questo modo il programma è indipendente dal sistema operativo.
- **Portabilità:** La capacità di un servizio/componente implementato su un sistema distribuito A, di essere eseguito senza modifiche su un sistema B.
- **Estendibilità:** La capacità di un sistema distribuito di aggiungere componenti e servizi e di essere integrati nel sistema distribuito già in esercizio.
- **Evolvability:** La capacità di un sistema distribuito di evolvere nel tempo, per esempio fare convivere differenti versioni di uno stesso servizio.

Interface Definition Language

Condizione necessaria per la openness è la stesura di una **documentazione e specifica delle interfacce software** chiave dei componenti di un sistema. Essa viene creato usando l'*Interface Definition Language*, la quale descrive la sintassi di un servizio/componente, funzioni disponibili, parametri di input/output, eccezioni etc.

La Specifica di un componente/servizio si dice propria se è:

- **Completa.** Una specifica è completa se ogni cosa necessaria ad una implementazione è stata specificata. Se una specifica non è completa l'implementatore deve aggiungere dettagli di specifica che a quel punto dipendono dall'implementazione.
- **Neutrale.** Una specifica è neutrale se non offre alcun dettaglio su una possibile implementazione

Middleware openness

Con middleware si intende un insieme di programmi informatici che fungono da intermediari tra diverse applicazioni e componenti software, che suportano lo sviluppo di applicazioni distribuite.

Funzionalità:

- **Accesso:** permettere di accedere a risorse locali e remote con le stesse modalità
- **Localione:** permettere di accedere alle risorse senza conoscerne la locazione
- **Concorrenza:** permettere ad un insieme di processi di operare concorrentemente su risorse condivise senza interferire tra loro
- **Guasti:** permettere il mascheramento dei guasti in modo che gli utenti possano completare le operazioni richieste anche se occorrono guasti hw e/o sw
- **Mobilità:** permettere di spostare risorse senza influenzare le operazioni utente
- **Prestazioni:** permettere di riconfigurare il sistema al variare del carico
- **Scalabilità:** permettere alle applicazioni di espandersi in modo scalabile senza modificare la struttura del sistema e degli algoritmi applicativi.



Tre tipi di middleware: communications middleware, database middleware and systems middleware.

Complessità del middleware

La complessità del middleware è relativa alla complessità delle relazioni tra i componenti del sistema. Le prestazioni di una soluzione basata su sistema distribuito non sempre migliorano rispetto ad una basata su sistema centralizzato. Il middleware, necessario per fornire servizi che sfruttano le caratteristiche di un sistema distribuito, in generale può diminuire le prestazioni.

Scalabilità

Un sistema è scalabile se rimane operativo con adeguate prestazioni anche se il numero di entità (risorse e di utenti) aumenta sensibilmente

- Computers connessi ad internet: 188 nel 1979, 130000 nel 1989, 56000000 nel 1999
- Web Servers connessi ad internet: 0 nel 1989, 5000000 nel 1999

La centralizzazione è contro la scalabilità:

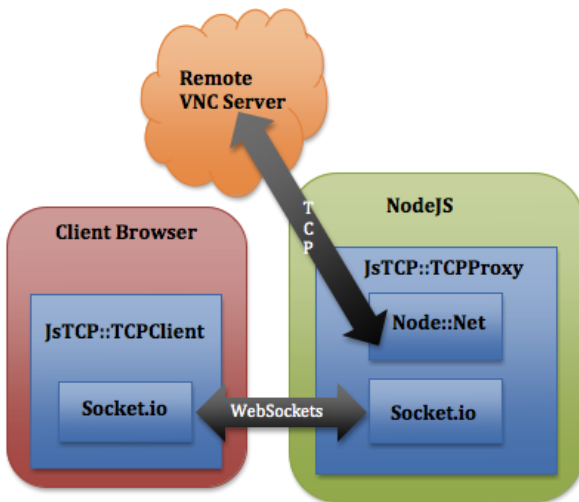
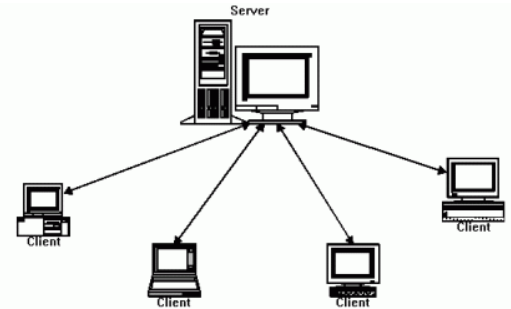
- Servizi (singolo servizio per tutti gli utenti)
- Dati (singola tabella per tutti gli utenti)
- Algoritmi (fare routing basato su informazioni complete)

Quindi, nei sistemi distribuiti, per controllare le perdite di prestazioni si usano algoritmi che non richiedono di dialogare con tutto il set di user o accedere all'intero set di dati di un sistema distribuito, ma che richiedono di dialogare con un numero di users (o di accedere ad un set di dati) che non cresca linearmente con il numero di users/taglia dei dati. Per evitare i colli di bottiglia nel sistema si usa la replicazione di servizi (distributed DNS) e Dati; da qui abbiamo problemi di coordinamento e problemi di consistenza.

Socket in Java

Internet è un insieme di sottoreti (LAN, WAN, ecc.) connesse tra loro tramite router o modem. In una rete alcune macchine possono fornire dei servizi ad altre macchine connesse tramite il modello client/server.

- **Client:** macchina o processo che richiede un servizio ad un'altra macchina o processo attraverso un protocollo.
- **Server:** fornisce un servizio al client.
- **Protocollo:** un set di regole che descrive come trasmettere dati.



Comunicazioni attraverso socket

I sistemi che usano il modello client/server comunicano attraverso le **socket**, ovvero dei punti di accesso per il canale di comunicazione di due parti che si risiedono su host diversi (o sullo stesso host).

L'immagine qua a sinistra chiarisce meglio il concetto; si può osservare che il *Client Browser* manda messaggi al *NodeJS* attraverso il canale di comunicazione gestito dalle *Socket.io*.

L'uso dei socket fornisce un'astrazione rispetto a meccanismi di più basso livello per la comunicazione in rete. Una socket in uso è solitamente legata ad un indirizzo ed ad una porta. La libreria Java che fornisce il supporto per le socket è **java.net**.

```
import java.net.*
```

Indirizzi IP e porte

Un indirizzo IP (Internet Protocol) è costituito da quattro numeri interi di 8 bit (con valori da 0 a 255) separati da punti (es. 151.97.253.200) e consente di individuare univocamente la posizione di una sottorete e di un host al suo interno. Un indirizzo IP assolve essenzialmente a due funzioni principali:

1. identificare un dispositivo sulla rete, e di conseguenza
2. fornirne il percorso per la sua raggiungibilità da un altro terminale o dispositivo di rete in una comunicazione dati a pacchetto.

Per usare le socket, oltre a conoscere l'indirizzo IP dell'host a cui connettersi, bisogna disporre dell'informazione sufficiente per collegarsi al processo server corretto. Per questo motivo esistono i **numeri di porta**.

Numeri di porta

I port number permettono di associare un servizio (un processo server che risponde alle richieste) ad un ben determinato numero. Il loro uso permette ad un calcolatore di effettuare più connessioni contemporanee verso altri calcolatori, facendo in modo che i dati contenuti nei pacchetti in arrivo vengano indirizzati al processo che li sta aspettando.

I numeri di porta sono interi di 16 bit (da 0 a 65535). Quelli da 0 a 1023 sono riservati per i servizi standard (es. FTP porta 21, http porta 80, etc.), mentre i numeri da 1024 a 65535 sono disponibili per i processi utente.

Le connessioni avvengono sempre specificando un indirizzo IP ed un numero di porta. I pacchetti appartenenti ad una connessione saranno quindi identificati dalla quadrupla [*<indirizzo IP sorgente>*, *<indirizzo IP destinazione>*, *<porta sorgente>*, *<porta destinazione>*]. I pacchetti nella direzione opposta avranno ovviamente sorgente e destinazione scambiati.

Mentre la porta di destinazione è l'identificativo univoco del processo applicativo, la porta di sorgente è assegnata casualmente in maniera tale da identificare univocamente la connessione da parte del mittente col destinatario eventualmente tra più computer all'interno di una rete locale.

Il livello di trasporto (tipicamente realizzato dal sistema operativo) associa a ciascuna porta utilizzata un punto di contatto (ad esempio, una socket), utilizzato da uno (o più) processi applicativi per trasmettere e/o ricevere dati.

Per poter inviare con successo un pacchetto con una certa porta di destinazione, ci deve essere un processo che è "in ascolto" su quella porta, ovvero che ha chiesto al sistema operativo di ricevere connessioni su quella porta

Tipi di socket

Esistono due modi principali per comunicare in rete, ed in corrispondenza a tali due modi di comunicazione si hanno i seguenti tipi di socket:

- **Connection Less:** lo scambio di dati a pacchetto tra mittente e destinatario (o destinatari) non richiede l'operazione preliminare di creazione di un circuito, fisico o virtuale, su cui instradare l'intero flusso dati in modo predeterminato e ordinato nel tempo (sequenziale).
 - **Socket a Datagrammi (Datagram):** trasferiscono messaggi di dimensione variabile, preservando i confini, ma senza garantire ordine o arrivo dei pacchetti. Supportate nel dominio Internet dal protocollo **UDP** (User Datagram Protocol). Non instaurano una connessione tra client e server, e non verificano l'arrivo del dato o il ri-invio. Hanno il vantaggio di trasferire velocemente i dati. La classe Java che usa questo protocollo è *DatagramSocket*. I dati sono impacchettati tramite la classe *DatagramPacket*.
- **Connection Oriented:** modalità di comunicazione dati tramite la quale i dispositivi terminali usano un protocollo di comunicazione per stabilire una connessione logica o fisica end-to-end tra gli agenti della comunicazione prima della trasmissione di qualsiasi tipo di dato.
 - **Stream Socket:** forniscono stream di dati affidabili ed ordinati. Nel dominio Internet sono supportati dal protocollo **TCP** (Transfer Control Protocol). Permettono di creare una connessione tra client e server con 2 canali (uno per ciascuna direzione). Le classi Java che usano questo protocollo sono *Socket* e *ServerSocket*.

