

JEE+
SPRING

JEE

Storia del web

Inizialmente i protocolli per il web erano in grado di gestire solo pagine HTML statiche.

Cosa si intende per pagine HTML statiche? File ipertestuali già pronti accessibili tramite browser web.

Applicazioni client/server

E prima del web?

Prima del web le applicazioni in rete erano gestite attraverso il modello client/server.

Nell'architettura client/server uno o più computer (chiamati terminali) si connettono ad un server che eroga un determinato servizio.

Applicazioni client/server

- ▶ Quali sono i limiti di questa architettura?
 - necessità di installare un software (la componente client) su ciascun terminale
 - impossibilità di accedere da terminali con sistemi operativi differenti
(a meno dello sviluppo di un software client per ogni sistema operativo...)

Standard web

I principali standard che vengono utilizzati nel web sono:

- **HTML**: è un linguaggio (XML based) con cui vengono scritte le pagine web;
- **HTTP**: è il protocollo di rete appartenente al 7° livello del modello ISO/OSI su cui è basato il Web (livello di applicazione);
- **URL (Uniform Resource Locator)**: stringa di caratteri che identifica univocamente una risorsa (pagina web, video, immagine, ...) in internet, disponibile su un server web.

Da dove deriva il nome ISO/OSI?

L'Organizzazione internazionale per la normazione (ISO) nel 1977 diede inizio, assieme all'Unione internazionale delle telecomunicazioni ITU-T, ad un'operazione di standardizzazione delle reti di calcolatori chiamata Open Systems Interconnection (OSI).

Cos'è un protocollo

Il protocollo è un insieme di regole che consentono di definire uno standard di comunicazione tra diversi computer attraverso la rete.

Per rete si intende un insieme di due o più computer connessi tra di loro, in grado di condividere informazioni.

I vari computer, per potersi scambiare informazioni, devono avere delle regole chiare (i protocolli) che consentano di attribuire ad un determinato comando un significato univoco.

La connessione ad internet è regolata dalla famiglia di protocolli TCP/IP

Cos'è un protocollo

Un protocollo descrive:

- il **formato del messaggio**
- il **modo in cui devono essere scambiati i messaggi** tra gli attori (i computer)

Per ogni servizio erogato tramite la rete (spedire email, stabilire connessioni remote, trasferire file...) esiste un protocollo.

Pensiamo ad un'email: il formato del messaggio ed il modo in cui viaggia attraverso la rete sono regolati da un protocollo che garantisce la corretta formattazione e trasmissione dal mittente (il computer dell'utente) al destinatario (il server di posta) e viceversa.

Cos'è un protocollo

Alcuni esempi di protocolli utilizzati:

- **Simple Mail Transfer Protocol (SMTP)** - per la gestione dei messaggi di posta elettronica
- **File Transfer Protocol (FTP)** - per il trasferimento di files tra macchine remote
- **Hypertext Transfer Protocol (HTTP)** - per la trasmissione di informazioni attraverso il WEB

La pila protocollare TCP/IP

Il TCP/IP è l'insieme dei protocolli di trasmissione usati per lo scambio di dati in rete.

Tutto ruota attorno ai protocolli fondamentali:

- TCP e UDP che organizzano la suddivisione (frammentazione) in pacchetti dei dati da inviare
- IP che gestisce l'instradamento dei pacchetti dal server al client e viceversa

Alla base di tutto ci sono due computer fisici dotati di una scheda di rete alla quale è assegnato un indirizzo IP pubblico.

Application Layer
(protocollo HTTP)

Transport Layer
(protocolli TCP e UDP)

Internet Layer
(protocollo IP)

Network Access Layer
(protocolli Ethernet, PPP)

Il protocollo TCP

Il protocollo TCP (Transmission Control Protocol) è il protocollo usato comunemente su Internet.

Quando viene richiesta una risorsa al server web, il client invia dei pacchetti TCP, richiedendo la risorsa. Il web server risponde inviando un flusso di pacchetti TCP.

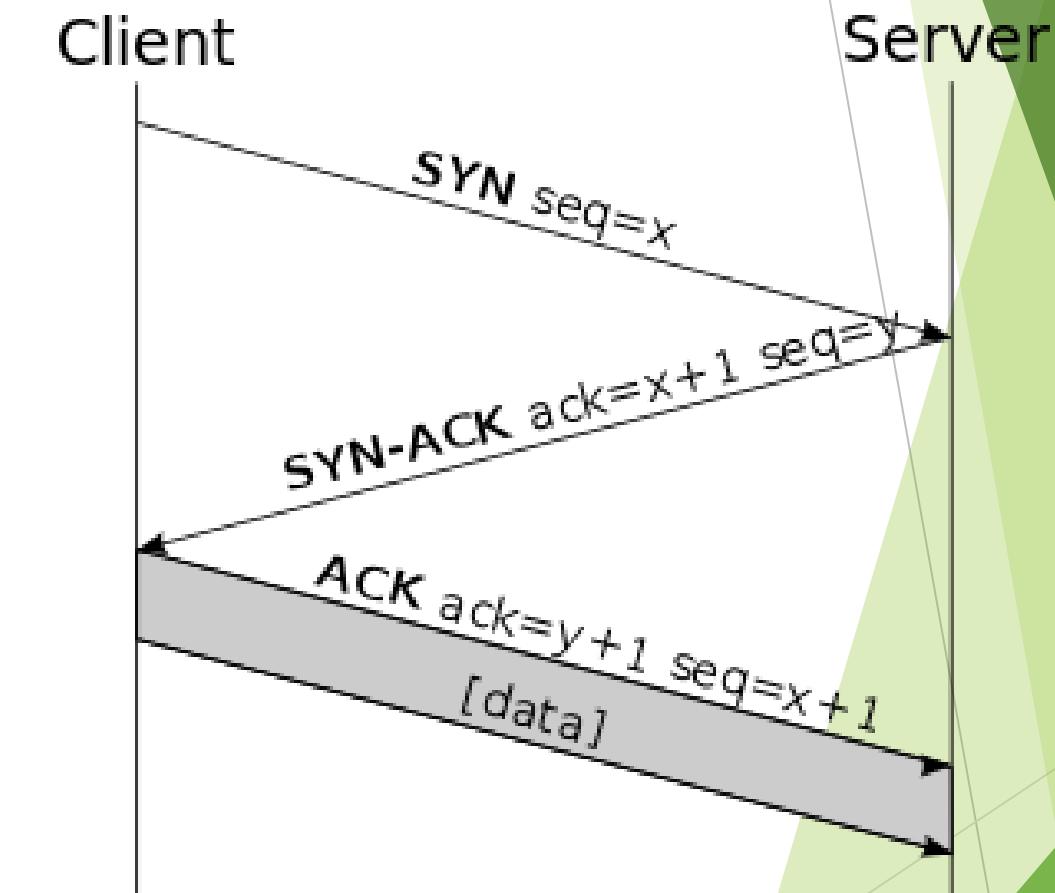
Il client mette i pacchetti ricevuti e visualizza la risorsa all'utente (pagina web, immagine, file scaricato,...).

Il protocollo TCP garantisce che il client riceva i tutti pacchetti senza perdita o danneggiamento.

Il protocollo TCP

Per aprire una connessione TCP usa uno schema detto Three Way Handshake.

- *Il Client invia un segmento SYN al Server*
- *Il Server invia un segmento SYN/ACK al Client*
- *Il Client invia un segmento ACK al Server*
- *A questo punto la connessione è stabilita*



Durante l'handshake, solitamente vengono inviati solo i segmenti dell'header, quindi il campo Data è vuoto.

Il protocollo UDP

Il protocollo UDP (User Datagram Protocol), è simile al protocollo TCP, solo che UDP non controlla gli errori.

Con il protocollo UDP i pacchetti vengono inviati al destinatario velocemente senza attendere che il destinatario li abbia ricevuti.

Il protocollo UDP non dà alcuna garanzia di ricezione dei dati, però la comunicazione è più veloce.

UDP è utilizzato generalmente per lo streaming video, per i giochi online, etc...

Le porte del protocollo TCP/IP

- ▶ Le porte sono uno strumento fondamentale per consentire ai protocolli TCP e UDP di gestire flussi di dati multipli attraverso un'unica connessione fisica alla rete (cioè attraverso un cavo di rete connesso ad un'unica scheda di rete...).
- ▶ Le porte sono numeri usati per identificare una connessione di trasporto tra quelle attive su un computer.
- ▶ La porta serve per identificare l'applicazione a cui instradare il flusso dati, consentendo al computer di eseguire molteplici applicazioni, con la garanzia che il flusso dati in ingresso ed uscita venga instradato correttamente.

Le porte del protocollo TCP/IP

Facendo un paragone con la vita reale...una porta può essere associata al numero di un conto corrente bancario. Una banca ha molteplici clienti e ciascun cliente ha il suo conto corrente.

Se voglio fare un bonifico ad un cliente della banca, oltre a specificare le coordinate della banca (che costituiscono una prima parte del codice IBAN), devo specificare anche il numero del conto corrente a sul quale accreditare i soldi (l'altra parte del codice IBAN).

Una cosa analoga si ha nelle comunicazioni in rete attraverso il protocollo TCP/IP.

Un computer (la banca) eroga diversi servizi (i conti correnti). Il computer connesso alla rete un indirizzo IP (la prima parte del codice IBAN che identifica la banca) ed ogni servizio è in ascolto su una porta (il numero di conto corrente).

Le porte del protocollo TCP/IP

L'Internet Assigned Numbers Authority (IANA) è un ente che si occupa di standardizzazione delle porte e dell'aggiornamento di un documento, il **ports-number**, che contiene l'elenco dei servizi e delle relative porte utilizzate.

In questo documento, le 65536 porte UDP e TCP è stato suddiviso in tre parti:

Well Known Ports (porta 0 – 1023) – sono riservate ai **servizi server** standard (ad es. la porta 80 è riservata al protocollo HTTP).

Registered Ports (porta 1024 – 49151) – sono associate ad alcuni servizi registrati (ad es. la porta 8080 è associata a Tomcat o come alternativa alla 80) ma il loro utilizzo è libero.

Dynamic and/or Private Ports (porta 49152 – 65535) – non sono associate a servizi e possono essere utilizzate liberamente.

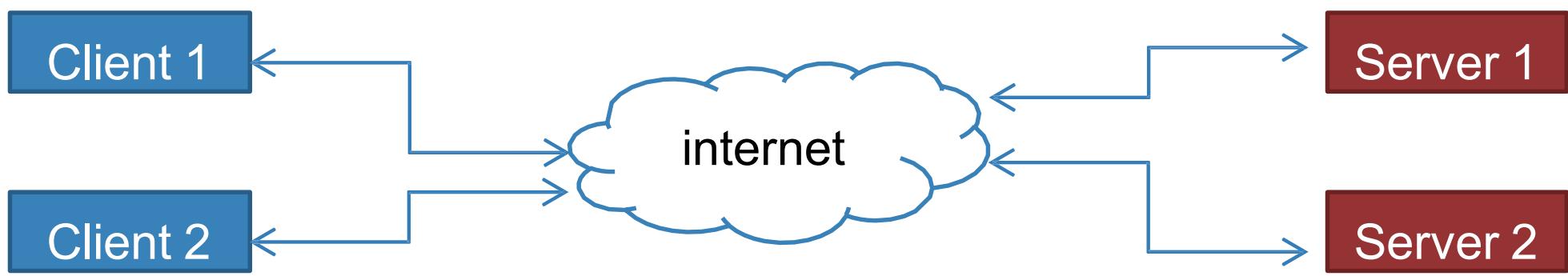
Le porte del protocollo TCP/IP

Esempi di porte e relative protocolli

- 21 FTP (File Transfer Protocol)
- 22 SSH (Secure Shell)
- 25 SMTP (Simple Mail Transfer Protocol)
- 80 HTTP (Hyper Text Transfer Protocol)
- 110 POP3 (Post Office Protocol 3)
- 143 IMAP (Internet Message Access Protocol)
- 443 HTTPS (Secure HTTP)

Il protocollo HTTP

- ▶ Il protocollo HTTP (HyperText Transfer Protocol) è un protocollo (a livello applicativo) usato per la trasmissione di informazioni sul web.
- ▶ HTTP è stato creato apposta per gestire il trasferimento di informazioni in formato HTML (HyperText Marked Language).



I client (i browser) ed i server (i web server o HTTP server) devono essere in grado di gestire il protocollo HTTP e possono scambiarsi i documenti ipermediati.

Il protocollo HTTP

Il protocollo HTTP è un protocollo "stateless" (cioè senza memoria).

Stateless vuol dire che il protocollo HTTP non conserva memoria della connessione fatta.

Ad ogni richiesta, il protocollo HTTP effettua una nuova connessione al server che viene chiusa al termine del trasferimento dell'oggetto richiesto (pagina HTML, immagine, ecc.).



Il protocollo HTTP

Come funziona il protocollo HTTP?

L'HTTP è basato sul meccanismo richiesta/risposta tra client e server.

- il **client** (tipicamente un browser) esegue una richiesta
- il **server** (il web server) restituisce la risposta

Nel protocollo HTTP abbiamo quindi due tipi di messaggi:

- **messaggi richiesta** (o HTTP request), inviato dal client al server
- **messaggi risposta** (o HTTP response), inviato dal server al client

Il protocollo HTTP

Il messaggio di richiesta (la HTTP request) è composto di tre parti:

- **request line**: composta da *metodo* (GET - POST -...), *URI* (uniform resource identifier) che indica l'oggetto della richiesta e *versione del protocollo*
- **header**: contiene una serie di informazioni, tra cui Host (server a cui si riferisce l'URL), User-Agent (browser, versione, ...)
- **body**: contiene il corpo del messaggio

Il protocollo HTTP

Il metodo **GET** consente di ottenere il contenuto della risorsa associata all'URI

<http://www.miosito.it/immagine.png>

<http://www.miosito.it/pagina1.html>

<http://www.miosito.it/pagina2.html?q=text¶m=aa>

Il metodo **POST** è usato generalmente per inviare informazioni al server, tipicamente i dati di un form.

Quando utilizziamo il metodo POST, l'URI indica cosa si sta inviando e il body ne indica il contenuto.

<http://www.miosito.it/pagina2.html>

(i dati **text** e **aa** saranno inviati nel body)

Il protocollo HTTP

Il messaggio di risposta (la HTTP response) è composto da tre parti:

- **status-line** (o **status-code**): contiene un *codice a tre cifre* che indica l'esito della ricezione della richiesta;
- **header**: contiene una serie di informazioni, tra cui **server** che indica tipo e versione del server e **content-type** che indica il tipo di contenuto restituito (o MIME - Multimedia Internet Mail Extensions).
- **body**: contiene il contenuto della risposta.