TCP

Protocollo di rete che si occupa di rendere affidabile la comunicazione dati in rete tra mittente e destinatario. Su di esso si appoggiano gran parte delle applicazioni della rete Internet. È presente solo sui terminali di rete (host) e non sui nodi interni di commutazione della rete di trasporto, implementato come strato software di rete all'interno del rispettivo sistema operativo ed il sistema terminale in trasmissione vi accede automaticamente dal browser attraverso l'uso di opportune chiamate di sistema definite nelle API di sistema.

- mentre TCP è un protocollo orientato alla connessione, pertanto per stabilire, mantenere e chiudere una connessione, è necessario inviare pacchetti di servizio i quali aumentano l'overhead di comunicazione. Al contrario, UDP è senza connessione ed invia solo i datagrammi richiesti dal livello applicativo;
- UDP non offre nessuna garanzia sull'affidabilità della comunicazione ovvero sull'effettivo arrivo dei datagrammi, sul loro ordine in sequenza in arrivo, al contrario il TCP tramite i meccanismi di acknowledgement e di ritrasmissione su timeout riesce a garantire la consegna dei dati, anche se al costo di un maggiore overhead (raffrontabile visivamente confrontando la dimensione delle intestazioni dei due protocolli);
- l'oggetto della comunicazione di TCP è il flusso di byte mentre quello di UDP è il singolo datagramma.

L'utilizzo del protocollo TCP rispetto a UDP è, in generale, preferito quando è necessario avere garanzie sulla consegna dei dati o sull'ordine di arrivo dei vari segmenti (come per esempio nel caso di trasferimenti di file). Al contrario UDP viene principalmente usato quando l'interazione tra i due host è idempotente o nel caso si abbiano forti vincoli sulla velocità e l'economia di risorse della rete (es. instant messaging).

UDP

A differenza del TCP, l'UDP è un protocollo di tipo connectionless, inoltre non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi, ed è perciò generalmente considerato di minore affidabilità. In compenso è molto rapido (non c'è latenza per riordino e ritrasmissione) ed efficiente per le applicazioni "leggere" o time-sensitive. Ad esempio, è usato spesso per la trasmissione di informazioni audio-video real-time.

Infatti, visto che le applicazioni in tempo reale richiedono spesso un bit-rate minimo di trasmissione, non vogliono ritardare eccessivamente la trasmissione dei pacchetti e possono tollerare qualche perdita di dati, il modello di servizio TCP può non essere particolarmente adatto alle loro caratteristiche.

Datagram Socket e classi correlate

Classe Datagram Socket

Usare le datagram socket consiste nell'implementazione di un programma che effettua i seguenti passi fondamentali:

- Client:
 - a) crea socket;
 - b) manda richiesta sulla socket con indirizzo, porta e messaggio;
 - c) riceve dati dalla socket;
 - d) chiude la socket.
- Server:
 - a) crea socket, per ascoltare richieste in arrivo;
 - b) riceve dati dalla socket;
 - c) invia dati sulla socket al client che ne ha fatto richiesta;
 - d) chiude la socket.

La classe Java *DatagramSocket* usa il protocollo UDP per trasmettere dati attraverso le socket. Essa permette ad un client di connettersi ad una determinata porta di un host per leggere e scrivere dati impacchettati attraverso la classe *DatagramPacket*.

Nota: l'indirizzo dell'host destinatario è sul *DatagramPacket*.

I metodi della classe *DatagramSocket* sono:

```
void DatagramSocket() throws SocketException
void DatagramSocket(int port) throws SocketException
void receive(DatagramPacket p) throws IOException
receive() //blocca il chiamante fino a quando un pacchetto è ricevuto

void setSoTimeout(int timeout) throws SocketException
//usandolo il chiamante di receive() si blocca al max timeout millisec

void send(DatagramPacket p) throws IOException
void close()
```

Classe DatagramPacket

La classe Java con cui sono rappresentati i pacchetti UDP da inviare e ricevere sulle socket di tipo datagram è la classe *DatagramPacket*. Un oggetto istanza di *DatagramPacket* si costruisce inserendo nella chiamata al suo costruttore (**lato client**):

- il contenuto del messaggio (i primi *length bytes* dell'array *buf*)
- l'indirizzo IP del destinatario
- il numero di porta su cui il destinatario è in ascolto

```
public DatagramPacket(byte buf[], int length, InetAddress address, int port)
```

Se il pacchetto deve essere ricevuto (**lato server**) basta definire una istanza di *DatagramPacket* per il contenuto:

```
public DatagramPacket(byte buf[], int length)
```

I metodi della classe *DatagramPacket* sono:


```

InetAddress getAddress()
//restituisce l'indirizzo IP della macchina da cui il pacchetto sta per essere mandato o da cui è stato ricevuto

void setAddress(InetAddress addr)

int getPort()
//restituisce la porta della macchina remota a cui il pacchetto sta per essere mandato o da cui è stato ricevuto

void setPort(int iport)
byte[] getData() //restituisce i dati del pacchetto
void setData(byte[] buf)

```

Classe InetAddress

Con la classe `InetAddress` sono rappresentati gli indirizzi Internet, astruendo dal modo con cui sono specificati (a numeri o a lettere). Il metodo statico `getByName()` restituisce una istanza di `InetAddress` rappresentante l'host specificato, es. www.google.it.

```
static InetAddress getByName(String hostname)
```

Se viene passato il parametro `null` viene restituita una istanza di `InetAddress` rappresentante l'indirizzo di default dell'host locale.

Il metodo statico `getByAddress()` restituisce una istanza `InetAddress` rappresentante l'host specificato come indirizzo IP.

```
public static InetAddress getByAddress(byte[] addr)
```

Il metodo statico `getAllByName()` restituisce un array di istanze `InetAddress`; utile in casi di più indirizzi IP registrati con lo stesso nome logico.

```
public static InetAddress[] getAllByName(String hostname)
```

Il metodo statico `getLocalHost()` restituisce una istanza di `InetAddress` per la macchina locale; se tale macchina non è registrata oppure è protetta da un firewall, l'indirizzo è quello di **loopback: 127.0.0.1**.

Loopback è un meccanismo di test per le schede di rete, che catturano i messaggi contenenti tali indirizzi e li rimandano indietro all'applicazione che li ha mandati.

```
public static InetAddress getLocalHost()
```

Tutti i precedenti metodi possono sollevare l'eccezione **UnknownHostException** se l'indirizzo specificato non può essere risolto.

Esempio di uso di InetAddress con l'host locale

```

import java.net.*;
public class testLocalHost{
    public static void main(String [] args) {
        try {
            InetAddress host1 = InetAddress.getByName(null);
            // ovvero
            InetAddress host2 = InetAddress.getLocalHost();
            System.out.println("Host locale "+ host1.getHostName());
        }
        catch (UnknownHostException e) {
            System.out.println("Problemi con l'indirizzo locale");
            e.printStackTrace();
        }
    }
}

```

Esempio di creazione di socket datagram

```
import java.net.*;
public class testDataSocket {
    public static void main(String [] args) {
        try {
            // creazione socket datagram
            DatagramSocket dsocket = new DatagramSocket();
            byte[] buf = new byte[256];

            // preparazione pacchetto
            DatagramPacket packet = new DatagramPacket(buf, buf.length);

            // ricezione pacchetto tramite datagram
            dsocket.receive(packet);
            InetAddress mittAddr = packet.getAddress();
            System.out.println("Pacchetto ricevuto da "+mittAddr);
        }
        catch(Exception e) {
            System.out.println("Problemi: ");
            e.printStackTrace();
        }
    }
}
```

Applicazione d'esempio DatagramSocket

Sviluppiamo un'applicazione in cui il client invia dei pacchetti che indicano il nome di un file testo e la linea del file che vuole ricevere dal server. Attende la risposta e manda sullo schermo ciò che ha ricevuto. Il server, ad ogni richiesta, estrae dal file la linea e la invia al client. Se file o linea non esistono notifica l'errore al client.

Lato Client

```
import java.io.*;
import java.net.*;
public class clientText {
    public static void main(String [] args) {

        //Creazione socket ed eventuale settaggio opzioni
        DatagramSocket socket = new DatagramSocket();
        socket.setSoTimeout(30000);

        //Interazione con l'utente:
        BufferedReader stdIn = new
        BufferedReader(new InputStreamReader(System.in));
        System.out.print("Numero linea: ");
        String richiesta = stdIn.readLine();

        //Creazione del pacchetto di richiesta da mandare al server
        pack = DatagramUtility.buildPacket(addr, LineServer.PORT, richiesta);

        //Invio del pacchetto al server
        socket.send(pack);

        //Attesa del pacchetto di risposta
        packetIN = new DatagramPacket(buf, buf.length);
        socket.receive(packetIN);

        //Estrazione delle informazioni dal pacchetto ricevuto
        risposta = DatagramUtility.getContent(packetIN); } }
```

```

import java.io.*;
import java.net.*;

public class serverText {
    public static void main(String [] args) {

        //Creazione socket
        DatagramSocket socket = new DatagramSocket(PORT);

        //Attesa del pacchetto di richiesta:
        DatagramPacket packet = new DatagramPacket(buf,buf.length);
        socket.receive(packet);

        //Estrazione delle informazioni dal pacchetto ricevuto:
        String richiesta = DatagramUtility.getContent(packet);
        StringTokenizer st = new StringTokenizer(richiesta);
        nomeFile = st.nextToken();
        numLinea = Integer.parseInt(st.nextToken());

        //Creazione del pacchetto di risposta con la linea richiesta:
        String l = LineUtility.getLine(nomeFile, numLinea);
        pack=DatagramUtility.buildPacket(mittAddr, mittPort, l);

        //Invio del pacchetto al client:
        socket.send(pack);
    }
}

```

Socket Multicast

Nel caso di socket datagram, posso usare il multicast per mandare un pacchetto ad un insieme di processi con una sola invocazione a *send()*. Multicast significa proprio mandare informazioni ad un gruppo di destinatari simultaneamente.

La libreria java.net mette a disposizione la classe **MulticastSocket** per inviare messaggi multicast.

Chi vuole ricevere pacchetti mandati in multicast deve conoscere:

- la porta della socket usata dal server
- l'indirizzo del gruppo a cui vengono inviati messaggi
- I client che vogliono ricevere messaggi multicast devono unirsi al gruppo.

Passi per l'uso della socket multicast:

1. Creazione di un indirizzo di gruppo

```
InetAddress gruppo = InetAddress.getByName(ind)
```

2. Creazione di una socket per il multicast

```
MulticastSocket msocket = new MulticastSocket(porta)
```

3. Connessione della socket al gruppo

```
msocket.joinGroup(gruppo)
```

Tutti i processi che eseguono *joinGroup()* sulla socket multicast potranno ricevere i pacchetti datagram inviati a quel gruppo, quando invocano: *msocket.receive(packet)*;

Un client deve invocare *receive()* periodicamente fino a quando vuole ricevere pacchetti.

Un client può decidere ad un certo punto di uscire dal gruppo:

```
msocket.leaveGroup(gruppo)
```

Il processo server crea la socket per una porta ed invia, quando vuole, pacchetti datagram al gruppo:

```
InetAddress group = InetAddress.getByName(ind);
MulticastSocket msocket = new MulticastSocket(porta);
msocket.joinGroup(group);
DatagramPacket packet =
DatagramUtility.buildPacket(group, porta, linea);
msocket.send(packet);
```

Metodi della classe MulticastSocket:

```
joinGroup(InetAddress addr) throws IOException
leaveGroup(InetAddress addr) throws IOException
send(DatagramPacket p)
```

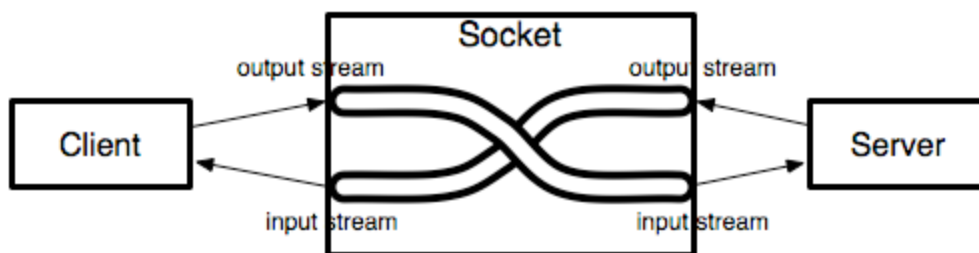
```
setTimeToLive(int ttl) throws IOException
```

//setta il tempo di vita per i pacchetti multicast mandati sulla socket (ttl deve essere compreso tra 0 e 255).

```
int getTimeToLive() throws IOException
```

Stream socket

Client e server instaurano una connessione che consiste di due flussi di dati, uno per l'input ed uno per l'output.



Le API invocate da client e server, per instaurare una connessione, sono diverse e quindi esistono classi di libreria Java diverse.

Dal lato client vi è il processo che vuole instaurare una connessione per inviare e ricevere dati. La classe Java che usa il client è *Socket*.

Dal lato server vi è un processo che attende una richiesta di connessione, la classe Java che usa è *ServerSocket*.

La connessione è individuata da 4 elementi:

- indirizzo IP del client,
- porta del client,
- indirizzo IP del server,
- porta del server.

Client di uno stream socket

Il client invoca il costruttore della socket, per creare la socket, specificando l'indirizzo e la porta dove risiede il processo server. L'istanziatura della socket e quindi la chiamata al costruttore instaura la connessione con il server, a patto che il server sia già in esecuzione e pronto per ricevere questa connessione.

```
Socket socket = new Socket(addr, PORT);
```

con *addr* (di tipo *InetAddress*) e *PORT* (di tipo *int*) che identificano il processo server

Se la creazione della stream socket ha successo, viene prodotta una connessione bidirezionale tra i due processi.

L'apertura della socket è implicita con il costruttore. La chiusura deve essere chiamata esplicitamente ed è necessaria per non impegnare troppe risorse di sistema (ovvero connessioni). Il numero di connessioni che un processo può aprire è limitato e quindi conviene mantenere aperte solo le connessioni necessarie.

Il metodo della classe Socket per chiudere la connessione e disconnettere il client dal server è *close()*.

Altri metodi della classe Socket sono:

```
InetAddress getInetAddress() //restituisce l'indirizzo remoto a cui la socket è connessa  
InetAddress getLocalAddress() //restituisce l'indirizzo locale  
int getPort() //restituisce la porta remota  
int getLocalPort() //restituisce la porta locale
```

Dopo la connessione, il client crea uno stream di input, tramite *getInputStream()*, con cui può ricevere i dati dal server.

```
InputStreamReader isr = new InputStreamReader(socket.getInputStream());
```

La classe *InputStreamReader* converte un flusso di byte in un flusso di caratteri. Per migliorare l'efficienza, creiamo un buffer che consente di leggere un gruppo di caratteri:

```
BufferedReader in = new BufferedReader(isr);
```

Ovvero, allo stream del socket applichiamo i filtri *InputStreamReader* e *BufferedReader*. Per leggere una linea dallo stream di input della socket:

```
String s = in.readLine()
```

Questa ci restituisce la stringa inviata dal server. Il client crea anche uno stream di output con cui invierà i dati al server in modo analogo allo stream di input.

```
OutputStreamWriter osw = new OutputStreamWriter(socket.getOutputStream());  
BufferedWriter bw = new BufferedWriter(osw);  
PrintWriter out = new PrintWriter(bw, true);
```

Per inviare una stringa sullo stream di output: *out.println(s)*

Quando la connessione non è più necessaria è bene che il client liberi la risorsa "connessione". Per fare ciò chiude la socket con: *out.close(); in.close(); socket.close();*

Server di uno stream socket

Dal lato server, bisogna creare una istanza di *ServerSocket*:

```
ServerSocket serverSocket = new ServerSocket(PORT);
```

e dopo mettere il server in attesa di una connessione per mezzo di:

```
Socket clientSocket = serverSocket.accept()
```

La chiamata ad *accept()* blocca il server fino a quando un client non instaura una connessione, tuttavia tramite il metodo *setSoTimeout(t)* si può bloccare la *accept()* solo per t millisecondi.

La *accept()* restituisce un oggetto di tipo *Socket*. Questo permette al server di usare gli stream che il client ha stabilito. *clientSocket.getInputStream()*

Il server crea gli stream di input e di output per poter poi ricevere e trasmettere dati. Allo stream di input del client corrisponderà lo stream di output del server e viceversa (vedi figura).

Una volta instaurata una connessione, la trasmissione dei dati avviene con gli stessi metodi sia nel client che nel server.

Creazione stream di input (nel server):

```
InputStreamReader isr = new InputStreamReader(clientSocket.getInputStream());  
BufferedReader in = new BufferedReader(isr);
```

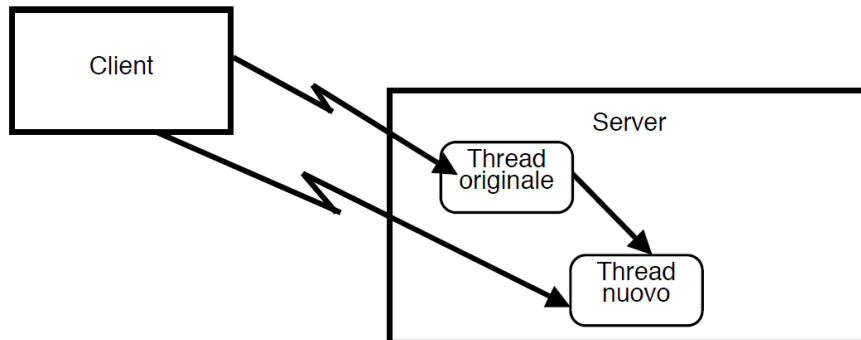
Creazione stream di input (nel server):

```
in.readLine();
```

Server e socket paralleli

Quando un server accetta una connessione esso può generare un nuovo thread per servire la richiesta, mentre il thread originale continua a stare in attesa (su `serverSocket.accept()`) di connessioni da altri client.

Questa soluzione permette di non perdere richieste di connessioni e di servire più client che fanno richieste in parallelo.



JAVASERVER PAGES

JavaServer Pages, di solito indicato con l'acronimo **JSP** (letto anche talvolta come Java Scripting Preprocessor) è una tecnologia di programmazione Web in Java per lo sviluppo della logica di presentazione di applicazioni Web, fornendo contenuti dinamici in formato HTML o XML. Si basa su un insieme di speciali tag, all'interno di una pagina HTML, con cui possono essere invocate funzioni predefinite sotto forma di codice Java (JSTL) e/o funzioni Javascript. In aggiunta, permette di creare librerie di nuovi tag che estendono l'insieme dei tag standard (JSP Custom Tag Library).

Nel contesto della piattaforma Java, la tecnologia JSP è correlata con quella delle **servlet**: all'atto della prima invocazione, le pagine JSP vengono infatti tradotte automaticamente da un compilatore JSP in servlet. Una pagina JSP può quindi essere vista come una rappresentazione ad alto livello di un servlet. Per via di questa dipendenza concettuale, anche l'uso della tecnologia JSP richiede la presenza, sul Web server, di un servlet container, oltre che di un server specifico JSP detto motore JSP (che include il compilatore JSP); in genere, servlet container e motore JSP sono integrati in un unico prodotto (per esempio, Tomcat svolge entrambe le funzioni).

JSP è una tecnologia alternativa rispetto a numerosi altri approcci alla generazione di pagine Web dinamiche, per esempio PHP, o ASP o la più tradizionale CGI. Differisce da queste tecnologie non tanto per il tipo di contenuti dinamici che si possono produrre, quanto per l'architettura interna del software che costituisce l'applicazione Web (e, di conseguenza, sui tempi di sviluppo, la portabilità, la modificabilità, le prestazioni, e altri aspetti di qualità del software).

JSP è quindi una soluzione “embed”, una pagina JSP è costituita da markup (X)HTML frammentato da sezioni di codice Java. Si potranno quindi modificare le parti sviluppate in Java lasciando inalterata la struttura HTML o viceversa.

PRINCIPALI VANTAGGI

Le JSP si basano su tecnologia Java ereditandone i vantaggi garantiti dalla metodologia object oriented e dalla quasi totale portabilità multiplatforma. Quest'ultima caratteristica si rivela tanto più vantaggiosa quanto più ci fosse necessità di ambienti di produzione con sistemi operativi diversi tra loro (es Windows e Linux).