Il protocollo HTTP

Gli status-code più comuni sono:

- **200 OK.** La richiesta è stata ricevuta, capita e accettata.
- **301 Moved Permanently.** La risorsa richiesta non è raggiungibile perché è stata spostata in modo permanente.
- **400 Bad Request.** La risorsa richiesta non è comprensibile al server.
- **404 Not Found.** La risorsa richiesta non è stata trovata dal server.
- **500 Internal Server Error.** Si è verificato un problema interno ed il server non è in grado di rispondere alla richiesta (generalmente si ha quando si verifica un errore applicativo).

Cos'è e come funziona un'applicazione web

Joomla

WordPress

Moodle

Prestashop

WooCommerce

Liferay Portal

...

Ticketone

Sistemi di pagamento

Testate giornalistiche

e-formare

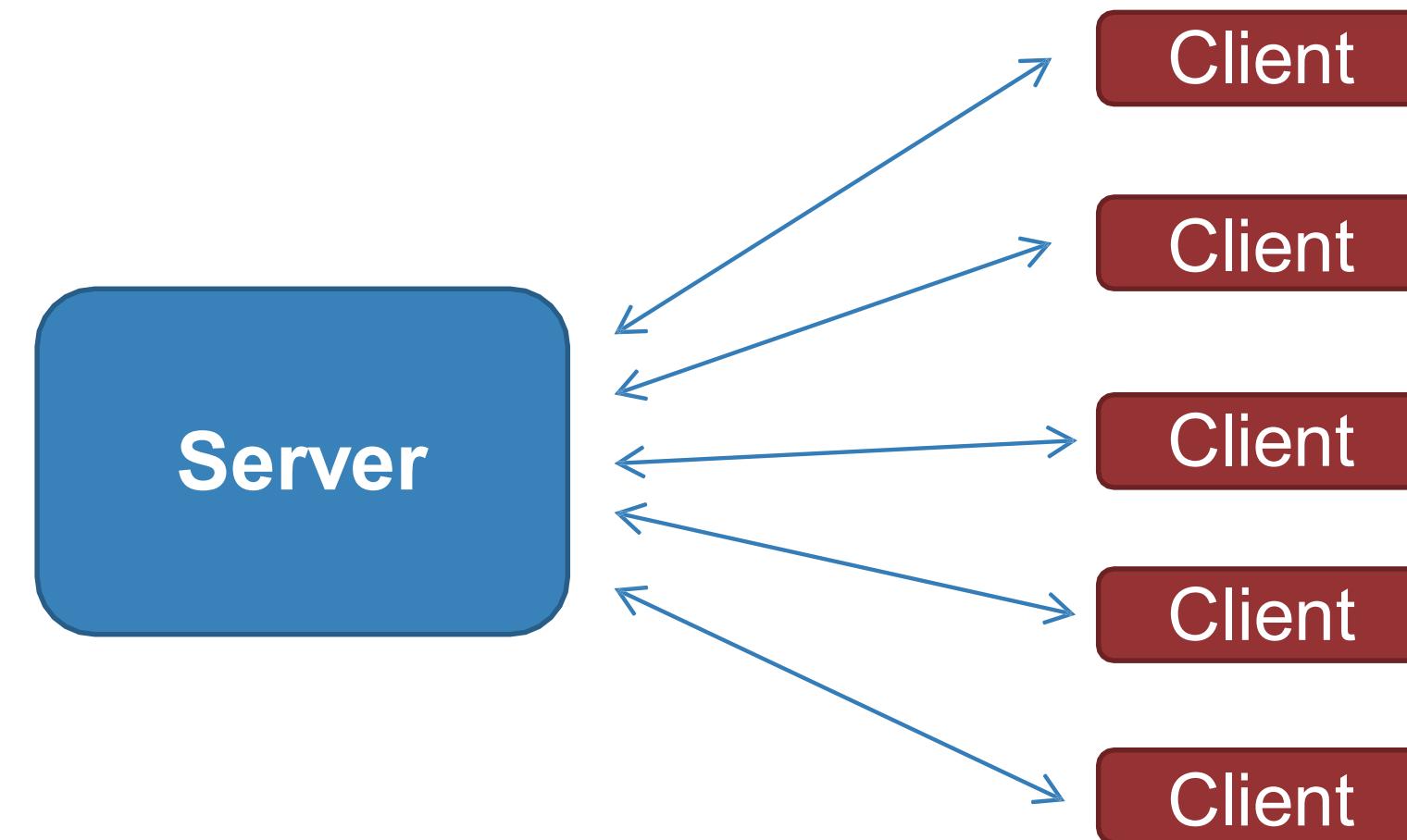
Home Banking

Google Docs

Questi sono alcuni esempi di web application

Cos'è e come funziona un'applicazione web

- ▶ Un'applicazione web (o web application) è un software accessibile via web.
- ▶ L'accesso può avvenire in una rete privata (intranet) o in internet.
- ▶ Le prime applicazioni web generavano pagine standard HTML/XHTML.
- ▶ Con l'evolversi delle tecnologie e con la nascita di nuovi standard, iniziarono a diffondersi i documenti in formati più "neutri", come l'XML.



Applicazione web vs Sito statico

Sito statico

Le pagine sono già pronte e risiedono sul server.

Non sono possibili interazioni significative tra l'utente e le pagine web. Le uniche interazioni sono la consultazione e la navigazione tra le pagine.

Un sito statico non è in grado di elaborare dati inseriti dall'utente attraverso i form!

Applicazione web vs Sito statico

Sito statico

Ogni aggiornamento (cambia un'immagine, cambia la grafica, ...) deve essere effettuato manualmente nel file html.

Il linguaggio html non consente di effettuare import di altre pagine.

Se un sito è composto da 200 pagine html statiche, ogni cambiamento deve essere fatto manualmente su tutti i file che compongono il sito.

Applicazione web vs Sito statico

Sito dinamico

Le pagine vengono generate dinamicamente.

Per ogni aggiornamento (cambia un testo nel banner laterale, cambia la grafica, ...) non è necessario ripetere le operazioni per ogni pagina.

Il sito dinamico è in grado di elaborare i dati inviati dall'utente attraverso i form!

I linguaggi di programmazione utilizzati per creare siti dinamici (tra cui Java), consentono di effettuare import di altre pagine (tipicamente header, footer e singoli blocchi che si possono riutilizzare nel sito).

Applicazione web vs Sito statico

Vantaggi Sito statico

- Velocità di visualizzazione delle pagine

Svantaggi Sito statico

- Difficoltà di manutenzione
- Impossibile creare applicazioni complesse

Vantaggi Applicazione web

- Manutenzione più facile
- Possibilità di creare applicazioni complesse

Svantaggi Applicazione web

- Velocità di visualizzazione delle pagine

Client web

Il web browser è un software che consente l'accesso ad applicazioni web ed altre risorse (pagine web statiche, immagini o video).

Il browser svolge due compiti:

- componente client del protocollo HTTP, gestisce il download delle risorse dal server web;
- visualizzazione dei contenuti ipertestuali e riproduzione di quelli multimediali.

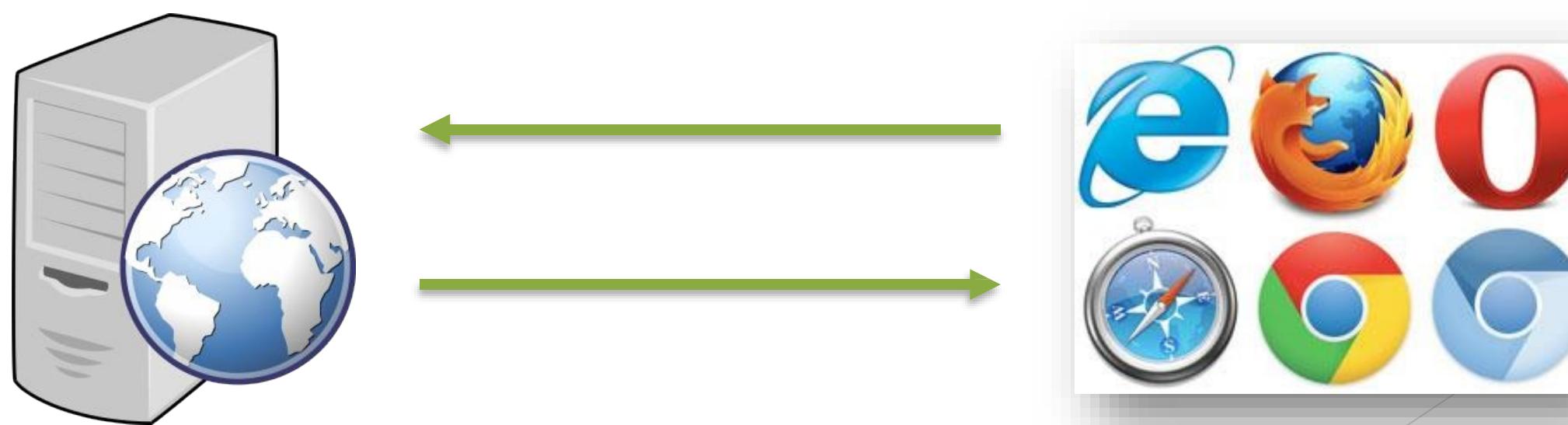


Server web

Un server web (o web server) è un software all'interno del quale vengono installate le applicazioni web.

Il server web gestisce le richieste di pagine web effettuate da un client (generalmente web browser).

La comunicazione tra web server e client avviene tramite il protocollo HTTP.



Server web

Il protocollo HTTP utilizza la porta TCP (Transmission Control Protocol) 80.

Alcune volte viene utilizzata anche la porta 8080.

In caso di utilizzo del protocollo HTTP sicuro (HTTPS) viene utilizzata la porta 443.

Come funziona un'applicazione web

Dalla richiesta alla risposta

L'accesso ad una risorsa web può iniziare:

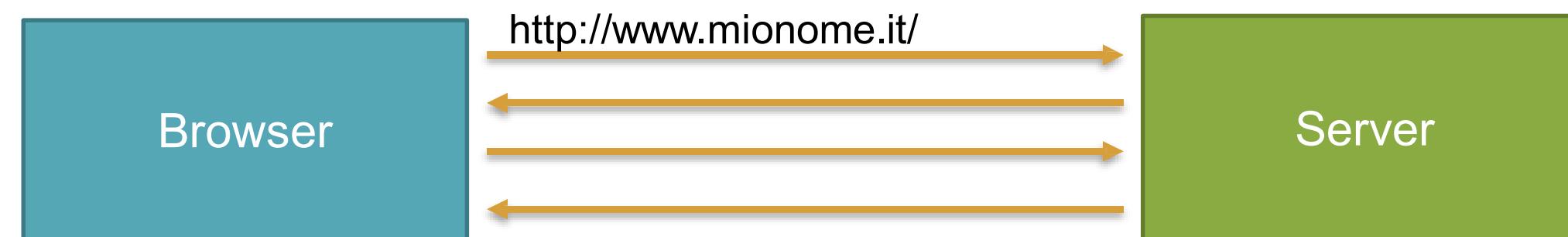
- digitando l'URL nel browser web
- cliccando su un collegamento ipertestuale (link) presente in una pagina già visualizzata
- cliccando su un collegamento ipertestuale presente in altre risorse (ad esempio un'e-mail)

Come funziona un'applicazione web

Dalla richiesta alla risposta

Quando richiediamo l'accesso ad una risorsa, il browser web inizia il processo di richiesta della risorsa al server.

Browser e server si scambiano una serie di messaggi che si concludono con il download della risorsa sul computer dell'utente



Come funziona un'applicazione web

Dalla richiesta alla risposta

Esempio di URL: <http://www.mionome.it/index.php/chi-sono>

Server name: **www.mionome.it**

Risorsa richiesta: **/index.php/chi-sono**

Prima di ogni cosa, il server name viene risolto in un indirizzo IP (ad es. 89.234.123.33) usando il **Domain Name System (DNS)** un database globale e distribuito.

L'indirizzo IP identifica il server web al quale il browser web può inviare le richieste.

Come funziona un'applicazione web

► Esempio - la richiesta di una pagina web

1. il browser effettua una richiesta richiedendo il testo HTML
2. l'HTML ricevuto viene interpretato dal browser
3. il browser effettua tante richieste per quante risorse sono presenti nel codice HTML (immagini, video, fogli di stile,...)
4. ricevuti tutti i file, il browser visualizza la pagina sullo schermo, sulla base delle specifiche HTML, CSS e di altri linguaggi web
5. la pagina visualizzata conterrà tutte le risorse contenute nel codice HTML

Linguaggi client side vs server side

Linguaggi server side

I linguaggi server side consentono di creare applicazioni web che interagiscono in maniera dinamica con il client, interpretando i dati inviati dall'utente o creando pagine elaborando porzioni di codice non HTML inserite nella pagina web.

L'elaborazione del codice porta alla generazione di un flusso di codice HTML o di altro codice (ad esempio JSON o XML) che viene restituito al client.

I linguaggi per lo sviluppo di Web application

Linguaggi server side

- **CGI (Common gateway Interface) e/o Perl:** è stata la prima tecnologia in grado di processare le richieste provenienti dal browser restituendo codice HTML.
- **Java:** offre diversi strumenti e framework per la creazione di web application (Servlet, JSP, Struts, Spring, JSF...)
- **PHP:** è un linguaggio di scripting interpretato largamente diffuso. I principali CMS di uso comune sono scritti in PHP (Wordpress, Joomla, ...)
- **Python:** è un linguaggio di programmazione object oriented che tra le altre cose, consente di scrivere web application
- **.NET (di proprietà Microsoft):** è il framework creato da Microsoft per la realizzazione di web application

I linguaggi per lo sviluppo di Web application

Linguaggi client side

Consentono di eseguire istruzioni e comandi direttamente sul client (nel browser) dell'utente, ad esempio creazione dinamica di form, chiamate asincrone ad altri servizi, etc...

- **JavaScript**: attualmente è il principale linguaggio client side

Cos'è JEE

JEE è l'acronimo di Java Platform Enterprise Edition.

Prima della versione 5, era conosciuta con il nome di Java 2 Enterprise Edition o J2EE.

Cos'è JEE?

È un insieme di specifiche tecniche che consentono di sviluppare servizi robusti, sicuri ed efficienti.