Essere object oriented significa anche avere nel DNA una predisposizione al riuso del codice: grazie ai Java Bean si possono includere porzioni di codice che semplificano l'implementazione di applicazioni anche complesse, anche senza approfondite conoscenze di Java, e ne rendono più semplice la modifica e la manutenzione.

La gestione delle sessioni introdotta con le JSP favorisce lo sviluppo di applicazioni di commercio elettronico fornendo uno strumento per memorizzare temporaneamente lo stato delle pagine fino alla chiusura del browser.

CICLO DI VITA DELLA RICHIESTA

Una Java Server Page può essere invocata utilizzando due diversi metodi:

la richiesta può venire effettuata direttamente ad una pagina JSP, che grazie alle risorse messe a disposizione lato server, è in grado di elaborare i dati di ingresso per ottenere e restituire l'output voluto

la richiesta può essere filtrata da una servlet che, dopo l'elaborazione dei dati, incapsula adeguatamente i risultati e richiama la pagina JSP, che produrrà l'output.

SERVLET

APPLET JAVA

Molti programmatori che iniziano a cimentarsi con Java imparano quasi subito a destreggiarsi con gli Applet, ovvero programmi scritti in linguaggio Java che possono essere eseguiti da un web browser con elaborazione lato client. Le applet appaiono visivamente collocate all'interno di pagine Web, e sono solitamente usate per creare pagine dotate di funzioni interattive con l'utente non realizzabili con altre tecnologie per il Web statico, appartenendo dunque al paradigma del Web dinamico. Le Java applets sono eseguibili dai web browser che utilizzano la Java virtual machine (JVM).

Incapsulate all'interno di pagine web, le Applet sono utilizzate per fornire contenuti interattivi che il linguaggio HTML non è in grado di offrire. Per eseguirne il contenuto, la maggioranza dei Web Browser utilizza una sandbox, in modo da impedire alle applet di accedere alle informazioni salvate in locale sul computer. Il codice sorgente delle Applet viene scaricato dal web server attraverso il web browser ricevendo anche la pagina html che lo contiene. In alternativa, l'Applet può provvedere ad aprire una finestra personale senza doversi appoggiare a codice html per mostrare l'interfaccia grafica.

Le Applet mostrate nelle pagine web sono identificati dal tag html `<applet>...</applet>` (non ufficiale), al quale è preferibile `<object></object>`. Il tag html utilizzato specifica la posizione del sorgente dell'Applet da scaricare. Dato che il bytecode Java è indipendente dalla piattaforma, gli Applet Java possono essere eseguiti senza problemi dai browser delle principali piattaforme.

Un discorso analogo non può farsi, invece, per il cugino degli Applet: Le Servlet.

SERVLET

Una servlet è un programma scritto in Java e residente su un server, in grado di gestire le richieste generate da uno o più client, attraverso uno scambio di messaggi tra il server ed i client stessi che hanno effettuato la richiesta. Tipicamente sono collocate all'interno di Application Server o Web Application Server come, ad esempio, **Tomcat** (contenitore servlet open source sviluppato dalla Apache Software Foundation. Implementa le specifiche JavaServer Pages (JSP) e Servlet di Sun Microsystems, fornendo quindi una piattaforma software per l'esecuzione di applicazioni Web sviluppate in linguaggio Java).

Visto che, come detto, le Servlet sono scritte in Java esse possono avvalersi interamente delle Java API che, come è noto, consentono di implementare con relativa semplicità anche svariate funzionalità complesse e importanti come, ad esempio, l'accesso ad un database. Ad esse si aggiungono, poi, le più specifiche servlet API, che mettono a disposizione un'interfaccia standard utilizzata per gestire la comunicazione tra un client Web ed una Servlet.

È importante sottolineare il fatto che le Servlet, a differenza degli Applet, *non hanno delle GUI* associate direttamente ad esse. Pertanto, le librerie AWT e Swing non verranno mai utilizzate direttamente quando si desidera implementare una Servlet.

Altrettanto interessante è il fatto che, almeno in teoria, una Servlet non rappresenta unicamente (come invece si è spesso portati a ritenere) applicazioni basate sul protocollo HTTP (ovvero, un'applicazione di tipo Web). Per gli scenari non Web, infatti, si può fare riferimento alla classe `javax.servlet.GenericServlet` che, appunto, è in grado di utilizzare un protocollo generico. D'altra parte è necessario ammettere che, quasi sempre, quando si parla di Servlet si fa riferimento implicitamente ad una estensione particolare della classe `GenericServlet` identificata, per la precisione, dalla classe `HttpServlet`. Entrambe queste classi (`GenericServlet` e `HttpServlet`), implementano l'interfaccia `javax.servlet.Servlet`.

L'uso più frequente delle servlet è la generazione di pagine web dinamiche a seconda dei parametri di richiesta inviati al server dal client browser dell'utente. Negli ultimi anni non viene eseguita la programmazione delle servlet direttamente, ma si preferisce usare dei framework web che implementano la specifica servlet, oppure delle JavaServer Pages che vengono poi tradotte (compilate) in servlet a runtime.

I programmi che implementano le specifiche dei servlet possono girare all'interno di qualunque **servlet container** e non sono vincolati ad un particolare server. L'attuale versione della specifica servlet è la 3.0, corrispondente alla JSR 315. Lo standard delle servlet rientra all'interno di un vasto insieme di standard detto Java EE[3].

Una servlet può avere molteplici funzionalità e può essere associata ad una o più risorse web. Un esempio potrebbe essere un meccanismo per il riconoscimento dell'utente (**login**): quando si digita un URL del tipo `miosito/login` viene invocata una servlet che verifica la correttezza delle credenziali di accesso inserite appoggiandosi ad un database e indirizza ad una pagina di conferma o di errore a seconda del risultato.

Sotto quest'ottica una servlet è un programma che deve rispettare determinate regole e che processa in un determinato modo una richiesta HTTP. Nulla vieta che all'interno dello stesso server web possano girare più servlet associate a URL diversi, ognuna delle quali farà cose diverse e estenderà le funzionalità del server web.

FLUSSO DI UNA SERVER

Lo schema seguente mostra come lavora un HttpServlet



1. Un client invia una richiesta (request) per una servlet ad un web application server.
2. Qualora si tratti della prima richiesta, il server istanzia e carica la servlet in questione avviando un thread che gestisca la comunicazione con la servlet stessa. Nel caso, invece, in cui la Servlet sia già stata caricata in precedenza (il che, normalmente, presuppone che un altro client abbia effettuato una richiesta antecedente quella attuale) allora verrà, più semplicemente, creato un ulteriore thread che sarà associato al nuovo client, senza la necessità di ricaricare ancora la Servlet.
3. Il server invia alla servlet la richiesta pervenutagli dal client
4. La servlet costruisce ed imposta la risposta (response) e la inoltra al server
5. Il server invia la risposta al client.

È importante sottolineare che il contenuto della risposta non è necessariamente basato su un algoritmo contenuto all'interno della Servlet invocata (questo può essere, semmai, il caso di applicazioni molto elementari) ma, anzi, è spesso legato ad interazioni della Servlet stessa con altre sorgenti di dati (data source) rappresentate, ad esempio, da Data Base, file di risorsa e, non di rado, altre Servlet.

HTTPSERVLETREQUEST E HTTPSERVLETRESPONSE

Abbiamo visto, esaminando il flusso di esecuzione di una servlet, che il flusso stesso è incentrato su due componenti fondamentali: la richiesta (request, inviata dal client verso il server) e la risposta (response, inviata dal server verso il client). In Java, questi due componenti sono identificati, rispettivamente, dalle seguenti interfacce:

javax.servlet.http.HttpServletRequest

javax.servlet.http.HttpServletResponse

Un oggetto di tipo HttpServletRequest consente ad una Servlet di ricavare svariate informazioni sul sistema e sull'ambiente relativo al client. L'oggetto di tipo HttpServletResponse, invece, costituisce la risposta da inviare al client e che, come si è detto, può essere dipendente da svariati data source.

SERVLETCONTEXT

Un'altra importante interfaccia messa a disposizione dal Servlet engine è la **javax.servlet.ServletContext**. Attraverso di essa è possibile trovare un riferimento al contesto (context) di un'applicazione, ovvero ad una serie di informazioni a livello globale condivise tra i vari componenti che costituiscono l'applicazione stessa.

METODI PRINCIPALI DI UNA SERVLET

Creare una Servlet vuol dire, in termini pratici, definire una classe che derivi dalla classe HttpServlet. I metodi più comuni per molti dei quali si è soliti eseguire l'overriding nella classe derivata sono i seguenti:

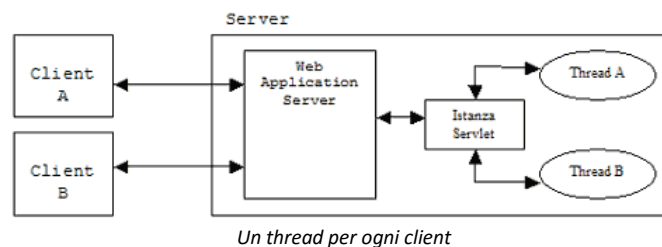
- **void doGet(HttpServletRequest req, HttpServletResponse resp)**
Gestisce le richieste HTTP di tipo GET. Viene invocato da service()
- **void doPost(HttpServletRequest req, HttpServletResponse resp)**
Gestisce le richieste HTTP di tipo POST. Viene invocato da service()
- **void service(HttpServletRequest req, HttpServletResponse resp)**
Viene invocato al termine del metodo
- **void doPut(HttpServletRequest req, HttpServletResponse resp)**
Viene invocato attraverso il metodo service() per consentire di gestire una richiesta HTTP di tipo PUT. Tipicamente, una richiesta del genere consente ad un client di inserire un file su un server, similmente ad un trasferimento di tipo FTP.

- **void doDelete(HttpServletRequest req, HttpServletResponse resp)**
Viene invocato attraverso il metodo service() per consentire ad una Servlet di gestire una richiesta di tipo HTTP DELETE. Questo genere di richiesta viene utilizzata per rimuovere un file dal server.
- **void init()**
Viene invocato soltanto una volta dal Servlet Engine al termine del caricamento della servlet ed appena prima che la servlet stessa inizi ad esaudire le richieste che le pervengono.
- **void destroy()**
È invocato direttamente dal Servlet Engine per scaricare una servlet dalla memoria.
- **String getServletInfo()**
È utilizzato per ricavare una stringa contenente informazioni di utilità sulla Servlet (ad es.: nome della Servlet, autore, copyright). La versione di default restituisce una stringa vuota.
- **ServletContext getServletContext()**
Viene usato per ottenere un riferimento all'oggetto di tipo ServletContext cui appartiene la Servlet.

VANTAGGI DELLE SERVLET

Qualcuno potrebbe domandarsi per quale motivo possa essere più vantaggioso utilizzare le Servlet piuttosto che affidarsi a tecnologie, ancora abbastanza utilizzate, come la Common Gateway Interface (CGI). Beh, le differenze non sono trascurabili. Tra le più evidenti possiamo citare:

- **Efficienza.** Come abbiamo detto le servlet vengono istanziate e caricate una volta soltanto, alla prima invocazione. Tutte le successive chiamate da parte di nuovi client vengono gestite creando dei nuovi thread che si prendono carico del processo di comunicazione (un thread per ogni client) fino al termine delle rispettive sessioni. Con le CGI, tutto questo non accadeva. Ogni client che effettuava una richiesta al server causava il caricamento completo del processo CGI con un degrado delle performance facilmente immaginabile.



- **Portabilità:** Grazie alla tecnologia Java, le servlet possono essere facilmente programmate e “portate” da una piattaforma ad un’altra senza particolari problemi.
- **Persistenza:** Dopo il caricamento, una servlet rimane in memoria mantenendo intatte determinate informazioni (come la connessione ad un data base) anche alle successive richieste.
- **Gestione delle sessioni:** Come è noto il protocollo HTTP è un protocollo stateless (senza stati) e, pertanto non in grado di ricordare i dettagli delle precedenti richieste provenienti da uno stesso client. Le servlet (lo vedremo in un altro articolo) sono in grado di superare questa limitazione.

ESEMPIO

Mettiamo in pratica le nozioni basilari che abbiamo finora enunciato e costruiamo una semplice Servlet il cui compito sarà quello di mostrare una tabella di conversione della temperatura da gradi Celsius a Fahrenheit.

```

import java.io.IOException;
import javax.servlet.*;
import java.text.DecimalFormat;
import java.io.PrintWriter;

public class CelsiusToFahrenheit extends HttpServlet
{
    private static final DecimalFormat FMT = new DecimalFormat("#0.00");
    private static final String PAGE_TOP = "<html>"
  
```

```

+ "<head>"
+ "<title>Tabella di conversione da Celsius a Fahrenheit</title>"
+ "</head>"
+ "<body>"
+ "<h3>Tabella di conversione da Celsius a Fahrenheit</h3>"
+ "<table>"
+ "<tr>"
+ "<th>Celsius</th>"
+ "<th>Fahrenheit</th>"
+ "</tr>"
;

private static final String PAGE_BOTTOM = ""
+ "</table>"
+ "</body>"
+ "</html>"
;

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println(PAGE_TOP);
    for(double cels = 15; cels <= 35; cels += 1.0)
    {
        Double fahre = (cels * 1.8) + 32;
        out.println("<tr>");
        out.println("<td>" + FMT.format(cels) + "</td>");
        out.println("<td>" + FMT.format(fahre) + "</td>");
        out.println("</tr>");
    }
    out.println(PAGE_BOTTOM);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    this.doGet(request, response);
}

public String getServletInfo()
{
    return super.getServletInfo();
}
}

```

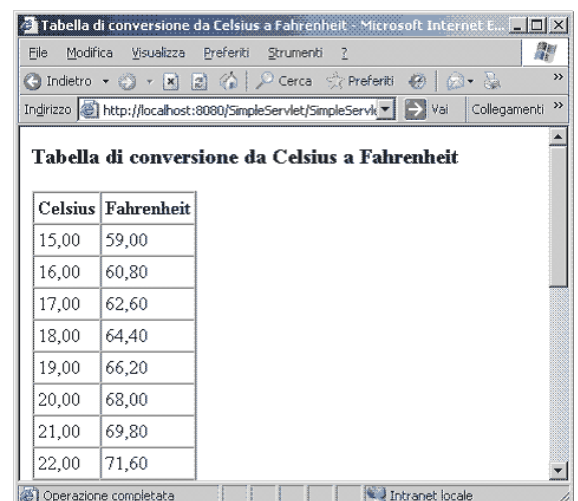
Abbiamo suddiviso la pagina HTML generata dalla servlet in 3 parti: La parte superiore (TOP), la parte inferiore (BOTTOM) ed infine, quella centrale. Le prime due sono rappresentate da stringhe statiche, definite come costanti (PAGE_TOP, PAGE_BOTTOM). La parte centrale, invece, è generata dinamicamente e gestita dal metodo doGet() della Servlet.

Si noti come, per comodità, sia stato fatto l'override anche del metodo doPost(), il cui codice si preoccupa semplicemente di inoltrare la richiesta al metodo doGet().

EFFETTUARE IL DEPLOY DELLA SERVLET SU TOMCAT

Per sapere come effettuare il deploy su Tomcat della Servlet mostrata nel nostro esempio, effettuato correttamente il deploy e digitando, sul browser l'indirizzo della servlet (<http://localhost:8080/SimpleServlet/SimpleServlet>) potremo visualizzare il contenuto web generato dalla nostra servlet.

A destra il risultato finale:



Celsius	Fahrenheit
15,00	59,00
16,00	60,80
17,00	62,60
18,00	64,40
19,00	66,20
20,00	68,00
21,00	69,80
22,00	71,60

JSP E SERVLET

Come abbiamo accennato le JSP affiancano le servlet che comunque giocano un ruolo fondamentale nella creazione di applicazioni Web.

Come nel nostro esempio, le servlet possono anche essere utilizzate per costruire pagine HTML da inviare direttamente al client. A differenza di uno script CGI le servlet vengono caricate una sola volta, al momento della prima richiesta, e rimangono residenti in memoria, pronte per servire le richieste fino a quando non vengono chiuse, con ovvi vantaggi in fatto di prestazioni (solo la prima richiesta risulta un po' più lenta nel caricamento, ma le successive vengono evase molto velocemente).

CICLO DI VITA DI UN SERVLET

Il ciclo di vita di una servlet (cioè il tempo che intercorre dal suo caricamento al suo rilascio) si concentra su tre metodi fondamentali: `init()`, `service()` e `destroy()`.

- **Init()**
Questo metodo viene chiamato una sola volta, subito dopo la sua istanziazione. È in questo metodo che la servlet istanzia tutte le risorse che utilizzerà poi per gestire le richieste
- **Service()**
Questo metodo è incaricato di gestire le richieste effettuate dal client, e ovviamente potrà iniziare il suo lavoro esclusivamente dopo la chiamata del metodo `init()`
- **Destroy()**
Questo metodo segna la chiusura della servlet, è qui che si effettuano eventuali salvataggi di informazioni utili ad un prossimo caricamento della servlet

Ulteriori approfondimenti su JSP e SERVLET:

<http://www.html.it/pag/15145/la-prima-pagina-jsp/>

<http://www.html.it/pag/16736/utilizzo-delle-pagine-jsp/>

<http://www.html.it/pag/16737/le-servlet1/>

TOMCAT

Tomcat è un software open source. La sua funzionalità di spicco è quella del Web Application Server, ovvero un server capace di gestire e supportare le pagine JSP e le servlet. È importante sapere che Tomcat è in grado di svolgere anche le veci di un Web Server, anche se limitato alla gestione di sole pagine statiche.

Abbiamo due possibilità per usarlo: o installiamo la full edition NetBeans o Tomcat + un ide a scelta

- Con netbeans: <http://technology.amis.nl/2012/01/02/installing-tomcat-7-and-configuring-as-server-in-netbeans/>
- Con altri ide: Installazione: <http://www.html.it/articoli/tomcat-lapplicazione-servita-1/> Avvio: <http://www.html.it/articoli/tomcat-lapplicazione-servita-2/>

COMPONENTI

Tomcat versione è composto da con Catalina (il contenitore di servlet), Coyote (il connettore HTTP) e Jasper (il motore JSP).

- **Catalina**: Catalina è il contenitore di servlet Java di Tomcat. Catalina implementa le specifiche di Sun Microsystems per le servlets Java e le "JavaServer Pages (JSP, Pagine JavaServer)". In Tomcat un elemento del Reame rappresenta un database di usernames, passwords e ruoli (analoghi dei gruppi di UNIX) assegnati a quegli utenti. Differenti implementazioni del Reame permettono a Catalina di essere integrato in ambienti dove tali informazioni di autenticazione sono già state create e supportate, e poi gli permettono di utilizzare tali informazioni per implementare una cosiddetta "Container Managed Security" come descritto nelle Specifiche delle Servlet.
- **Coyote**: Coyote è il componente "connettore HTTP" di Tomcat. Supporta il protocollo HTTP 1.1 per il web server o per il contenitore di applicazioni. Coyote ascolta le connessioni in entrata su una specifica porta TCP

sul server e inoltra la richiesta al Tomcat Engine per processare la richiesta e mandare indietro una risposta al client richiedente.