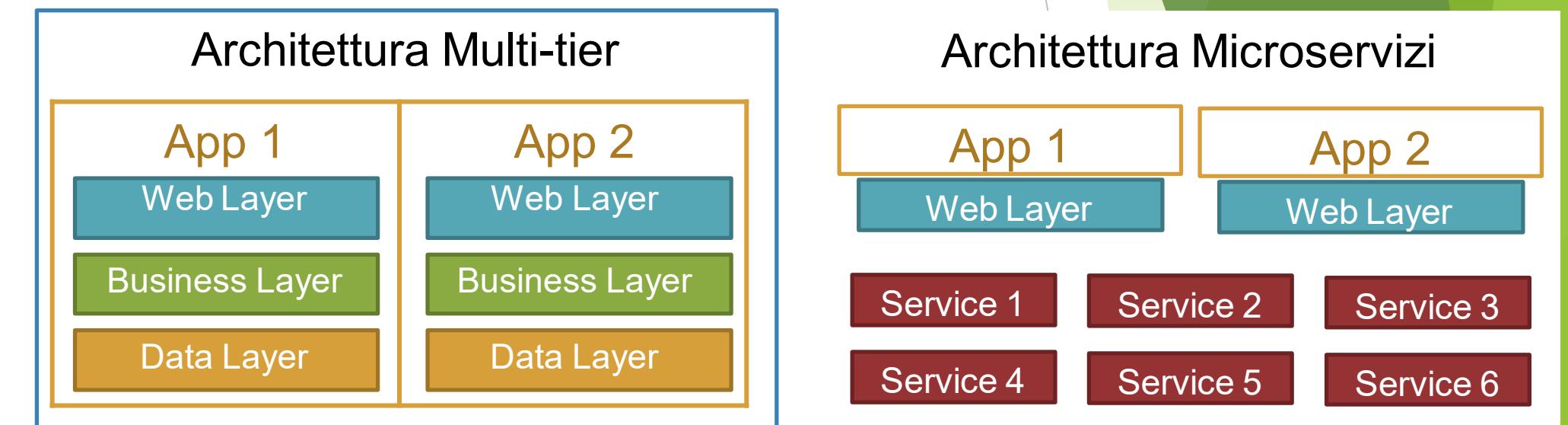
Ad oggi JEE è una delle piattaforme tecnologiche più importanti per lo sviluppo di applicazioni di livello enterprise, specie in contesti dove è imprescindibile avere sicurezza e robustezza del software:

- Banche
- Assicurazioni
- Pubblica amministrazione
- etc...

Cos'è JEE

Fino a qualche anno fa, le applicazioni che seguivano la specifica JEE erano sviluppate con architettura multi-tier, che prevede il disaccoppiamento tra:

- gestione delle interfacce utente (le pagine web)
- gestione della logica di business
- gestione del salvataggio dei dati.



Le nuove specifiche JEE consentono anche lo sviluppo applicazioni basate su architettura a microservizi, che prevede lo sviluppo di tanti piccoli componenti autonomi che interagiscono tra loro.

Questa evoluzione ha portato anche al cambiamento del nome dei software che implementano le specifiche JEE: da Application server a Referencing runtimes.

Cos'è JEE

Ecco le principali specifiche JEE, raggruppate per tipologia:

Specifiche Web Service: queste specifiche consentono di realizzare Web Service REST e SOAP. Le specifiche sono:

- RESTFUL
- JAX-WS
- JSON Processing e JSON Binding
- JAXB

Cos'è JEE

- ▶ Specifiche Web: queste specifiche consentono di realizzare le componenti relative al presentation layer. Le specifiche sono:
 - Servlet e JSP
- ▶ Specifiche Enterprise: queste specifiche consentono di realizzare le componenti utilizzate in contesti applicativi di grandi dimensioni che necessitano di scalabilità, sicurezza, persistenza, etc... Le specifiche sono:
 - EJB, Context e Dependency Injection ecc...
- ▶ Specifiche per l'interazione con i database: queste specifiche consentono di gestire la persistenza dei dati.
 - ▶ Le specifiche sono:
 - JPA, JDBC

Cos'è JEE

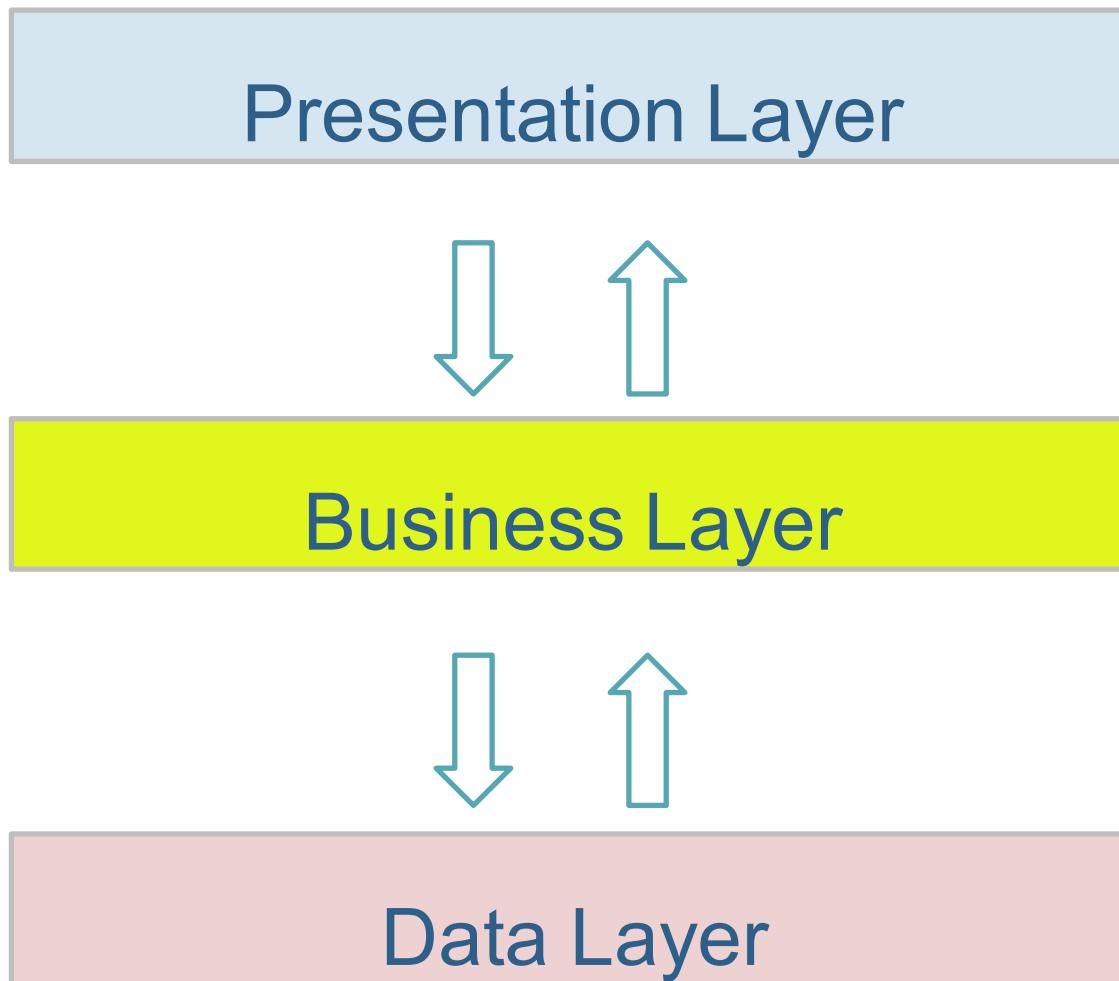
Quale Application server utilizzeremo?

Apache Tomcat 8.5.

Anatomia di un'applicazione web

Un software (compresa la web application) è tipicamente composta da 3 livelli applicativi:

Questa suddivisione in livelli è un **design pattern** (cioè uno schema architetturale applicabile ad un problema ricorrente) che consente di separare nettamente il codice sorgente della parte di presentazione (l'interfaccia utente) dal codice di accesso ai dati e dal codice per la loro gestione.



Presentation Layer

Il presentation layer rappresenta l'interfaccia utente e si occupa di:

- acquisire** dati inseriti dall'utente, generalmente attraverso i form ed altre interazioni dell'utente (clic su link, selezioni di opzioni, etc...)
- visualizzare** i dati all'utente, attraverso tabelle, grafici, form di modifica dati, etc...

I linguaggi di programmazione che la fanno da padrone nel presentation layer sono:

- HTML
- CSS
- JavaScript (ed i vari framework)

In Java il presentation layer si realizza mediante JSP (JavaServer Pages) e JSF (JavaServer Faces) che contengono HTML, CSS, JavaScript e Scriptlet.

Business Layer

- ▶ Il business layer implementa il Domain Model, cioè tutta la parte di software che modella i dati ed implementa i metodi per l'accesso ai dati.
- ▶ Il business layer non deve dipendere dall'interfaccia utente e deve poter essere utilizzato da qualsiasi tipo di applicazione (web, client, web service, etc...)
- ▶ Per fare un esempio... nel business layer di un e-commerce troveremo l'entity OrdineDiVendita (che mappa la struttura dati del database) ed i metodi per gestire il CRUD (Create, Read, Update, Delete).

Data Layer

Il data layer si occupa della gestione dell'accesso ai dati presenti in un database e della loro persistenza.

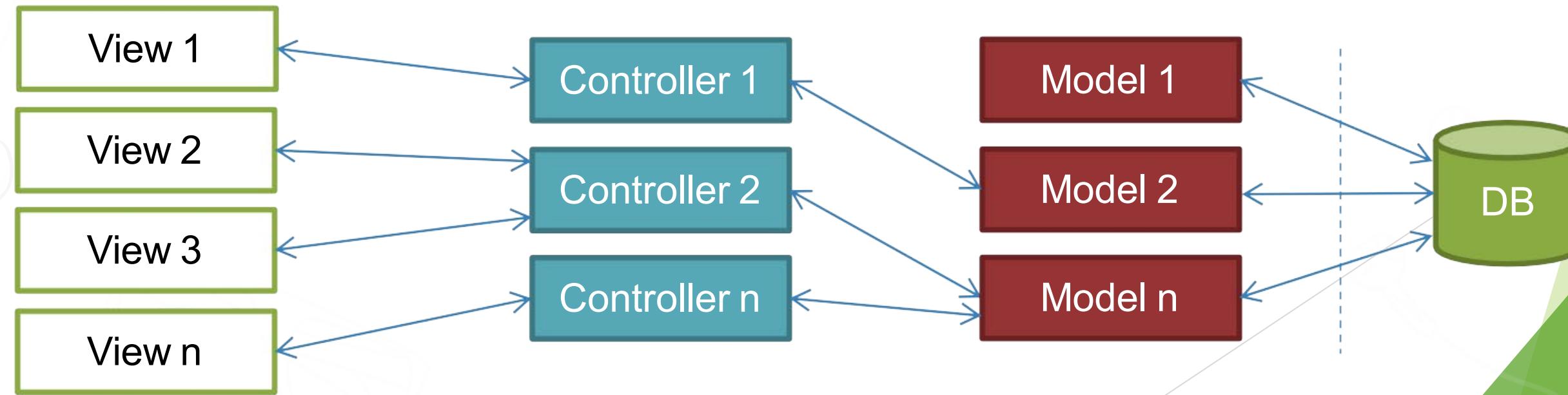
Riceve e gestisce le richieste di accesso (in lettura/scrittura) al DB da parte del Business Layer.

Il data layer contiene classi ed interfacce per implementare il CRUD (Create, Read, Update, Delete). Queste classi sono generalmente utilizzate all'interno del business layer.

Il JDBC è un esempio di data layer, così come Hibernate ed i vari ORM.

Cos'è il pattern MVC

- ▶ Il Model-View-Controller è un pattern architetturale utilizzato nello sviluppo di software.
- ▶ Questo pattern consente di separare nettamente la logica di presentazione dei dati (ovvero le pagine viste dall'utente) dalla logica di business (ovvero le funzionalità per l'accesso a tali dati).
- ▶ **Separare la vista dalla logica di business ci consente di riutilizzare parti di software evitando ridondanza di codice e funzionalità!**



Cos'è il pattern MVC

Lato server

È implementato in numerosi framework ed in vari linguaggi di programmazione:

- Java** (Spring, JSF e Struts)
- PHP** (Symfony, Laravel, CakePHP, ...)
- Python** (Django, TurboGears, Pylons, ...)

Lato client

È implementato da diversi framework JavaScript (AngularJS, JavascriptMVC, ...)

Come funziona l'MVC

Il pattern si basa sulla netta separazione delle attività tra model, view e controller:

- ❑ **model** implementa i metodi per l'accesso ai dati dell'applicazione e si interfaccia con il database o con eventuali livelli di astrazione dei dati (ad es. hibernate, web services, REST services, etc...);
- ❑ **controller** gestisce le richieste di accesso alle view inviate dall'utente, interfacciandosi, se necessario con il model per l'accesso ai dati;
- ❑ **view** gestisce le componenti visualizzate dall'utente (pagine testuali, form, liste, dati recuperati dal model, etc...);

Come funziona l'MVC

Come funziona l'MVC per le Web application?

1. Un utente effettua una richiesta di accesso ad una URL, attraverso un client (browser, app, etc...)
2. Alla URL risponde una web application che elabora la richiesta ricevuta, accede molto probabilmente ad un database e risponde con una pagina HTML

Un'applicazione sviluppata senza il pattern MVC avrà una pagina web che ogni volta che viene chiamata si occuperà di accedere ai dati presenti sul database, elaborarli e generare l'HTML. **La pagina sarà un MIX tra HTML e logica!**

E se ho un'altra pagina che deve accedere a parte dei dati visualizzati in un'altra pagina, devo implementare un'altra volta le funzioni di accesso al database!