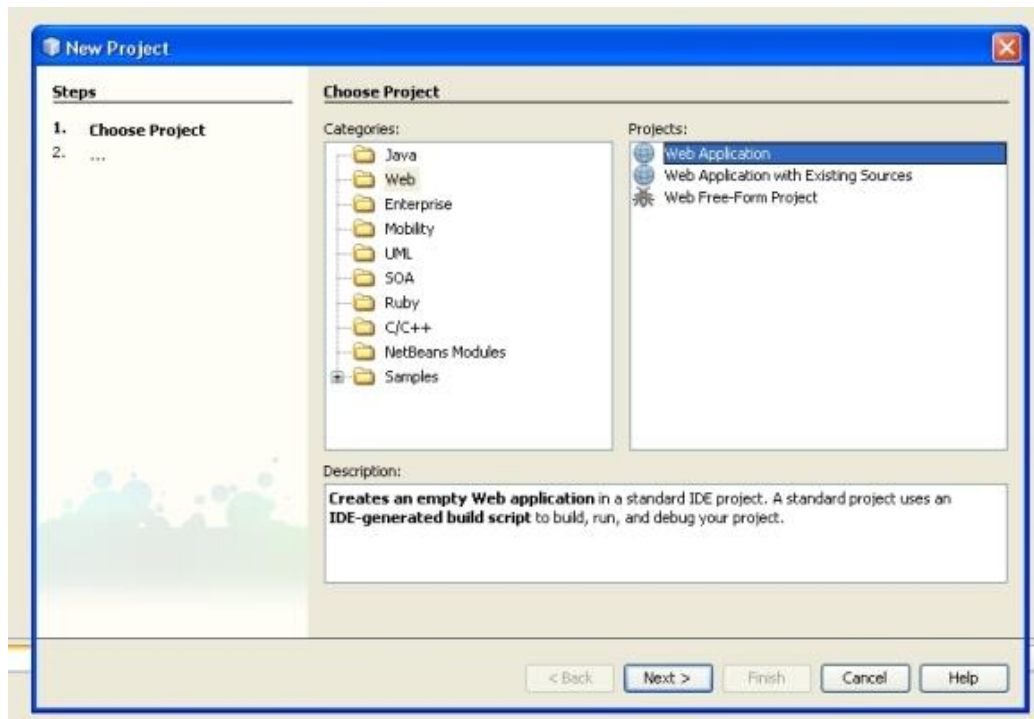
- Jasper: Jasper è il motore JSP di Tomcat. Jasper analizza i file JSP per compilarli in codice Java come servlets (che verranno poi gestite da Catalina). Al momento di essere lanciato, Jasper cerca eventuali cambiamenti avvenuti ai file JSP e, se necessario, li ricompila.

ESECUZIONE DI UNA SERVLET CON NETBEANS

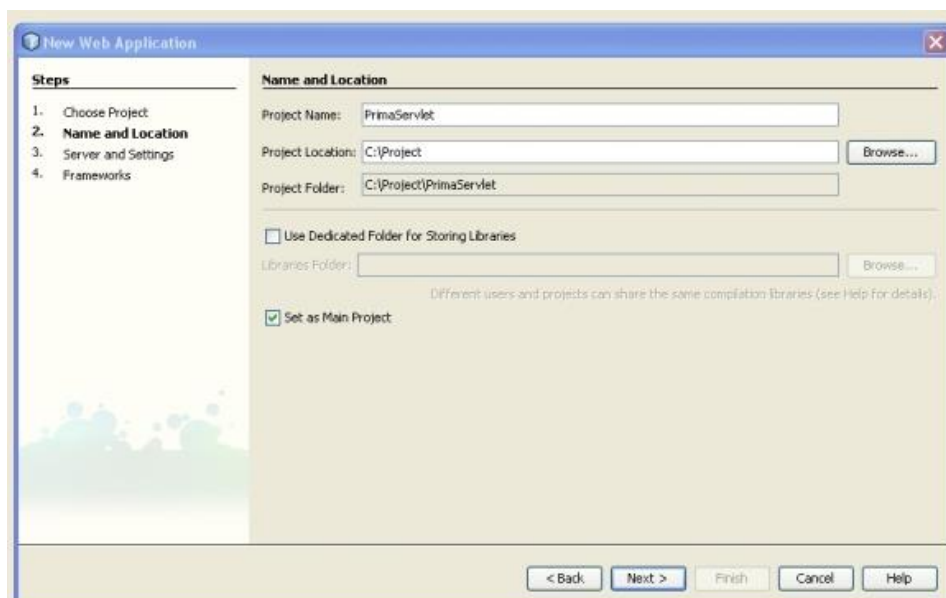
Vogliamo realizzare una servlet che ci comunica la data corrente su una pagina .xml e visualizzarla nel browser.

CREAZIONE PROGETTO

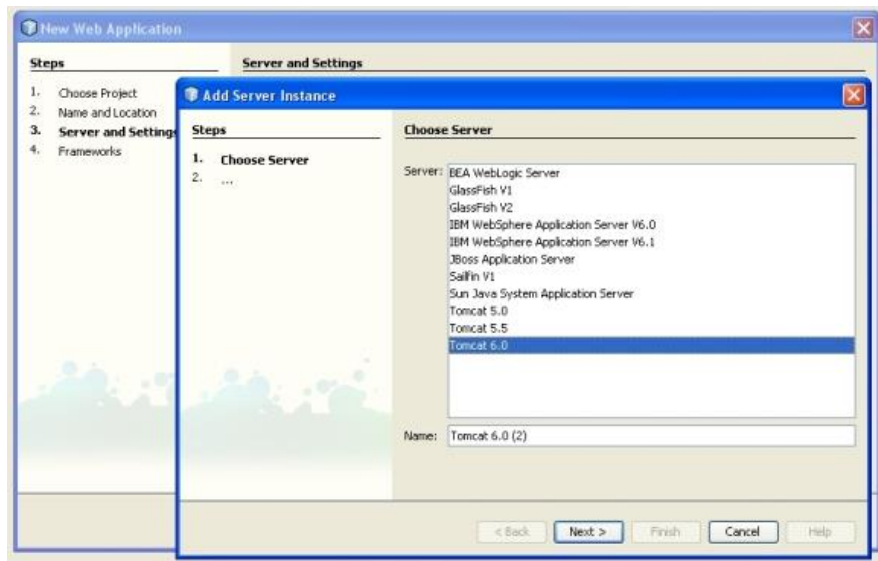
Una volta fatto partire Netbeans cliccate su file e successivamente su new project, quindi su web, web application e infine next.



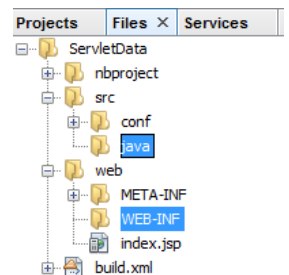
A questo punto si aprirà una finestra che vi chiede il nome del progetto e dove collocarlo sul disco. Nel nostro esempio abbiamo chiamato il progetto “PrimaServlet” e l’abbiamo impostato come progetto principale spuntando la casella apposita.



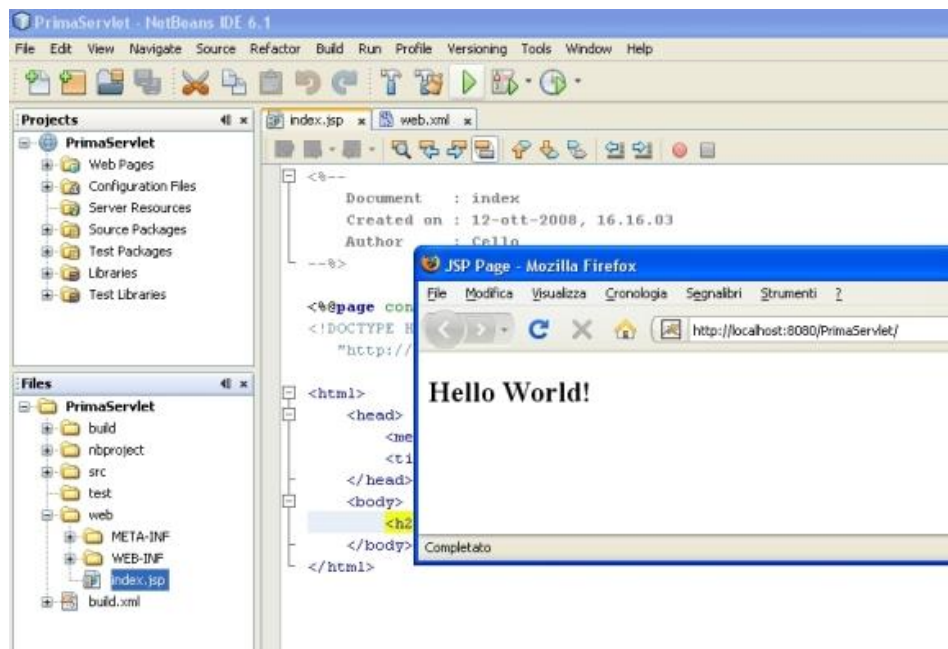
Ora si aprirà la finestra “*Server and Settings*” che serve a specificare che application server utilizzare e il percorso dove è presente sul nostro pc. Cliccate su Add quindi selezionate la versione di Tomcat che avete precedentemente scaricato e cliccate su next.



A questo punto NetBeans avrà creato tutto l'occorrente per permetterci di scrivere la nostra prima servlet. All'interno della cartella *web* inseriremo le nostre pagine *jsp*, all'interno della cartella *src/java* inseriremo i nostri *file.java* e all'interno della cartella *web/WEB-INF* sarà presente il file *web.xml* nel quale andranno specificati vari parametri di configurazione che permetteranno alla nostra servlet di funzionare come: il nome utilizzato per invocare la servlet (il suo alias), una descrizione della servlet, il percorso o i percorsi che fanno in modo che il contenitore di servlet invochi la servlet.

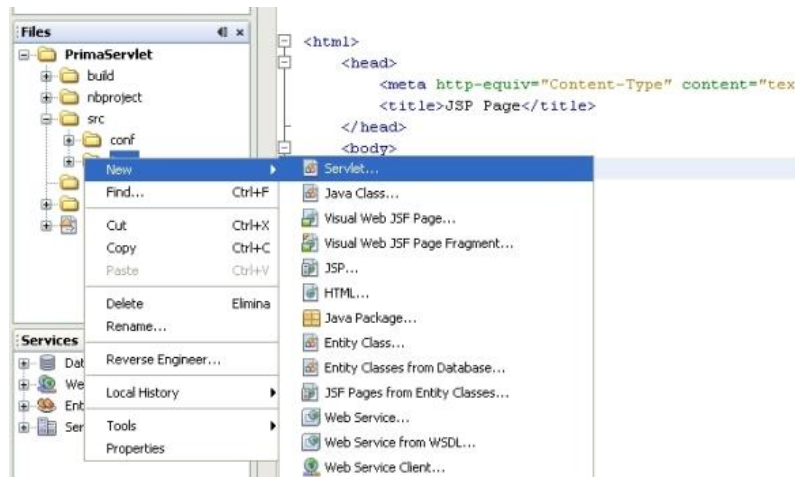


Per ora NetBeans non ha fatto altro che crearci una pagina jsp con scritto “Hello World!” che sarà invocata cliccando sul tasto “run main project” e la struttura per permetterci di lavorare.