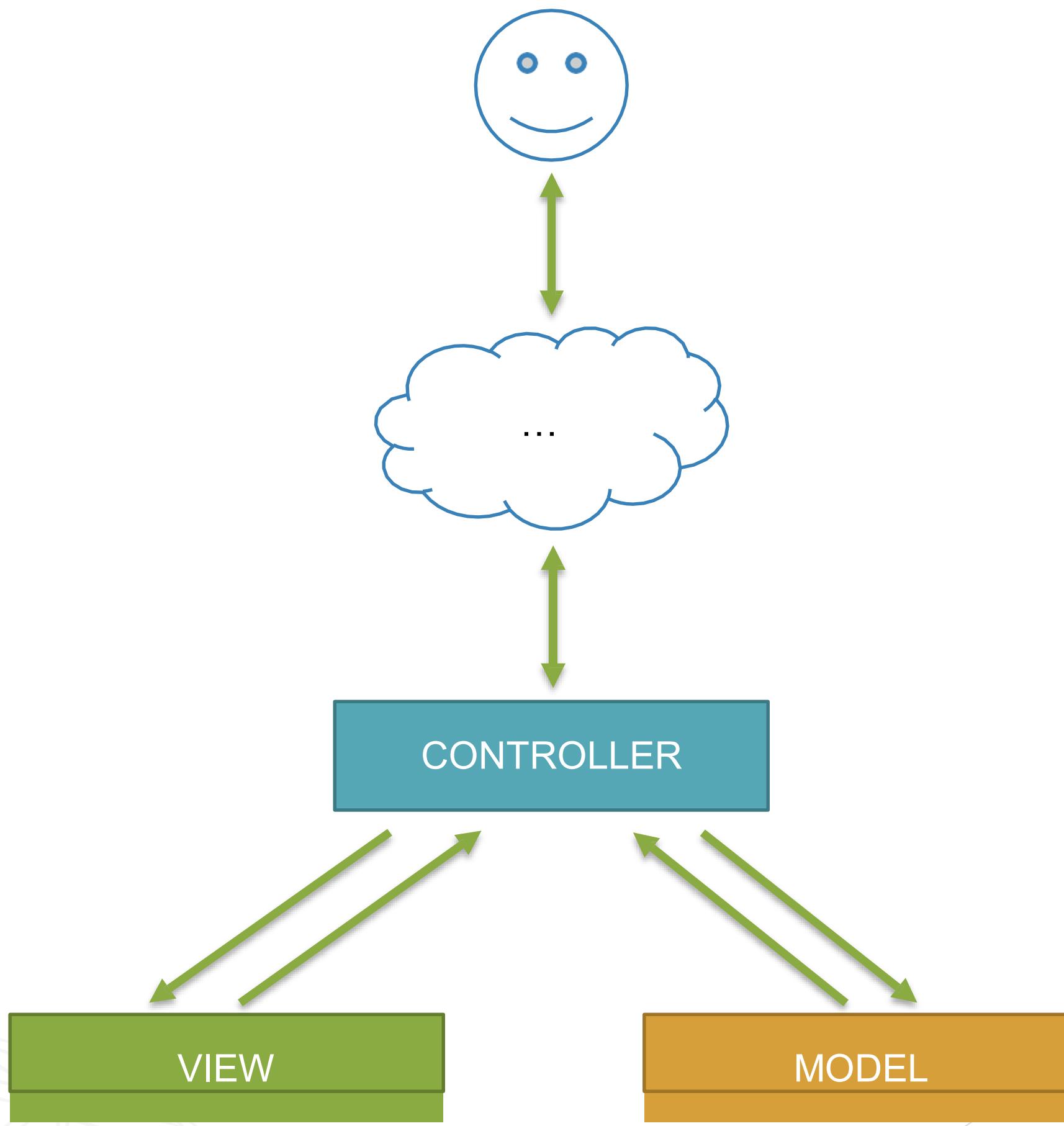
Come funziona l'MVC

Come funziona l'MVC per le Web application?

Un'applicazione sviluppata con il pattern MVC avrà, invece, tre componenti:

1. il model che si occuperà dei dati e fornirà i metodi per accedervi
2. la view (cioè la pagina web) che si occuperà di creare il codice HTML
3. il controller che fa da intermediario tra model e view e che si occuperà di intercettare e gestire la URL richiesta dall'utente

Come funziona l'MVC



Cos'è il file ear

Gli Enterprise Archives sono dei file che consentono di racchiudere in un unico file diversi componenti JEE:

- war
- jar

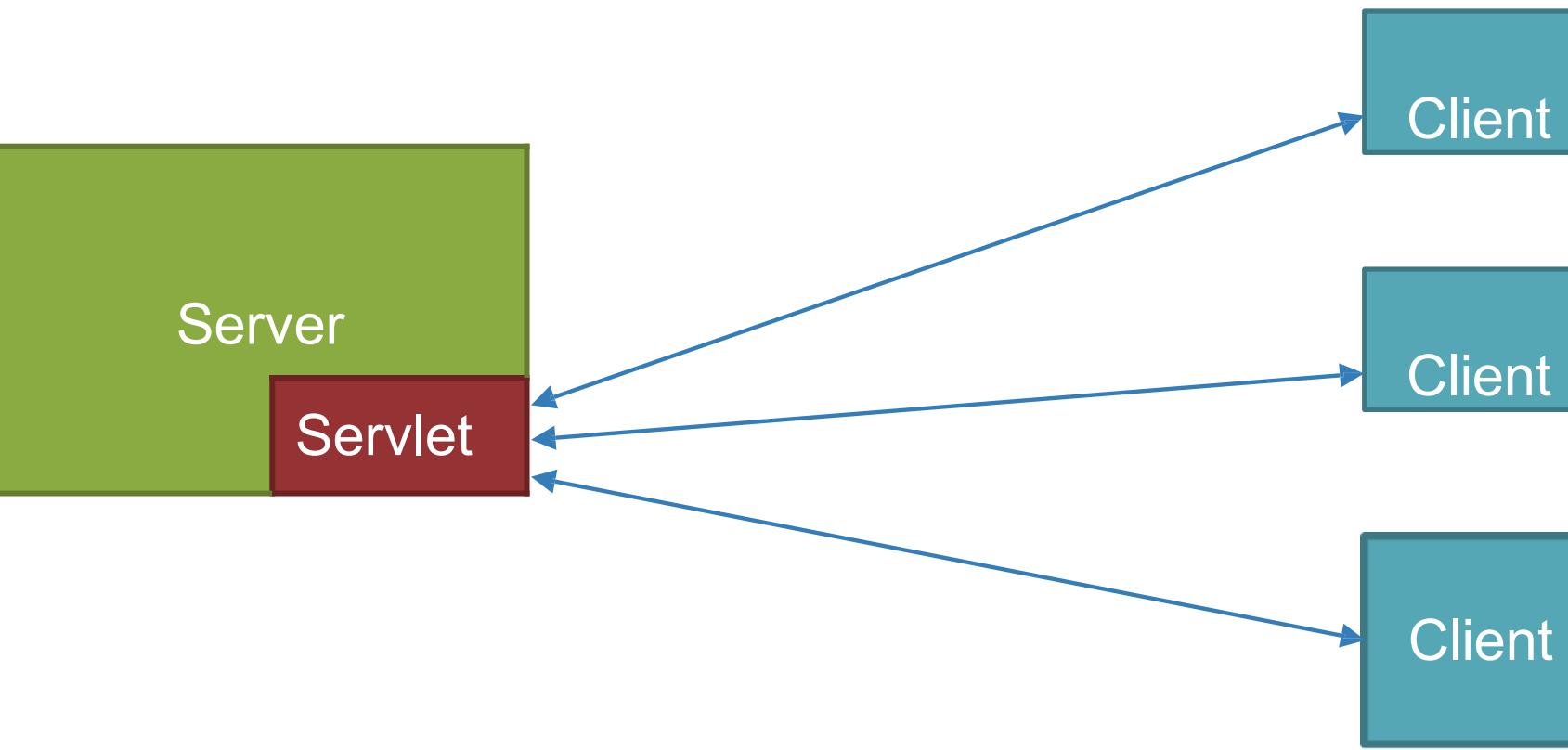
Un ear è sostanzialmente un file compresso che può contenere al suo interno web application e jar.

L'idea che sta dietro alla necessità di avvalersi di questo tipo di file (ear, war, jar) è, fondamentalmente, incentrata sul concetto di facilitare il riutilizzo e il riassemblamento dei componenti in gioco qualora li si voglia impiegare in altre applicazioni J2EE.

Cos'è una Servlet

Una **Servlet** è un software scritto in Java che è in grado di ricevere ed elaborare richieste da uno o più client.

All'interno di un server è possibile far girare più Servlet, ciascuna con compiti diversi



Le Servlet generalmente vengono installate (teoricamente si dice fare il «deploy») all'interno di Servlet Container (ad es. Tomcat).

Il Servlet Container è una piattaforma software in grado di eseguire applicazioni Web sviluppate in Java.

Cos'è una Servlet

Una Servlet non ha delle GUI associate, quindi le librerie SWT, AWT e Swing non vengono utilizzate quando si crea una Servlet.

Le Servlet possono essere utilizzate per:

- creare pagine web dinamiche
- effettuare connessioni ad un db con JDBC e restituire dati in diversi formati
- effettuare l'autenticazione di un utente
- ...

Cos'è una Servlet

In ambito lavorativo, per la creazione di web application generalmente non si sviluppano direttamente Servlet, ma si utilizzano framework che implementano la specifica servlet:

- Spring
- JSF
- Struts
- ...

Le vecchie applicazioni web erano composte principalmente da Servlet e JavaServer Pages (JSP), mischiando logica codice HTML con logica applicativa.

Una servlet può essere utilizzata per sviluppare un singolo servizio (ad es. il download di un file PDF...)

Oggi questo approccio non è più utilizzato e si preferisce separare nettamente la logica di business dalla parte di presentazione.

Cos'è un'HTTPServlet

Il package che contiene classi ed interfacce di riferimento è `javax.servlet`.

Esistono diverse tipologie di Servlet, ognuna delle quali estende la classe principale `javax.servlet.GenericServlet`.

Quando si lavora in ambito web, la classe da utilizzare per la realizzazione di Servlet è `javax.servlet.http.HttpServlet`.

Creare una Servlet vuol dire creare una classe che estende la classe HttpServlet!

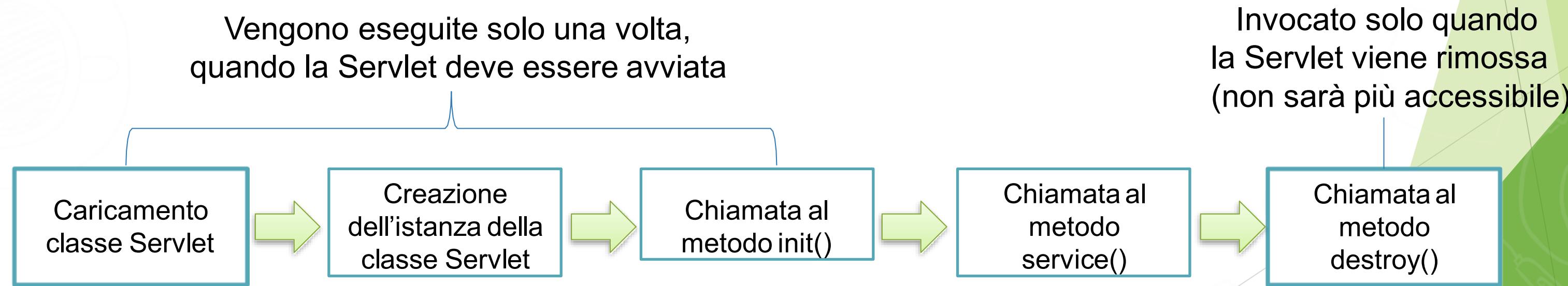
```
@WebServlet("/path-per-invocare-la-servlet")
public class TestServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        ...
    }

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

Ciclo di vita di una Servlet

Una Servlet ha un ciclo di vita ben definito che definisce:

- Come caricarla
- Come istanziarla
- Come inizializzarla
- Come vengono gestite le richieste da parte dei client
- Come rimuoverla



Ciclo di vita di una Servlet

Il ciclo di vita è gestito attraverso i seguenti metodi dell'interfaccia javax.servlet.Servlet:

- ❑ `init(ServletConfig config)`
- ❑ `service(ServletRequest request, ServletResponse response)`
- ❑ `destroy()`

Tutte le Servlet devono implementare questi metodi (direttamente oppure attraverso le classi astratte GenericServlet o HttpServlet).

I metodi init, service e destroy, vengono invocati dal container per gestire il ciclo di vita della Servlet.

Ciclo di vita di una Servlet

FASE 1 e 2

Il Servlet container è responsabile del caricamento e della creazione dell'istanza della Servlet. Il caricamento può avvenire:

- all'avvio del Servlet engine
- all'arrivo della prima richiesta da parte di un client

FASE 3

La fase di inizializzazione si ha solo dopo aver creato l'istanza della Servlet, viene effettuata dal container e consente di inizializzare le risorse di cui ha bisogno la Servlet per gestire le richieste (ad es. aprire le connessioni JDBC, collegarsi ad altri servizi, etc...).

Queste operazioni vengono eseguite una sola volta.

Ciclo di vita di una Servlet

FASE 4

La Servlet gestisce le richieste dei client.

La richiesta è rappresentata da un oggetto di tipo `ServletRequest`

La risposta è rappresentata da un oggetto di tipo `ServletResponse`

Questi due oggetti sono passati in ingresso al metodo `service` dell'interfaccia `Servlet`
`(service(ServletRequest req, ServletResponse res))`.

Nelle richieste che avvengono su protocollo HTTP, gli oggetti sono di tipo `HttpServletRequest` e `HttpServletResponse` (package `javax.servlet.http`).

FASE 5

Il container si occupa di invocare il metodo `destroy()` per liberare le risorse utilizzate dalla Servlet e completare l'elaborazione in corso. A questo punto la Servlet viene rimossa dal container.

La classe astratta HttpServlet, oltre ad implementare i metodi dell'interfaccia Servlet, implementa dei metodi aggiuntivi per gestire le richieste basate su HTTP.

Questi metodi sono i due metodi principali:

- doGet() per gestire richieste HTTP GET
- doPost() per gestire richieste HTTP POST

Tutti i metodi prendono in ingresso due oggetti di tipo HttpServletRequest e HttpServletResponse. I metodi più usati nelle web application sono doGet() e doPost().

HttpServletRequest e HttpServletResponse

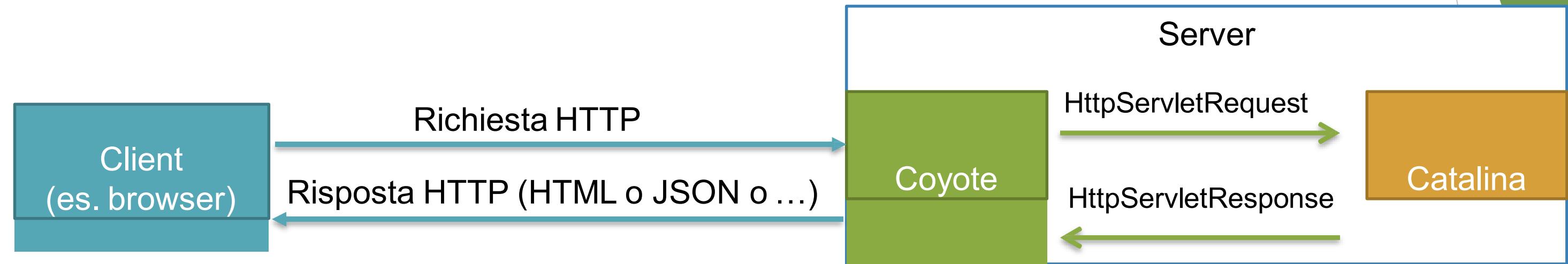
HttpServletRequest ed HttpServletResponse sono **interfacce** contenute nel package `javax.servlet.http`.

Ecco come funzionano

Quando richiediamo l'accesso ad una risorsa web (ad es. quando scriviamo l'URL di una pagina web nella barra degli indirizzi), abbiamo il seguente scenario:

- ❑ Il connettore HTTP (Coyote per Tomcat) riceve la richiesta HTTP che può essere di tipo GET o POST e la reindirizza al servlet container
- ❑ Se la servlet non è ancora in memoria, il servlet container (Catalina per Tomcat) la carica e la inizializza mediante l'invocazione del metodo `init()`
- ❑ Il servlet container incapsula la richiesta HTTP in un oggetto di tipo HttpServletRequest e la passa al metodo `doPost` (se la richiesta è di tipo POST) o `doGet` (se la richiesta è di tipo GET) della servlet
- ❑ La servlet elabora la richiesta e scrive la risposta (ad esempio un codice HTML) nella HttpServletResponse
- ❑ L'HttpServletResponse viene reindirizzata al web server che si occuperà di inviarla al client via HTTP

HttpServletRequest e HttpServletResponse



L'interfaccia `HttpServletRequest` rappresenta la richiesta effettuata dal client alla nostra web application.

L'oggetto di tipo `HttpServletRequest` ricevuto dalla web application (ad esempio da una Servlet) contiene tutti i dati inviati dal client (ad es. le informazioni sul browser, i dati di un form HTML, lo stream di un file, etc...).

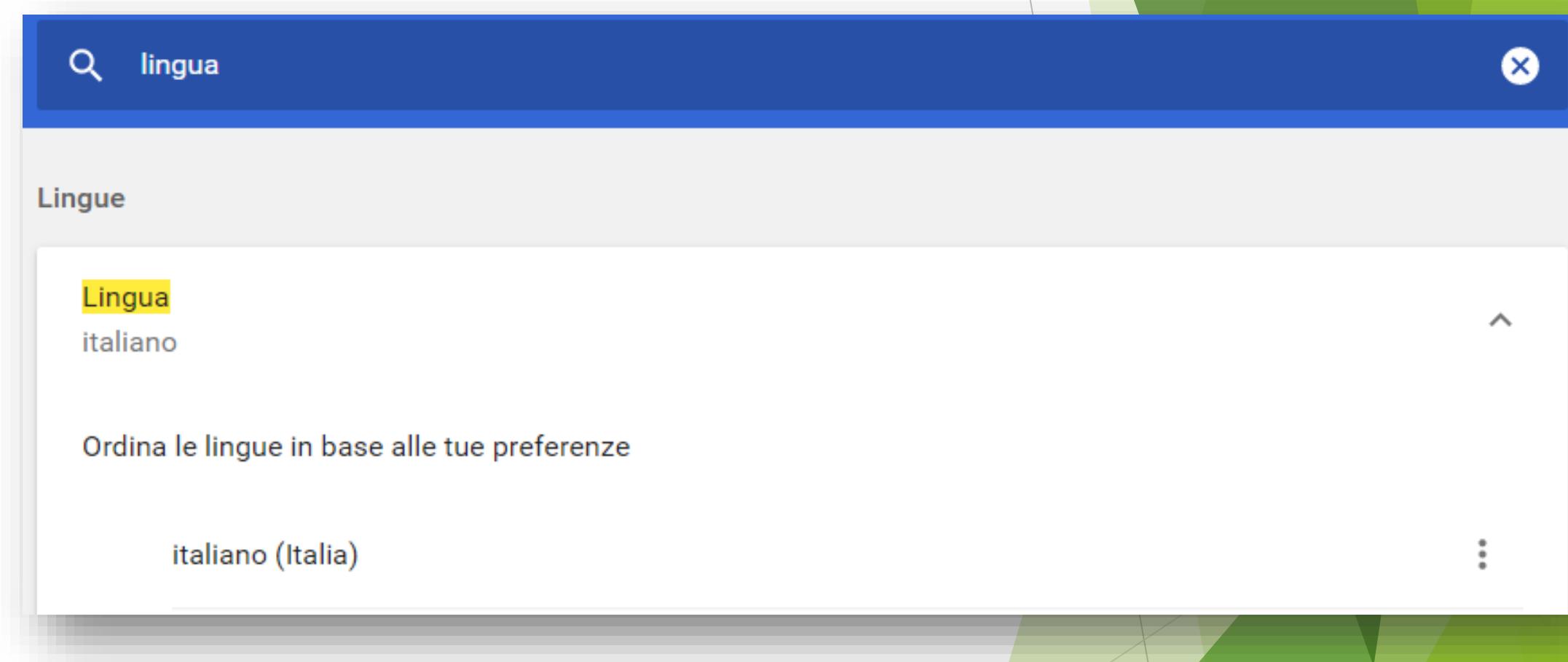
L'interfaccia `HttpServletResponse` conterrà tutte le informazioni della risposta inviata al client (ad es. il codice HTML da visualizzare o un JSON o un XML).

I principali metodi dell'interfaccia HttpServletRequest

`request.getMethod()`: restituisce il nome del metodo HTTP utilizzato per effettuare la richiesta al server (GET, POST, PUT)

`request.getLocale()`: restituisce il Locale (cioè un oggetto che rappresenta una specifica area, ad es. "en" (English), "it" (Italia)) predefinito del client ed è visibile nel campo

"Accept-Language" dell'header. Se il client non inserisce questa informazione nella request, il metodo ritorna il default Locale del server.



Impostazione del Locale in Google Chrome

`request.getCharacterEncoding()`: restituisce il nome del character encoding (la codifica dei caratteri) utilizzato nel body della request. Alcuni nomi di codifiche sono: ISO 8859-1, UTF-8.

I principali metodi dell'interfaccia HttpServletRequest

`request.getContentType()`: restituisci il MIME type del body della request o null se il MIME type è sconosciuto. Il MIME type (Multipurpose Internet Mail Extensions) è un formato nato per la posta elettronica in aggiunta al protocollo SMTP (Simple Mail Transfer Protocol).

Tuttavia oggi è ampiamente utilizzato nel protocollo HTTP per codificare i messaggi scambiati tra client e server web.

Alcuni MIME type che utilizzati nello sviluppo di applicazioni web sono:

- `text/html`: utilizzato per le pagine html
- `multipart/form-data` (attributo `enctype="multipart/form-data"` dell'elemento `form`): utilizzato per le pagine html che contengono form che inviano file in upload
- `application/pdf`: utilizzato per visualizzare file PDF
- `application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`: utilizzato per visualizzare file Excel
- `application/json`: utilizzato per visualizzare contenuti in formato JSON

I principali metodi dell'interfaccia HttpServletRequest

`request.getContextPath()`: restituisce la parte della request URI che corrisponde al path associato alla servlet o in generale alla web application invocata. Il path inizia con il simbolo "/" e non termina mai con il simbolo "/" (ad es. /login può essere il context path associato ad una servlet che effettua il login).

Le servlet che si trovano nel root context (in Tomcat si trovano nella cartella ROOT) il context path è "".

`request.getCookies()`: restituisce un array contenente tutti i cookies inviati con la request. I cookies vengono inseriti all'interno di oggetti di tipo `javax.servlet.http.Cookie`.

`request.getHeader(String name)`: ritorna il valore del parametro specificato in ingresso che si trova nell'header della request (ad es. l'user-agent).

`request.getHeaderNames()`: ritorna un enumeration contenente tutti nomi dei parametri presenti nell'header della request.

I principali metodi dell'interfaccia HttpServletRequest

`request.getQueryString()`: restituisce la query string contenuta nell'URL. La query string è la stringa che si trova dopo il simbolo «?». Il metodo ritorna null se la URL non contiene query string.

Nella query string i parametri sono composti dalla coppia **nome=valore**. In caso di più parametri in query string, si usa il simbolo «&» per separarli.

Esempio:

- URL senza query string: www.mysite.it/about
- URL con query string: www.mysite.it/about?param1=abc¶m2=def (La query string è param1=abc¶m2=def)

`request.getParameter(String name)`: restituisce il valore associato ad un parametro passato nella request (compresi quelli passati nella query string). Se il parametro non esiste il metodo ritorna null.

I principali metodi dell'interfaccia HttpServletRequest

`request.getParameterMap()`: restituisce un oggetto di tipo `java.util.Map` che contiene l'elenco dei parametri (con relative valori) presenti nella request.

`request.getParameterNames()`: restituisce un enumeration contenente tutti i nomi dei parametri passati nella request.

`request.getParameterValues(String name)`: restituisce un array di stringhe associate ad un parametro. Il tipico esempio è un form che contiene degli input che hanno lo stesso nome (generalmente dei checkbox che consentono la selezione di più opzioni).

`request.getAttribute(String name)`: restituisce un oggetto (può essere una stringa ma anche un oggetto complesso) associato al nome passato in ingresso o null se non esiste un attributo con quel nome.

`request.getAttributeNames()`: restituisce un enumeration contenente i nomi degli attributi presenti nella request.

I principali metodi dell'interfaccia HttpServletRequest

- ▶ `request.getRemoteAddr()`: restituisce l'indirizzo IP del client che ha effettuato la request.
- ▶ `request.getRemoteUser()`: restituisce una stringa contenente il parametro utilizzato per il login dall'utente. (generalmente l'username o l'email): Il metodo ritorna null se non esiste un utente autenticato.
- ▶ `request.getUserPrincipal()`: restituisce un oggetto di tipo `java.security.Principal` che contiene il nome dell'utente autenticato o null se non esiste un utente autenticato. L'interfaccia `Principal` definisce un metodo `getName()` per recuperare il nome associato all'oggetto.
- ▶ `request.getRequestDispatcher(String path)`: restituisce un oggetto di tipo `RequestDispatcher` che agisce come wrapper per la risorsa che è associata al path specificato.

I principali metodi dell'interfaccia HttpServletRequest

`request.getRequestURL()`: restituisce un oggetto di tipo StringBuffer contenente l'URL della request e contiene il protocollo, il server name, il numero della porta ed il server path. La stringa ritornata non contiene la query string.

`request.getRequestURI()`: restituisce la porzione di URL che si trova dopo il server name

Esempio:

URL: <http://miosito.it/miaservlet/param?method=abc>

`requestURL()`: <http://miosito.it/miaservlet/param>

`requestURI()`: /miaservlet/param

I principali metodi dell'interfaccia HttpServletRequest

`request.getServerName()`: Restituisce il nome del server su cui è in esecuzione la web application.

`request.getServerPort()`: Restituisce il numero della porta del server su cui è in esecuzione la web application.

`request.getServletPath()`: restituisce la parte di URL a cui è associata la servlet.

Il servlet path è configurato:

- con il `<servlet-mapping>` (per tutte le versioni di servlet)
- con il parametro `urlPatterns={"/miaurl"}` nell'annotation `@WebServlet` (per le servlet 3.x)

`request.getSession()`: restituisce un oggetto di tipo `javax.servlet.http.HttpSession` contenente la sessione associata alla request ricevuta. Se non esiste una session, questo metodo la crea.

`request.setAttribute(String name, Object value)`: memorizza un attributo nella request. Tutti gli attributi vengono cancellati tra una request e l'altra.

I principali metodi dell'interfaccia HttpServletResponse

`response.addCookie(Cookie arg0)`: aggiunge un cookie alla response

`response.addHeader(String name, String value)`: aggiunge un parametro nell'header della response.

`response.encodeURL(String url)`: effettua l'encode dell'URL includendo il session ID. L'implementazione del metodo contiene la logica che determina se il session ID debba essere incluso nell'URL (ad es. se il browser supporta i cookies l'URL encoding non è necessario).

`response.flushBuffer()`: forza il l'invio del contenuto presente nel buffer al client.

`response.getOutputStream()`: restituisce un oggetto di tipo javax.servlet.ServletOutputStream utilizzabile per scrivere dati binary nella response (ad esempio per inviare un PDF al client...)

L'invocazione del metodo `flush()` sull'oggetto effettua l'invio della response al client.

I principali metodi dell'interfaccia HttpServletResponse

`response.getWriter()`: restituisce un oggetto di tipo `java.io.PrintWriter` che può essere utilizzato per inviare testo (ad esempio codice HTML) al client.

`response.sendError(int statusCode, String message)`: invia un error con lo status code indicato al client.

`response.sendRedirect(String URL)`: è utilizzata per reindirizzare la risposta ad un'altra risorsa (ad es. una servlet o una jsp). Il parametro da passare in input è una URL relativa o assoluta. Il metodo lavora a livello client perché utilizza la barra degli indirizzi del browser per effettuare una nuova request (quando si riceve la risposta, nella barra degli indirizzi comparirà la URL dove si verrà reindirizzati).

`response.setStatus(int sc)`: imposta lo status code della response

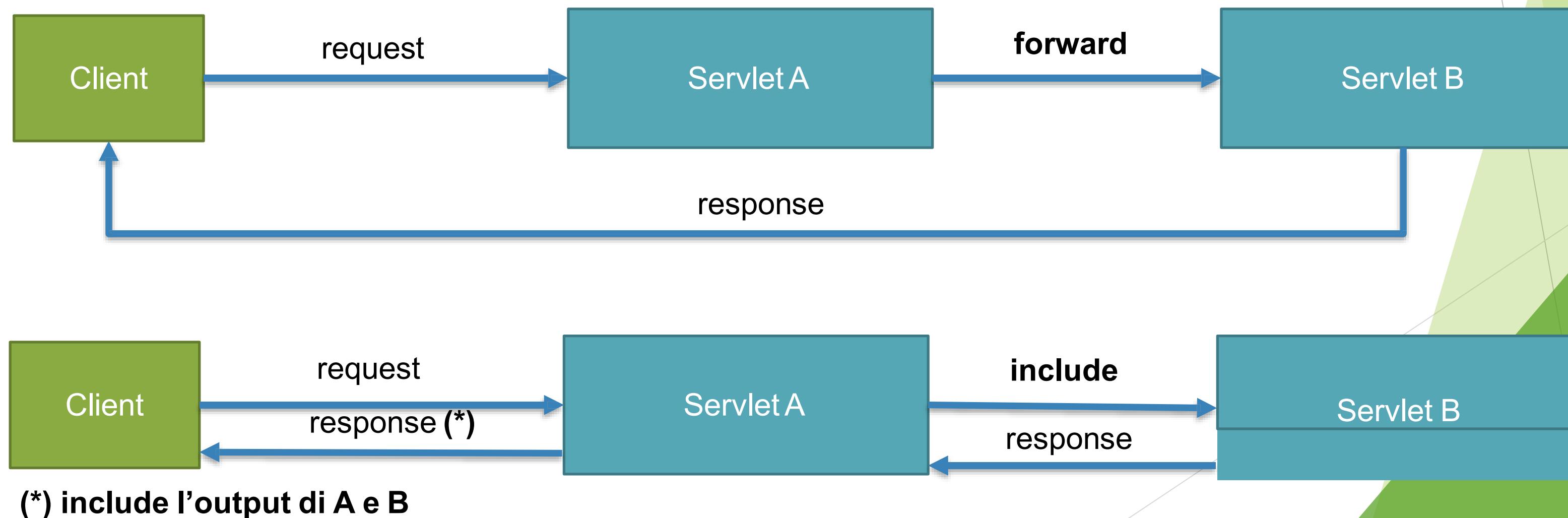
Request Dispatching

- ▶ Quando sviluppiamo una web application, difficilmente riusciamo a gestire tutte le funzionalità con una servlet.
- ▶ Nel paradigma MVC, inoltre, le richieste arrivano ad uno o più controller che elaborano i dati e reindirizzano la request alla risorsa interessata (una JSP o un'altra Servlet).
- ▶ Per gestire questa attività di «dispatching» possiamo utilizzare l'interfaccia `javax.servlet.RequestDispatcher`. Per inoltrare la request possiamo fare in due modi:
 - ▶ □ `request.getServletContext().getRequestDispatcher(String jspPath)` per reindirizzare la request ad una jsp;
 - ▶ □ `request.getServletContext().getNamedDispatcher(String servletName)` per invocare un'altra servlet.
- ▶ Entrambi i metodi ritornano un oggetto di tipo `RequestDispatcher`.

Request Dispatching

L'interfaccia RequestDispatcher mette a disposizione due metodi per inoltrare la richiesta alla nuova risorsa:

- `forward(request, response)` delega l'invio della risposta alla risorsa a cui ha inoltrato la request;
- `include(request, response)` per includere l'output ottenuto da un'altra risorsa web (ad es. un'altra servlet).



Cosa sono le Java Server Pages

Le Java Server Pages (JSP) sono una tecnologia importantissima nello sviluppo di applicazioni web. La principale caratteristica di una pagina JSP è che è costituita da un mix tra codice HTML e codice Java.

In sostanza, attraverso le JSP possiamo creare pagine che generano dinamicamente codice HTML!

Un file JSP deve avere estensione.jsp.

Il codice Java è racchiuso tra i simboli <%> e <%>

Pagina HTML