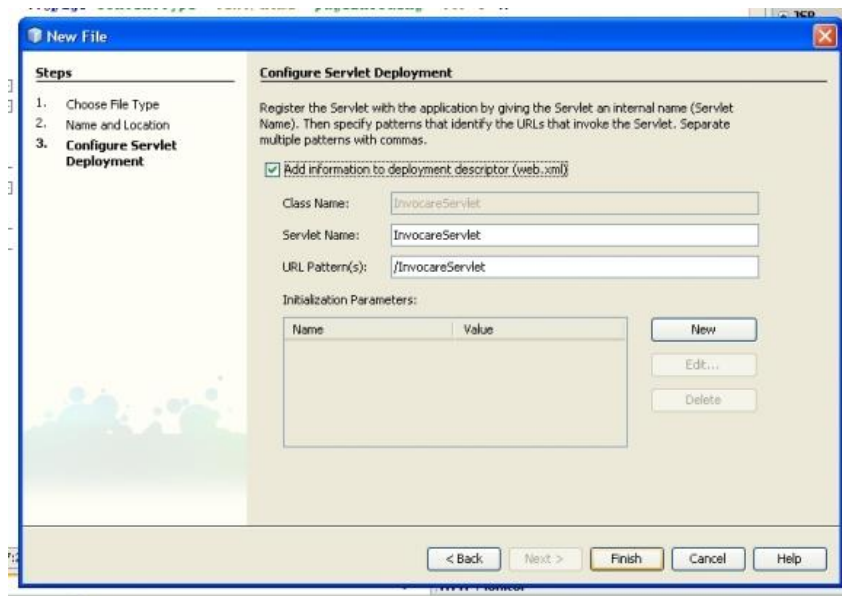


CREAZIONE SERVLET

Per creare la nostra prima servlet cliccate con il tasto destro sulla cartella *src/java* nella finestra file alla vostra sinistra, quindi su new e poi su servlet.



Come Class name specificate *“InvocareServlet”*, lasciate il resto inalterato e cliccate su next. A questo punto vi verranno chiesti il nome della servlet e l'url pattern che sono due parametri che NetBeans andrà a inserire nel file web.xml di cui abbiamo parlato prima e che vedremo successivamente in dettaglio. Specificate come nome servlet lo stesso nome che avete utilizzato per la classe e come url pattern /InvocareServlet che significa che l'indirizzo relativo che forniremo al browser per invocare la nostra servlet sarà PrimaServlet/InvocareServlet.



A questo punto NetBeans avrà creato un file InvocareServlet.java che è proprio la nostra servlet e avrà apportato delle modifiche al file web.xml.

Esaminiamo il file InvocareServlet.java

I servlet basati sul web tipicamente estendono la classe HttpServlet sovrascrivendone alcuni o tutti i suoi metodi. Infatti la prima riga di codice della nostra servlet è proprio **public class InvocareServlet extends HttpServlet {...** Che crea una nuova classe *“InvocareServlet”* che estende HttpServlet ereditandone i metodi.

I due tipi di richieste Http più comuni sono GET e POST rispettivamente per ottenere informazioni da un'indirizzo o per inviare informazioni al server (ad esempio i dati di una form). Per rispondere a tali richieste la classe HttpServlet delle servlet definisce i metodi **doGet** e **doPost** che ricevono come argomenti un'oggetto HttpServletRequest contenente la richiesta del client e un'oggetto HttpServletResponse contenente la risposta per il client.

Possiamo notare che all'interno del codice della nostra Servlet NetBeans ha fatto in modo da non fare distinzione tra i due metodi richiamando indistintamente il metodo processRequest da lui creato sia nel caso in cui sia stato invocato il metodo doGet che nel caso in cui sia stato invocato il doPost.

Infatti se guardiamo in dettaglio il codice i metodi **doGet** e **doPost** in questo modo quando vengono invocati richiamano il metodo **processRequest** passandogli l'oggetto `HttpServletRequest request` e `HttpServletResponse response`.

Esaminiamo ora il metodo `processRequest`

- **throws `ServletException`, `IOException`** significa che se `processRequest` non fosse in grado di gestire la richiesta del client lancerebbe una `javax.servlet.ServletException` e se `processRequest` incontrasse un'errore durante l'elaborazione del flusso di dati (leggendo dal client o scrivendo al client), lancierebbe una `java.io.IOException`.
- **`.setContentType("text/html;charset=UTF-8")`**; è un metodo dell'oggetto `response` che permette al browser di comprendere e gestire il contenuto della risposta per il client che in questo caso sarà una semplice pagina html.
- **`PrintWriter out = response.getWriter()`**; permette al server di inviare al client il documento html.

Le righe all'interno del try, invece, creano il documento html inviato al client; NetBeans le ha commentate quindi per fare in modo da visualizzare la pagina dobbiamo rimuovere `/* TODO output your page here */` e `/* */`.

Avevamo detto che all'interno del file `web.xml` contenuto nella cartella `web/WEB-INF` devono essere specificati i parametri di configurazione che permetteranno alla nostra servlet di funzionare.

Esaminiamo il contenuto del file `web.xml` modificato da NetBeans

l'elemento `servlet-name` è il nome che scegliamo per la servlet, l'elemento `servlet-class` specifica il nome completo della classe della servlet e l'elemento `servlet-mapping` indica l'url relativo che forniremo al browser per richiamare la nostra servlet.

La nostra prima servlet ora è pronta e il suo compito è quello di restituire una pagina html quando c'è la richiesta `PrimaServlet/InvocareServlet`; Per testare il suo funzionamento cliccate sul pulsante "run main project" nella parte alta di NetBeans. A questo punto dovrebbe aprirsi il vostro browser con il contenuto della pagina `index.jsp` contenuta nella cartella `web`; Se tutto è andato per il verso giusto richiamando **`PrimaServlet/InvocareServlet`** richiamerete la vostra servlet e quindi la pagina html presente all'interno del try nel metodo `processRequest`.



VISUALIZZAZIONE METODO

Se vogliamo visualizzare il metodo `data`, basta inserirlo nella servlet come segue:

```
import java.util.Date;

public Date getDate()
{
    return new Date();
}
```

E modifichiamo nel metodo `processRequest` la riga

```
out.println("<h1>Servlet InvocaData at " + request.getContextPath() + "</h1>");
```

In

```
out.println("<h1>Servlet InvocaData at " + request.getContextPath() + "<br>" + getDate() + "</h1>");
```

AZIONI

A questo punto potremmo complicare leggermente la servlet facendo in modo che questa riceva e stampi a video un parametro passatogli da una form inserita nell'`index.jsp`.

Il procedimento è molto semplice basterà inserire una form nell'index.jsp specificando come action della form la nostra servlet:

```
<form name="form1" action="InvocareServlet" method="Get">
  <input type="text" name="parametro" value="" />
  <input type="submit" value="ok" name="ok" />
</form>
```

e inserendo nel try della servlet la riga:

```
out.println(request.getParameter("parametro"));
```

che recupererà il parametro passatogli dalla form e lo stamperà a video.

WEB SERVICES

Un Web service è un componente applicativo. Possiamo definirlo come un sistema software in grado di mettersi al servizio di un'applicazione comunicando su di una medesima rete tramite il protocollo HTTP. Un Web service consente quindi alle applicazioni che vi si collegano di usufruire delle funzioni che mette a disposizione.

Es. Potremmo ipotizzare un Web service che chiameremo “*cambiavalute*”. Il nostro Web service fornisce le seguenti operazioni: cambio euro/dollaro e viceversa. Questo Web service potrebbe essere offerto da un istituto bancario ed una nostra applicazione potrebbe utilizzarlo per effettuare le operazioni di cambio senza doversi preoccupare dei tassi in vigore al momento dell'operazione.

Già dopo questo primo esempio dovrete aver notato che le operazioni svolte da un Web service non sono nulla di eclatante, qualsiasi comune applicazione potrebbe infatti effettuare l'operazione di cambio. Ciò che una comune applicazione però non può fare è **mettersi in comunicazione** con un altro software come ha fatto cambiavalute nel nostro esempio. Un Web service infatti comunica tramite protocolli e standard definiti “aperti” e quindi sempre a disposizione degli sviluppatori.

Un Web service è in grado di offrire un'interfaccia software assieme alla descrizione delle sue caratteristiche, cioè è in grado di farci sapere che funzioni mette a disposizione (senza bisogno di conoscerle a priori) e ci permette inoltre di capire come vanno utilizzate. Ciò significa che (sempre per rimanere al nostro esempio) con una semplice connessione a cambiavalute, anche senza conoscerlo, possiamo stabilire le operazioni che fornisce e possiamo subito iniziare ad usarle perchè ogni operazione ha una sua descrizione comprendente i parametri che si aspetta di ricevere, quelli che restituirà ed il tipo di entrambi.

DEFINIZIONE

Un Web Service (servizio web) è dunque un sistema software progettato per supportare l'interoperabilità (capacità di un sistema o di un prodotto informatico di cooperare e di scambiare informazioni o servizi con altri sistemi o prodotti in maniera più o meno completa e priva di errori, con affidabilità e con ottimizzazione delle risorse) tra diversi elaboratori su di una medesima rete ovvero in un contesto distribuito. Tale caratteristica si ottiene associando all'applicazione un'interfaccia software che espone all'esterno i servizi associati; altri sistemi possono interagire con l'applicazione stessa attivando le operazioni descritte nell'interfaccia (servizi o richieste di procedure remote) tramite appositi “messaggi” di richiesta: tali messaggi di richiesta sono inclusi in una “busta” (la più famosa è SOAP), formattati secondo lo standard XML, incapsulati e trasportati tramite i protocolli del Web (solitamente HTTP), da cui appunto il nome web service.

Proprio grazie all'utilizzo di standard basati su XML, tramite un'architettura basata sui Web Service (chiamata, con terminologia inglese, *Service oriented Architecture - SOA*) applicazioni software scritte in diversi linguaggi di programmazione e implementate su diverse piattaforme hardware possono quindi essere utilizzate, tramite le interfacce che queste “espongono” pubblicamente e mediante l'utilizzo delle funzioni che sono in grado di effettuare (i “servizi” che mettono a disposizione) per lo scambio di informazioni e l'effettuazione di operazioni complesse (quali, ad esempio, la realizzazione di processi di business che coinvolgono più aree di una medesima azienda) sia su reti aziendali come anche su Internet: la possibilità dell'interoperabilità fra diversi linguaggi di programmazione (ad esempio, tra Java e Python) e diversi sistemi operativi (come Windows e Linux) è resa possibile dall'uso di standard “aperti”.