```
<html>
  <body>
    <h1>Titolo</h1>
    <p>Testo della pagina...</p>
  </body>
</html>
```

Pagina JSP

```
<html>
  <body>
    <h1><% out.println(request.getParameter("titolo")); %></h1>
    <p><% out.println(request.getParameter("testo")); %></p>
  </body>
</html>
```

Ciclo di vita di una JSP

Il ciclo di vita di una JSP è il seguente:

1. La prima volta che viene richiesto un file JSP, Jasper procede alla compilazione del file generando una servlet che potrà essere gestita da Catalina. La servlet viene caricata in memoria in modo che dalla successiva richiesta sia già disponibile senza necessità di una nuova compilazione
2. L'elaborazione della servlet porta alla generazione di un output HTML che viene inviato al browser
3. Ad nuova richiesta della stessa JSP, il server verifica se il file .jsp è stato modificato: se non ci sono state modifiche viene utilizzata la servlet presente in memoria, altrimenti si procede come per il punto 1

Scriptlet

- ▶ Come già anticipato, la caratteristica principale di una JSP è che può contenere un mix tra codice HTML e codice Java.
- ▶ Il codice Java, per essere interpretato come tale, deve essere inserito all'interno dei simboli `<% %>`. Questo blocco di codice Java viene detto scriptlet.

```
<%
if(request.getAttribute("myVar") != null) {
    out.println("my Var non è null!");
} else {
    out.println("my Var è null!");
}
%>
```

Dichiarazioni

Per dichiarare una variabile o un metodo all'interno di una JSP è necessario utilizzare la seguente sintassi:

<% ! dichiarazione %>

```
<% ! String myVar = request.getParameter("myVar") %>

<% ! public String isPalindroma(String text){
    ...
    return ...;
}
%>
```

Espressioni

Per inserire un'espressione all'interno di una JSP, la sintassi è:

< % = espressione % >

```
<% ! String myVar = request.getParameter("myVar") %>

<% ! public String isPalindroma(String text){
    ...
    return ...;
}
%>
...

<p>La parola è palindroma? <b><%=isPalindroma(myVar) %></b></p>
```

Cosa sono le direttive

Le direttive consentono di specificare alcune caratteristiche necessarie per il corretto funzionamento della pagina e sono:

<%@ page: consente di definire alcune impostazioni relative alla compilazione della pagina attraverso i suoi attributi. I principali attributi sono:

- **Language**, specifica il linguaggio da utilizzare in fase di compilazione Esempio: **<%@ page language="Java"%>**
- **Import** specifica i package da includere nella pagina per il corretto funzionamento Esempio: **<%@ page import="java.util.List" %>**
- **isThreadSafe** se impostato a true indica che la pagina è in grado di servire più richieste contemporaneamente
Esempio: **<%@ page isThreadSafe="true" %>**

Cosa sono le direttive

<%@ include: consente di includere altri file (jsp , html o file di testo) alla pagina. Questa direttiva consente di creare porzioni di codice HTML e di includerle all'interno di più pagine.

Esempio: `<%@ include file="fileB.jsp" %>`

<%@ taglib: consente di utilizzare all'interno di una JSP dei custom tag.

Esempio: `<%@tagliburi="http://java.sun.com/jsp/jstl/core"prefix="c"%>`

Cosa sono le tag library

Le tag library sono dei componenti, da utilizzare all'interno dello strato di presentation (nelle jsp), che consentono di implementare numerose funzionalità (ad es. formattazione di testi e date, iterazione di elementi, et...).

Un custom tag è un componente XML (ad es. <c:foreach> ... </c:foreach>)

Cosa sono le tag library

I principali vantaggi nell'usare i custom tag sono:

- Riutilizzo all'interno di più pagine web ed anche di più web application
- Utilizzo dei tag XML based al posto di puro codice Java nelle scriptlet
- Gestione di logiche applicative riutilizzabili
- Eleganza nella produzione del codice
- Semplicità d'uso anche da parte di non sviluppatori

Come funzionano le tag library?

JSP

```
<html>  
...  
<mytag:nomeTag />  
</html>
```

```
public class jrun__demoejspa {  
    private ServletConfig config;  
    ...  
    /* il tag viene convertito in un oggetto */  
}
```

```
<html>  
...  
codice HTML generato dal TAG  
</html>
```

Cosa sono le JSTL - JSP Standard Tag Library

Sono un insieme di JSP tag, che implementano le principali funzionalità che abbiamo bisogno di utilizzare quando realizziamo un'applicazione web, ad esempio:

- Manipolazione di oggetti e stringhe
- Iterazione di liste
- Visualizzazione di parti di html in base a determinate condizioni
- Internazionalizzazione dei contenuti
- ...

Le JSTL sono raggruppate in 5 categorie:

core, formattazione, SQL, XML, funzione

Configurazione della web app per l'utilizzo di JSTL

Prima di tutto è necessario scaricare il jar contenente tutto il pacchetto JSTL.

Per scaricare il jar contenente le JSTL possiamo:

- Andare sul sito: <https://tomcat.apache.org/taglibs/standard/>

Il file jar andrà messo dentro WEB-INF/lib.

Configurazione della web app per l'utilizzo di JSTL

Configurazione delle JSP

Nelle pagine JSP, se vogliamo utilizzare i tag JSTL, dobbiamo utilizzare la direttiva `<%@taglib` in cui si desidera utilizzare una tag library, per esempio la *core*, si deve usare la direttiva taglib indicando:

- l'**uri** (che servirà al container per recuperare dalla mappa tutti i tag handler)
- il **prefix** che sarà utilizzato nella jsp per scrivere i tag

Ad esempio, se vogliamo utilizzare nella jsp i tag core, dobbiamo scrivere:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

e poi potremo utilizzare i tag nel seguente modo:

```
<c:if test="">...</c:if>
```

I tag core

I tag core consentono di effettuare operazioni di iterazione, verifiche condizionali, impostazione e rimozione di variabili, etc...

Per utilizzare i tag core, nelle JSP dobbiamo inserire la seguente direttiva:

```
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Vediamo i principali tag...

I tag core

<c:out>: utilizzato per visualizzare i risultati di un'espressione (è l'analogo di <% = ... %>)

<c:out value="stringa o espressione" default="valore se value è null" escapeXml="" />

(escapeXml se TRUE converte i caratteri speciali (ad es. < > & ') nella relativa codifica (ad es < diventa <))

<c: set>: utilizzato per impostare delle variabili con relativo valore.

<c:set var="nome della variabile" value="valore della variabile" scope="request, session..." />

I tag core

<c:if>: è l'analogo dello statement if (senza else...).

Se l'espressione restituisce true il tag esegue il codice contenuto nel suo body.

```
<c:if test="espressione"  
      var="nome della variabile dov'è salvato il risultato del test">  
    ...  
</c:if>
```

I tag core

< c:choose>, < c:when>, < c:otherwise>: sono l'analogo dello statement switch/case.

```
<c:choose>
  <c:when test="espressione da verificare"/>
  ...
  </c:when>
  <c:when test=" espressione da verificare "/>
  ...
  </c:when>
  ...
<c:otherwise>
  ...
</c:otherwise>
</c:choose>
```

I tag core

<c:import>: è l'analogo di <jsp: include> solo che consente anche di inserire URL assolute (<http://www.miosito.it>).

```
<c:import  
    url="url path"  
    var="variabile dove verrà salvata la url indicata"  
    scope="scope della variabile, request session...«  
    context="opzionale...simbolo / seguito dal nome di una web application locale" />
```

I tag core

`<c:forEach>`: itera liste di oggetti.

```
<c:forEach  
    items="lista di oggetti"  
    begin=""  
    end=""  
    step="avanzamento della lista, default è 1"  
    var="nome della variabile dove viene salvato l'i-esimo oggetto della lista"  
    varStatus="variabile contenente la posizione in cui si trova l'iterazione">  
    ...  
</c:forEach>
```

I tag core

- ▶ <c:forTokens>: consente di iterare una stringa, specificando il separatore (convertendo, quindi, la stringa in array di stringhe...)
 - ▶ <c:forTokens>
 - ▶ *items="stringa da iterare"*
 - ▶ *delims="separatore"*
 - ▶ *begin=<int>*
 - ▶ *end=<int>*
 - ▶ *step="avanzamento della lista, default è 1"*
 - ▶ *var="nome della variabile dove viene salvato l'i-esimo oggetto della lista"* *varStatus="variabile contenente la posizione in cui si trova l'iterazione">*
 - ▶ ...
 - ▶ </forTokens>

Come installare Tomcat

- 1) Collegarsi a <https://tomcat.apache.org/download-80.cgi>
- 2) Dopo averlo scaricato, va estratto all'interno di una cartella desiderata.
- 3) Per eseguire app JEE è necessario il JDK, non soltanto il JRE, perché non abbiamo bisogno soltanto dell'ambiente a runtime, ma anche del tool di compilazione.

Come è fatto Tomcat

- **bin**: ci sono diversi file per l'esecuzione del Referencing runtime (startup, shutdown). ESEMPIO di startup.
Per farlo funzionare è importante settare JAVA_HOME nelle variabili d'ambiente.
- **conf**: contiene i file di configurazione, come il server.xml che indica le porte in ascolto del server (8080).
- **lib**: contiene una serie di librerie che implementano i vari livelli di JEE.
- **logs**: contiene i file di logs di Catalina, host_manager ecc..
- **temp**: file temporanei rimossi quando il server viene spento.
- **webapps**: contiene tutte le nostre webapp deployate su TomEE.
- **work**: contiene tutti i file JSP compilati.

Come configurare Tomcat

- 1) Aprire Eclipse
- 2) Window->Preferences->Server->Runtime Environmnents->Add
- 3) Scegliere la versione di Tomcat corrispondente
- 4) Indicare la cartella di installazione di Tomcat.
- 5) Aggiungere la view "Servers" in basso attraverso Windows->Show view.
- 6) Visualizzare in dettaglio le informazioni del server appena creato facendo doppio click su di esso.

Come creare una prima Web App

- 1) Aprire Eclipse
- 2) File->New->Dynamic Web Project
- 3) Creare un Working Set
- 4) Cambiare destinazione file compilati in WebContent\WEB-INF\classes
- 5) Context root: è il nome a cui risponde la nostra web app (nel nostro server ci sarà più di una web app, ognuna con context-root diverso)
- 6) Content directory: contiene tutti i file con le info della nostra web app (jsp, file compilati ecc...)
- 7) Selezionare deployment descriptor (si troverà all'interno di WEB-INF)

Cos'è il deployment descriptor?

Si trovano diverse informazioni utili nell'esecuzione della nostra web app:

- welcome-page**: è il tag che ci permette di specificare quali pagine utilizzare come "benvenuto"
- error-page**: ci permette di reindirizzare l'utente in una pagina particolare quando si verifica un determinato errore (importante anche il tag error-code).
- filter**: sono utilizzati per configurare i filtri Java
- servlet**: ci permette di configurare una servlet specificando il path a cui deve rispondere ed il nome della servlet stessa

I limiti del protocollo HTTP

Come abbiamo già detto, il protocollo HTTP, è stateless.

Questo vuol dire che il server, dopo aver soddisfatto la richiesta ricevuta, chiude la connessione con il client, senza conservare informazioni sul client stesso.

Se lo stesso client effettua un'altra richiesta allo stesso web server, viene aperta una nuova connessione, viene elaborata la richiesta, viene inviata la risposta al client e viene chiusa la connessione.



Cos'è una sessione HTTP

Per sopperire al limite del protocollo HTTP, i web server devono implementare un meccanismo per la gestione delle richieste effettuate dagli utenti.

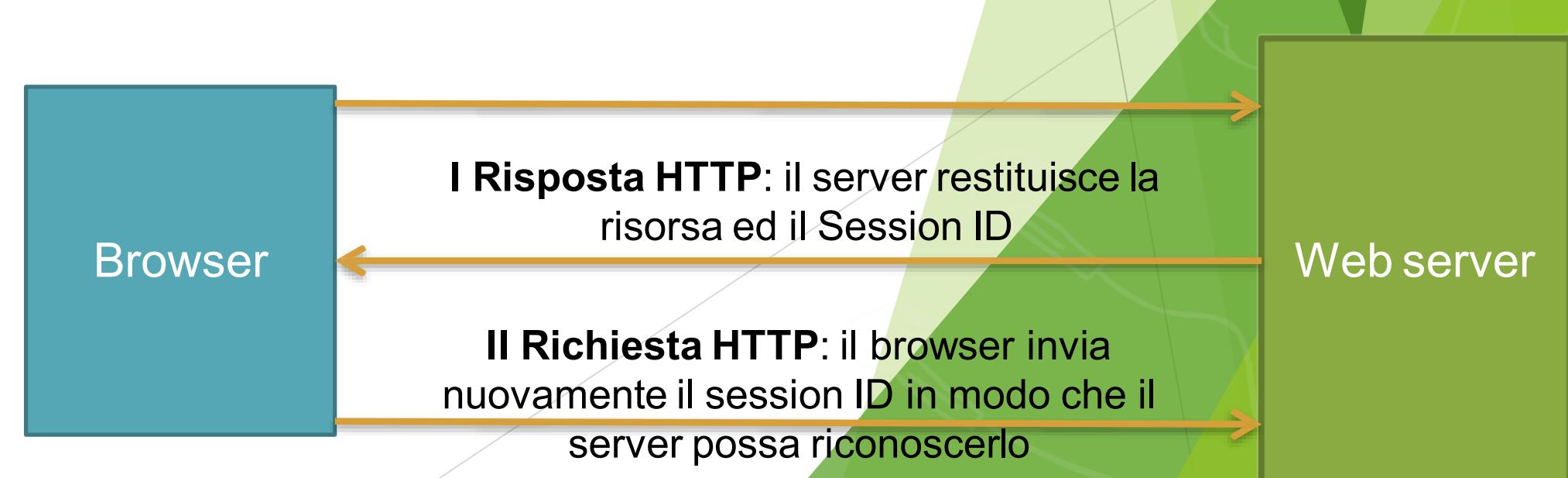
Questo meccanismo si basa sulla gestione della sessione HTTP, un processo che consente di tenere traccia delle richieste effettuate dai vari client.