COME FUNZIONANO

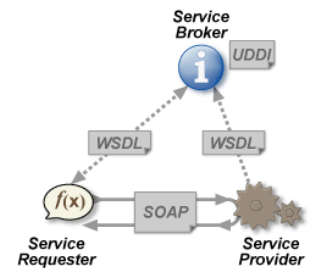
Ora che abbiamo capito cosa sono e a cosa servono i servizi web, cerchiamo di capire meglio come funzionano. Il protocollo di base per i Web service è HTTP. Questo protocollo si occupa di mettere in comunicazione il servizio web con l'applicazione che intende usufruire delle sue funzioni.

Oltre ad HTTP però, i servizi web utilizzano molti altri standard web, tutti basati su XML, tra cui:

- **XML Schema** - Serve per definire qual è la costruzione legale di un documento XML ([approfondimenti](#)).
- **UDDI** (Universal Description, Discovery and Integration) - è un registry (ovvero una base dati ordinata e indicizzata), indipendente dalla piattaforma hardware, che permette alle aziende la pubblicazione dei propri dati e dei servizi offerti su internet ([approfondimenti](#)).

- **WSDL** (Web Services Description Language) - Questo linguaggio serve a specificare dove si trovano i servizi e le operazioni esposte dal servizio web ([approfondimenti](#)).
- **SOAP** (Simple Object Access Protocol) – Le informazioni definite da WSDL vengono scambiate tra il Web service ([approfondimenti](#)).

È importante sottolineare che XML può essere utilizzato correttamente tra piattaforme differenti (Linux, Windows, Mac) e differenti linguaggi di programmazione. XML è inoltre in grado di esprimere messaggi e funzioni anche molto complesse e garantisce che tutti i dati scambiati possano essere utilizzati ad entrambi i capi della connessione. Si può quindi dire che i Web service sono basati su XML ed HTTP e che possono essere utilizzati su ogni piattaforma e con ogni tipo di software.



CARATTERISTICHE DEI WEB SERVICES

Alcuni dei vantaggi che è possibile ottenere con l'utilizzo dei Web Service sono i seguenti:

- I Web service permettono l'interoperabilità tra diverse applicazioni software e su diverse piattaforme hardware/software
- Utilizzano un formato dei dati di tipo testuale, quindi più comprensibile e più facile da utilizzare per gli sviluppatori (esclusi ovviamente i trasferimenti di dati di tipo binario)
- Normalmente, essendo basati sul protocollo HTTP, non richiedono modifiche alle regole di sicurezza utilizzate come filtro dai firewall
- Sono semplici da utilizzare e possono essere combinati l'uno con l'altro (indipendentemente da chi li fornisce e da dove vengono resi disponibili) per formare servizi "integrati" e complessi
- Permettono di riutilizzare applicazioni già sviluppate.
- Fintanto che l'interfaccia rimane costante, le modifiche effettuate ai servizi rimangono trasparenti
- I servizi web sono in grado di pubblicare le loro funzioni e di scambiare dati con il resto del mondo
- Tutte le informazioni vengono scambiate attraverso protocolli aperti (open source)

Di contro vi sono i seguenti aspetti da considerare:

- Attualmente non esistono standard consolidati per applicazioni critiche quali, ad esempio, le transazioni distribuite
- Le performance legate all'utilizzo dei Web Service possono essere minori di quelle riscontrabili utilizzando approcci alternativi di distributed computing quali Java RMI, CORBA, o DCOM. Questo svantaggio è legato alla natura stessa dei servizi web. Essendo basati su XML ogni trasferimento di dati richiede l'inserimento di un notevole numero di dati supplementari (i tag XML) indispensabili per la descrizione dell'operazione. Inoltre tutti i dati inviati richiedono di essere prima codificati e poi decodificati ai capi della connessione. Queste due caratteristiche dei Web service li rendono poco adatti a flussi di dati intensi o dove la velocità dell'applicazione rappresenta un fattore critico.
- L'uso dell'HTTP permette ai Web Service di evitare le misure di sicurezza dei firewall (le cui regole sono stabilite spesso proprio per evitare le comunicazioni fra programmi "esterni" ed "interni" al firewall). Quando si sviluppa un Web service è necessario tener conto del protocollo di base. È quindi indispensabile disporre di un'applicazione terza che gestisca le richieste HTTP oppure è necessario includerla direttamente nel codice del nostro programma qualora si desideri la sua totale indipendenza. Va detto comunque che generalmente il codice che implementa un Web service viene fatto eseguire da un Web server (es. Apache) tramite CGI (per es. con Python) o tramite appositi moduli (vedi PHP). Eseguendo il codice del Web service attraverso un server web la gestione di HTTP è immediatamente assicurata.

PERCHÉ CREARE UN WEB SERVICE

La ragione principale per la creazione e l'utilizzo di Web Service è il "*disaccoppiamento*" che l'interfaccia standard esposta dal Web Service rende possibile fra il sistema utente ed il Web Service stesso: modifiche ad una o all'altra delle applicazioni possono essere attuate in maniera "trasparente" all'interfaccia tra i due sistemi; tale flessibilità consente la creazione di sistemi software complessi costituiti da componenti svincolati l'uno dall'altro e consente una forte riusabilità di codice ed applicazioni già sviluppate.

Come accennato, i Web service hanno inoltre guadagnato consensi visto che, come protocollo di trasporto, possono utilizzare HTTP "over" TCP sulla porta 80; tale porta è, normalmente, una delle poche (se non l'unica) lasciata "aperta" dai sistemi firewall al traffico di entrata ed uscita dall'esterno verso i sistemi aziendali e ciò in quanto su tale porta transita il traffico HTTP dei web browser: ciò consente l'utilizzo dei Web Service senza modifiche sulle

configurazioni di sicurezza dell'azienda (un aspetto che se da un lato è positivo solleva preoccupazioni concernenti la sicurezza).

PILA PROTOCOLLARE DEI WEB SERVICE

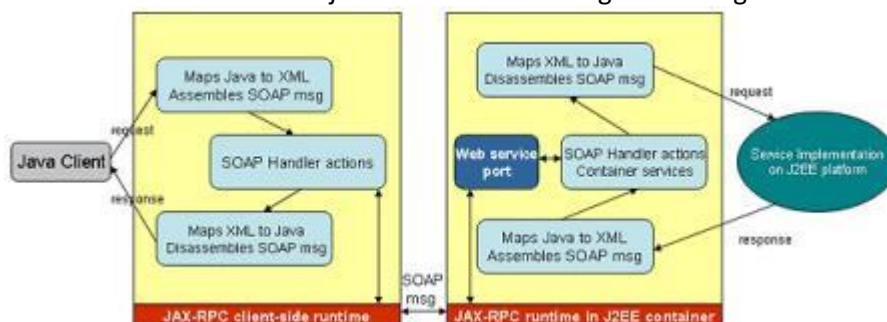
La pila protocollare dei Web Service è l'insieme dei protocolli di rete utilizzati per definire, localizzare, realizzare e far interagire tra di loro i Web Service; è principalmente composta di quattro aree:

- Trasporto del servizio: responsabile per il trasporto dei messaggi tra le applicazioni in rete, include protocolli quali HTTP, SMTP, FTP, XMPP ed il recente Blocks Extensible Exchange Protocol (BEEP).
- XML Messaging: tutti i dati scambiati sono formattati mediante "tag" XML in modo che gli stessi possano essere utilizzati ad entrambi i capi delle connessioni; il messaggio può essere codificato conformemente allo standard SOAP, come anche utilizzare JAX-RPC, XML-RPC o REST.
- Descrizione del servizio: l'interfaccia pubblica di un Web Service viene descritta tramite WSDL (Web Services Description Language) un linguaggio basato su XML usato per la creazione di "documenti" descrittivi delle modalità di interfacciamento ed utilizzo del Web Service.
- Elencazione dei servizi: la centralizzazione della descrizione e della localizzazione dei Web Service in un "registro" comune permette la ricerca ed il reperimento in maniera veloce dei Web Service disponibili in rete; a tale scopo viene attualmente utilizzato il protocollo UDDI.

REALIZZAZIONE DI UN WEB SERVICE JAVA

Realizzare un web service in Java non è una procedura particolarmente complicata. Grazie alla presenza di tool che supportano questa operazione un web container come Tomcat può esporre in maniera piuttosto semplice un servizio sotto forma di web service.

Il funzionamento di un web service in ambiente java è illustrato dalla figura che segue:



Focalizziamo l'attenzione sulla parte sinistra (lato server). L'implementazione del servizio è indipendente dal modo in cui esso verrà erogato, quindi potrà essere prevista semplicemente una classe java con relativi metodi. Il descrittore WSDL definisce in che modo si dovrà accedere alla classe che fornisce il servizio. Tutto quello che sta in mezzo (trasformazione del messaggio SOAP in comando java e viceversa) è a carico del middleware, che creerà l'opportuna infrastruttura.

Infatti, compito dello sviluppatore sarà: creare il servizio, definire il descrittore ed installare il tutto sulla macchina server. Lato client, la procedura sarà analoga. Sarà sufficiente conoscere l'indirizzo del descrittore WSDL e costruire attorno ad esso i giusti messaggi SOAP (lo farà sempre uno strato di middleware).

Tralasciando la procedura di pubblicazione di un servizio, quello che serve alla creazione è la definizione della logica (anche una semplice classe), un descrittore WSDL e l'installazione su un web container predisposto a pubblicare web services. Il middleware si occuperà di configurare opportunamente il servizio per essere accessibile.

In generale, il fatto di poter usufruire di tutti i servizi messi a disposizione dell'application server garantisce ottime performance al servizio creato. La gestione delle transazioni, l'eventuale accesso a sorgenti dati, le politiche di sicurezza, l'eventuale scalabilità, sono tutte caratteristiche curate dall'application server. In molti casi reali, soprattutto parlando di web services, acceduti da diverse organizzazioni distribuite geograficamente, questo è quanto meno necessario.

REALIZZAZIONE DI WEB SERVICES CON NETBEANS

<https://netbeans.org/kb/docs/websvc/jax-ws.html>