Ecco alcuni esempi in cui è fondamentale l'utilizzo della sessione:

- Accesso ad un'area riservata della nostra web application
- Il carrello di un sito di e-commerce
- Accesso ad un social network

La gestione della sessione HTTP in Java

- ▶ La sessione HTTP in Java viene gestita attraverso due componenti:
 - ▶ Il parametro JSESSIONID salvato all'interno di un cookie che rappresenta l'ID univoco di una sessione
 - ▶ L'interfaccia **javax.servlet.http.HttpSession**
- ▶ Quando una web application Java riceve una richiesta da un client, il Servlet Engine (Catalina) controlla se la request ricevuta contiene il JSESSIONID:
 - Se è presente, vuol dire che esiste già una sessione per il client



La gestione della sessione HTTP in Java

È importante sottolineare che la gestione della sessione, non intacca la logica stateless del protocollo HTTP.

Al termine di ogni richiesta, infatti, il server chiude sempre la connessione con il client.

HttpSession

Per recuperare l'oggetto HttpSession, l'interfaccia HttpServletRequest mette a disposizione il metodo `getSession()`.

HttpSession session = request.getSession();

Il metodo `getSession()` restituisce l'oggetto di tipo HttpSession se esiste una sessione attiva, altrimenti ne crea una nuova.

Cosa sono i filtri

I filtri sono **oggetti di tipo javax.servlet.Filter** e rappresentano un potente meccanismo in grado di effettuare operazioni di **pre-processing delle richieste e post-processing delle risposte**.

I filtri intervengono, pertanto, prima che una richiesta raggiunga la servlet o appena dopo che la risposta esca dalla servlet.

I filtri sono disponibili a partire dalla versione 2.3 delle specifiche Servlet.

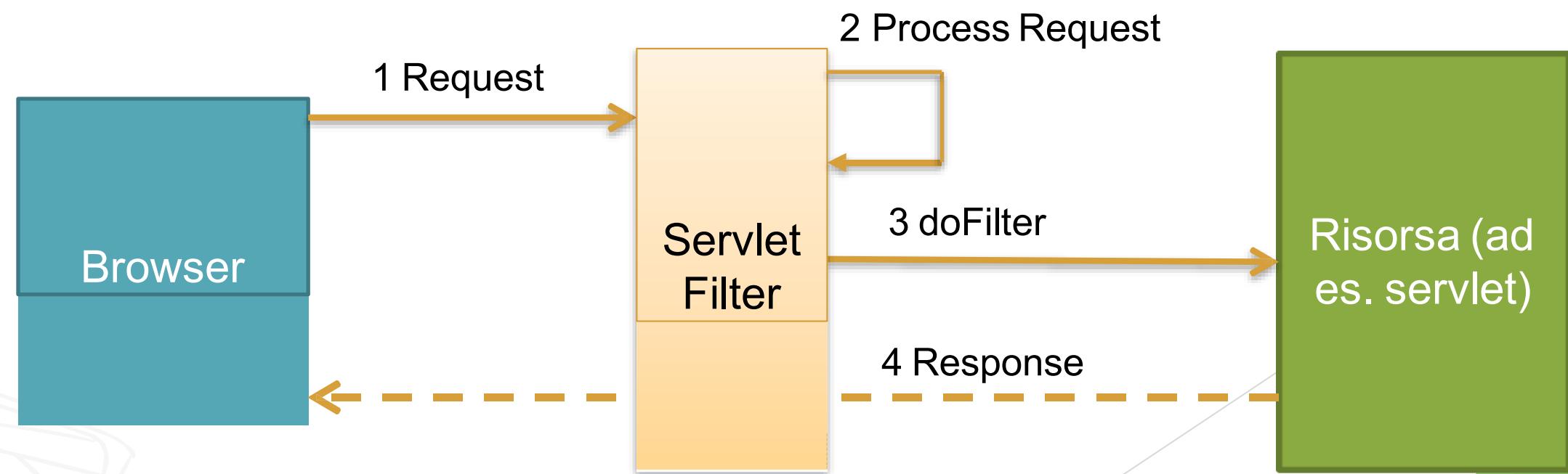
Esempi di filtri comunemente implementati sono:

- Authentication Filters
- Logging Filters
- Data compression Filters
- Encryption Filters
- ...

Pre-processing

Come funziona...

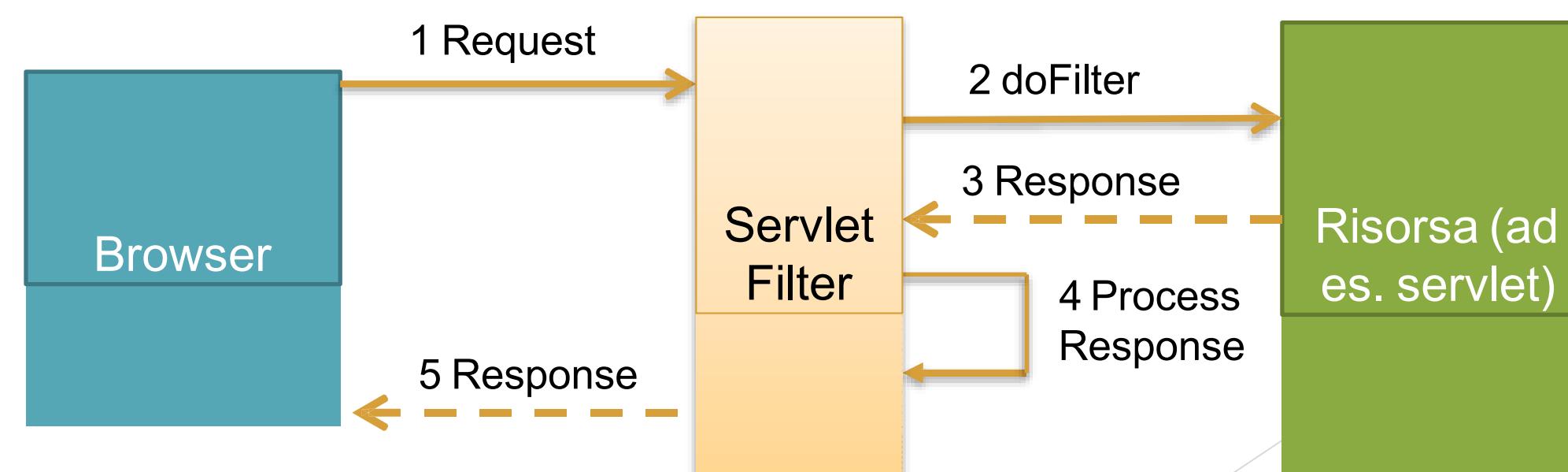
1. Il client invia una richiesta al server
2. Il filtro intercetta la richiesta
3. Il filtro pre-processa la richiesta (in questa fase può effettuare, ad es., accesso al database, accedere i parametri della request, etc...)
4. Il filtro richiama il metodo doFilter() per inoltrare la request alla servlet o in generale alla risorsa richiesta
5. La risorsa invocata elabora la richiesta e risponde al client.



Post-processing

► Come funziona...

1. Il client invia una richiesta al server
2. Il filtro intercetta la richiesta
3. Il filtro richiama il metodo doFilter() per inoltrare la request alla servlet o in generale alla risorsa richiesta
4. La risorsa invocata elabora la richiesta e risponde al client
5. Il filtro processa la response
6. La risposta intercettata, viene inviata al client



Come si implementa un Filtro

Per implementare un filtro in una web application è necessario:

1. Scrivere una classe che implementi l'interfaccia javax.servlet.Filter
2. Definire il filtro nel web.xml ed impostare l'ordine di esecuzione

```
<filter-mapping>
  <filter-name>Nome Filtro</filter-name>
  <url-pattern>/pathIntercettato</url-pattern>
</filter-mapping>
```

filter-name è il nome dato al filtro.

url-pattern specifica quale path (definito nell'elemento url-pattern o servlet-name) deve essere intercettato dal filtro che ha nome filter-name

Come si implementa un Filtro

I metodi definiti nell'interfaccia javax.servlet.Filter:

- ❑ **init(FilterConfig filterConfig)**: invocato dal server, dopo la creazione dell'istanza del filtro e appena prima dell'attivazione dello stesso.
- ❑ **doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**: invocato dal container, contiene tutte le operazioni che dovremo effettuare quando il filtro intercetta una URL.
- ❑ **destroy()**: invocato dal container per terminare il ciclo di vita di un filtro.

Filter chain

Java consente di eseguire più filtri associati in sequenza: è il concetto di filter chain!

Esempio:

```
<filter-mapping>
    <filter-name>Filtro1</filter name>
    <url-pattern>path</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>Filtro2</filter name>
    <url-pattern>path</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>FiltroN</filter name>
    <url-pattern>path</url-pattern>
</filter-mapping>
```

Per creare una filter chain, dobbiamo:

- creare i filtri
- associare ciascun filtro ad un path o una servlet

I filtri verranno eseguiti nell'ordine dei filter-mapping
(nell'esempio, prima verrà eseguito il Filtro1, poi il Filtro2,
etc...)



The background features a large, abstract graphic composed of numerous overlapping triangles in various shades of green. The triangles are oriented at different angles, creating a dynamic and layered effect. The colors range from bright lime green on the left to darker forest green on the right.

MAVEN

Che cos'è Maven?

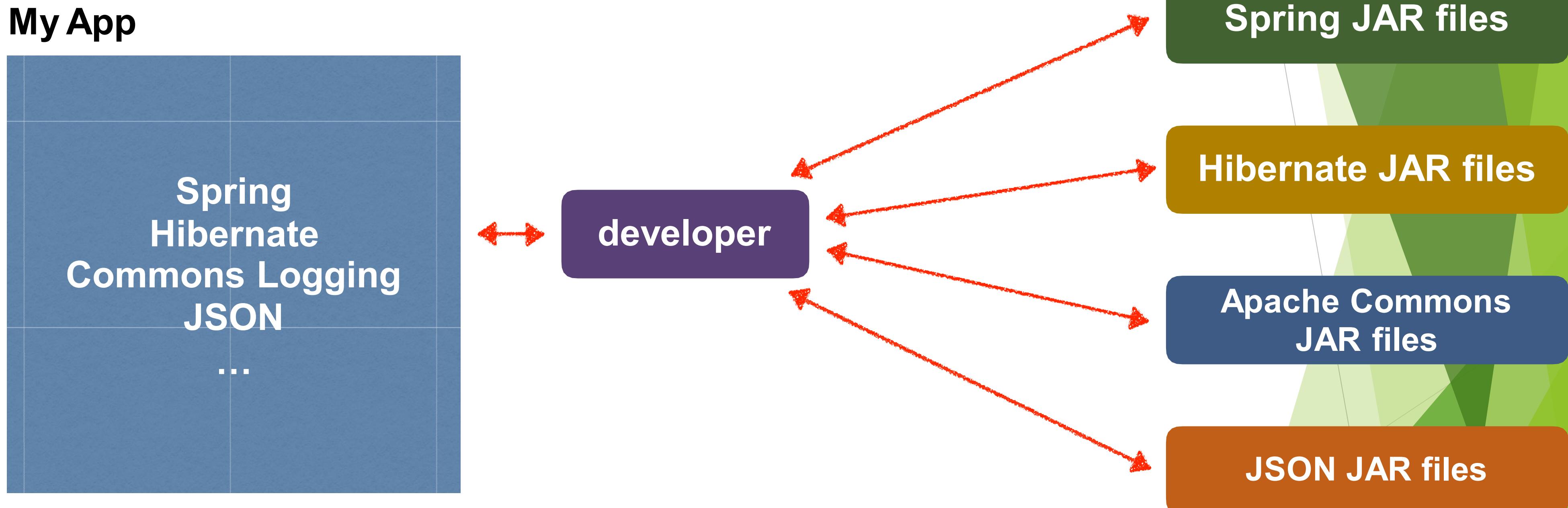
Maven è uno strumento di project management

L'uso più diffuso di Maven è per la gestione della compilazione e delle dipendenze

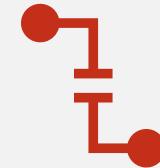
Quali problemi risolve Maven?

- ▶ Quando si compila il progetto Java, potrebbero essere necessari file JAR aggiuntivi
- ▶ Ad esempio: Spring, Hibernate, Commons Logging, JSON etc...
- ▶ Un approccio consiste nello scaricare i file JAR da ogni sito Web dei singoli progetti
- ▶ Aggiungere manualmente i file JAR al build path

Il mio progetto senza Maven



Soluzione Maven



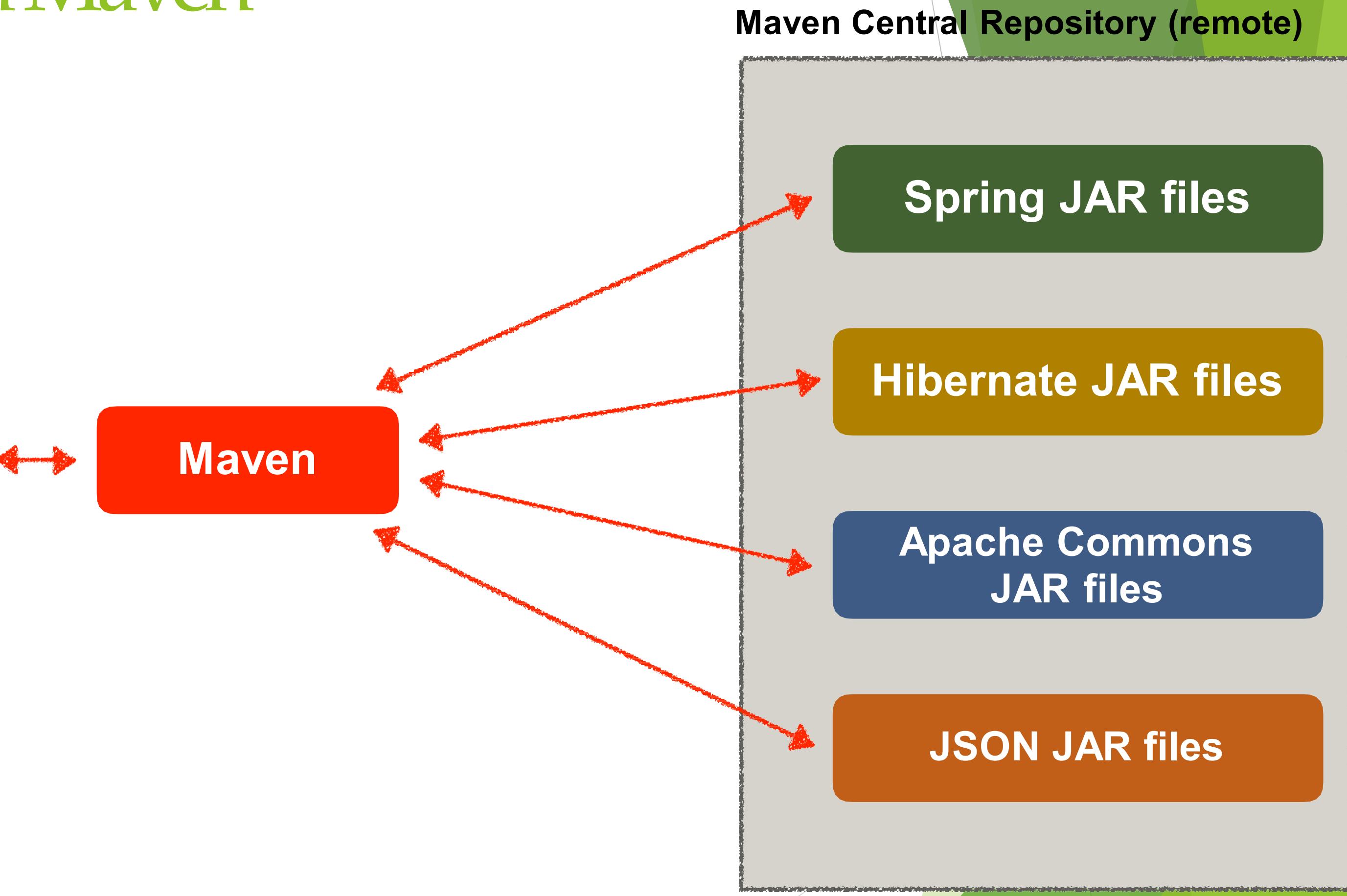
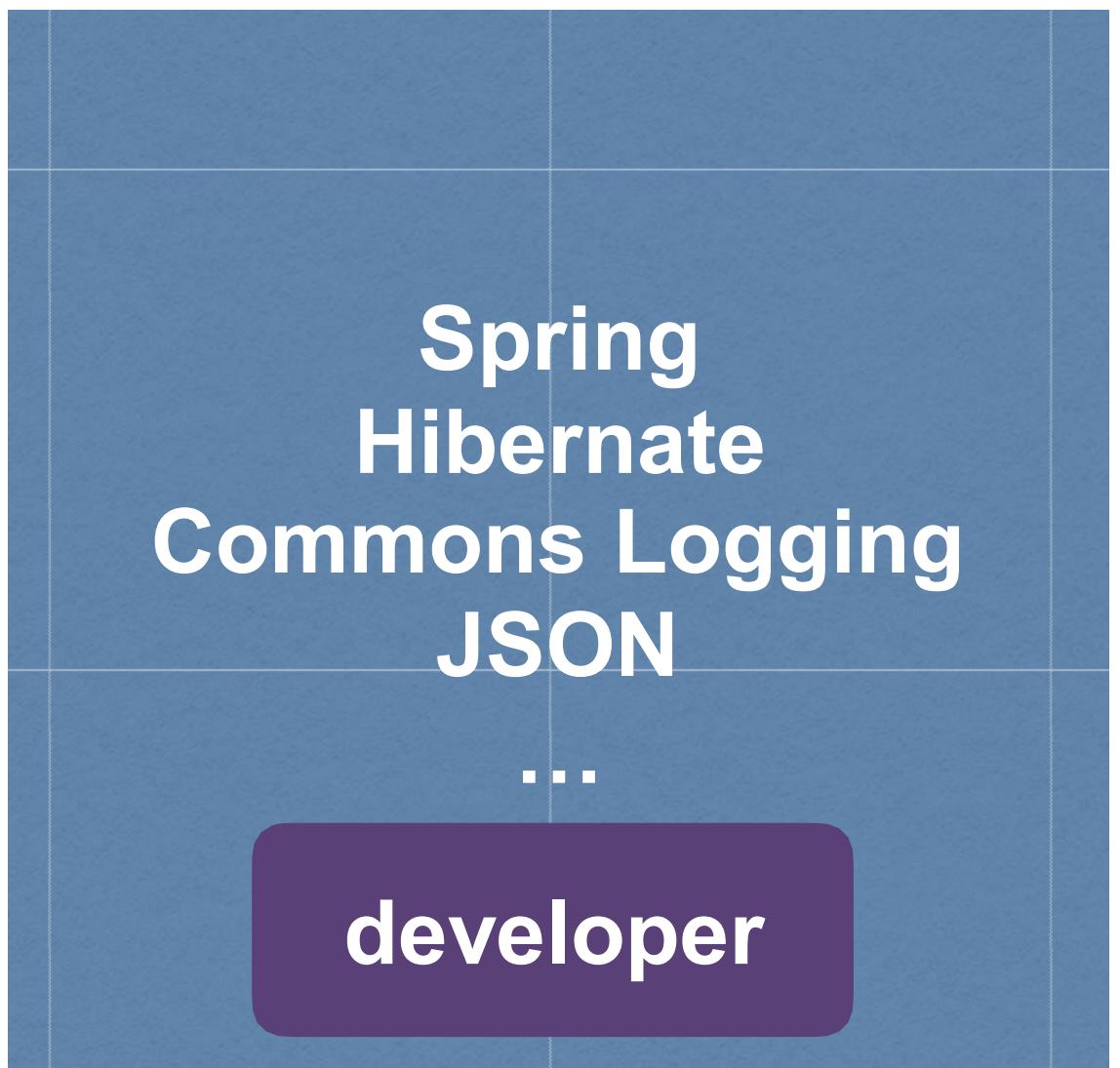
Comunicare a Maven le dipendenze necessarie
Spring, Hibernate ecc



Maven scaricherà i file JAR e li renderà disponibili durante la compilazione/esecuzione

Il mio progetto con Maven

My App



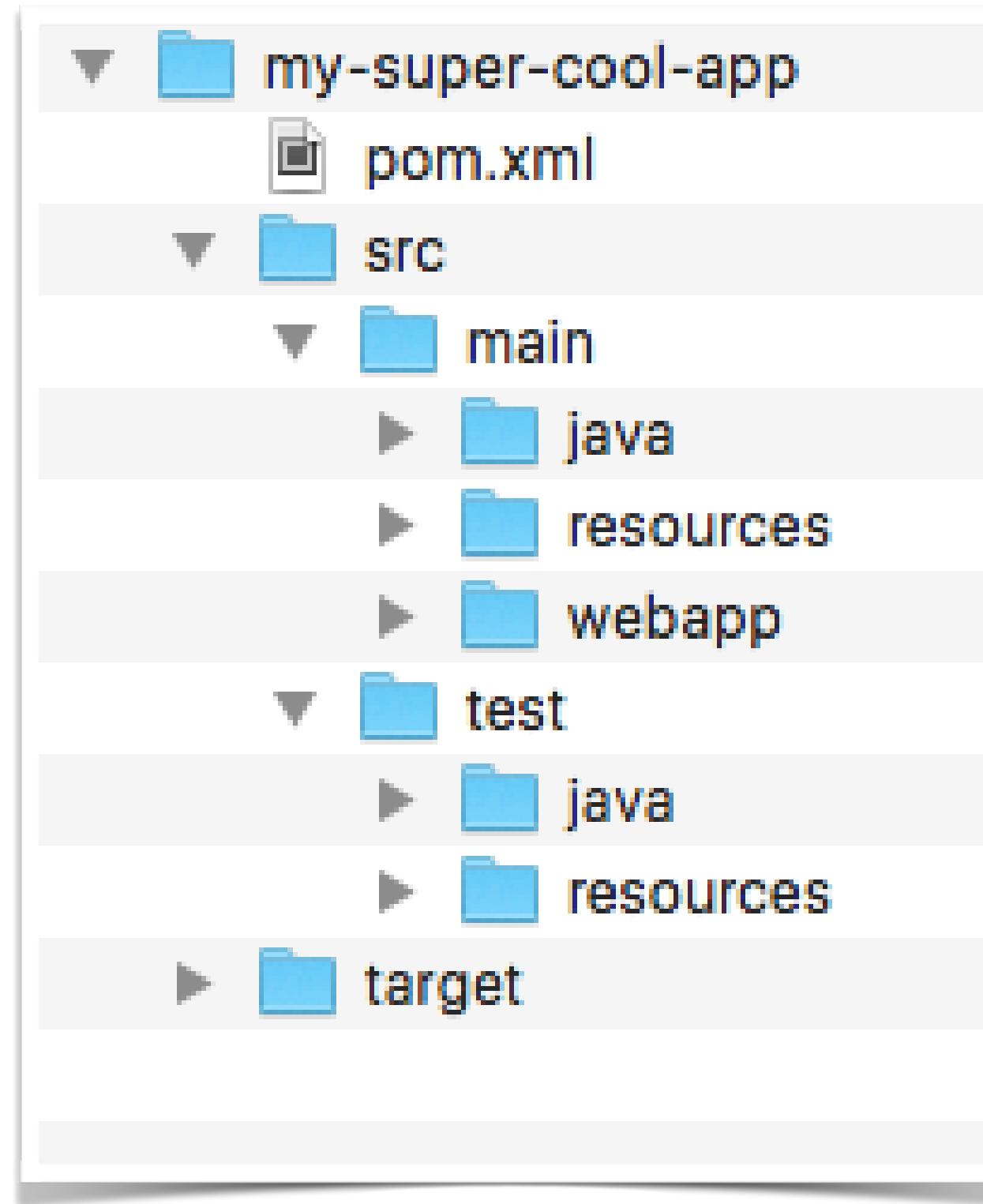
Gestione delle dipendenze

- ▶ Quando Maven recupera una dipendenza di progetto, scarica anche le dipendenze di supporto

Struttura directory standard

- ▶ Normalmente quando si partecipa a un nuovo progetto
- ▶ Ogni team di sviluppo presenta la propria struttura di directory
- ▶ Non ideale per i nuovi arrivati e non standardizzata
- ▶ **Maven risolve questo problema fornendo una struttura di directory standard**

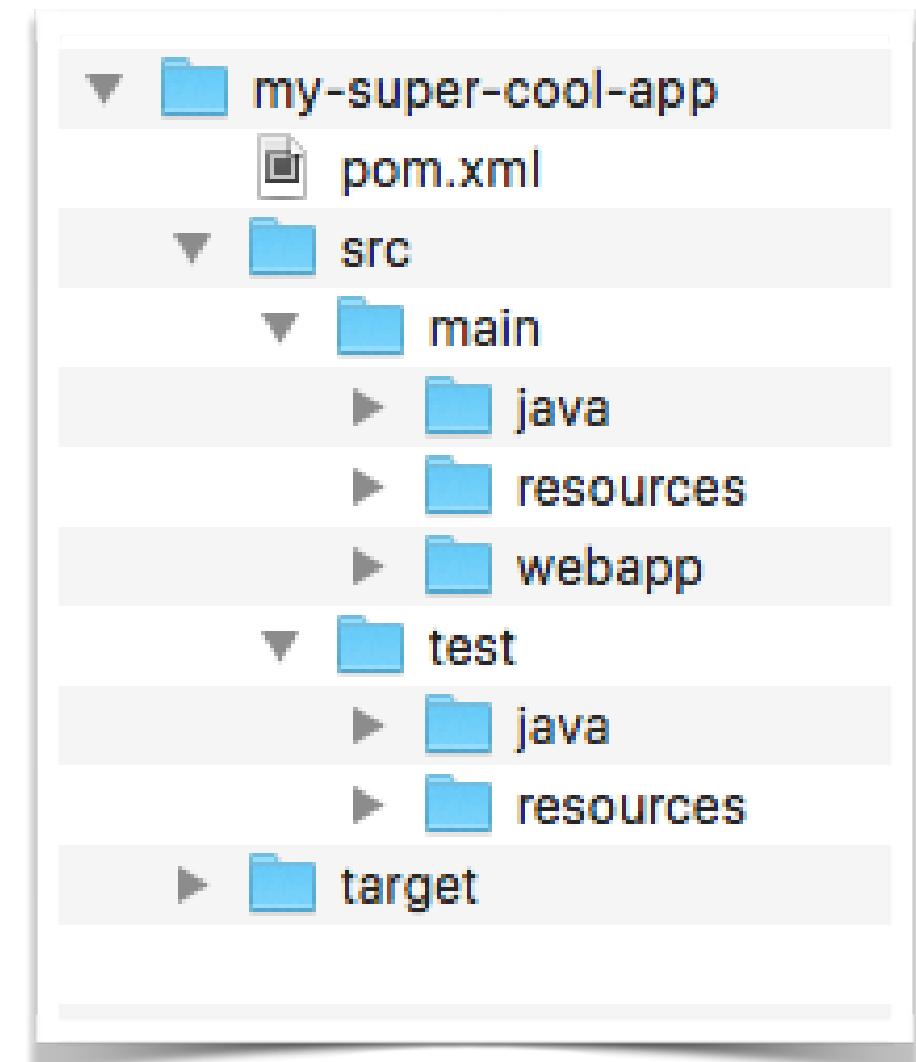
Standard Directory Structure



Directory	Description
src/main/java	Il codice sorgente Java
src/main/resources	Proprietà / file di configurazione utilizzati dall'app
src/main/webapp	File JSP e file di configurazione web, altre risorse web (immagini, css, js, ecc)
src/test	Codice e proprietà degli unit test
target	Directory di destinazione per il codice compilato. Creato automaticamente da Maven.

Vantaggi della struttura di directory standard

- ▶ I nuovi sviluppatori che aderiscono a un progetto
- ▶ Possono facilmente trovare codice, file di proprietà, unit test, file web ecc ...



Vantaggi della struttura di directory standard



Molti IDEs hanno support built-in per Maven



Eclipse, IntelliJ, NetBeans ecc



Gli IDE possono facilmente leggere/importare
progetti Maven

Vantaggi di Maven



Dependency Management



Maven troverà i file JAR per te



Niente più JAR mancanti



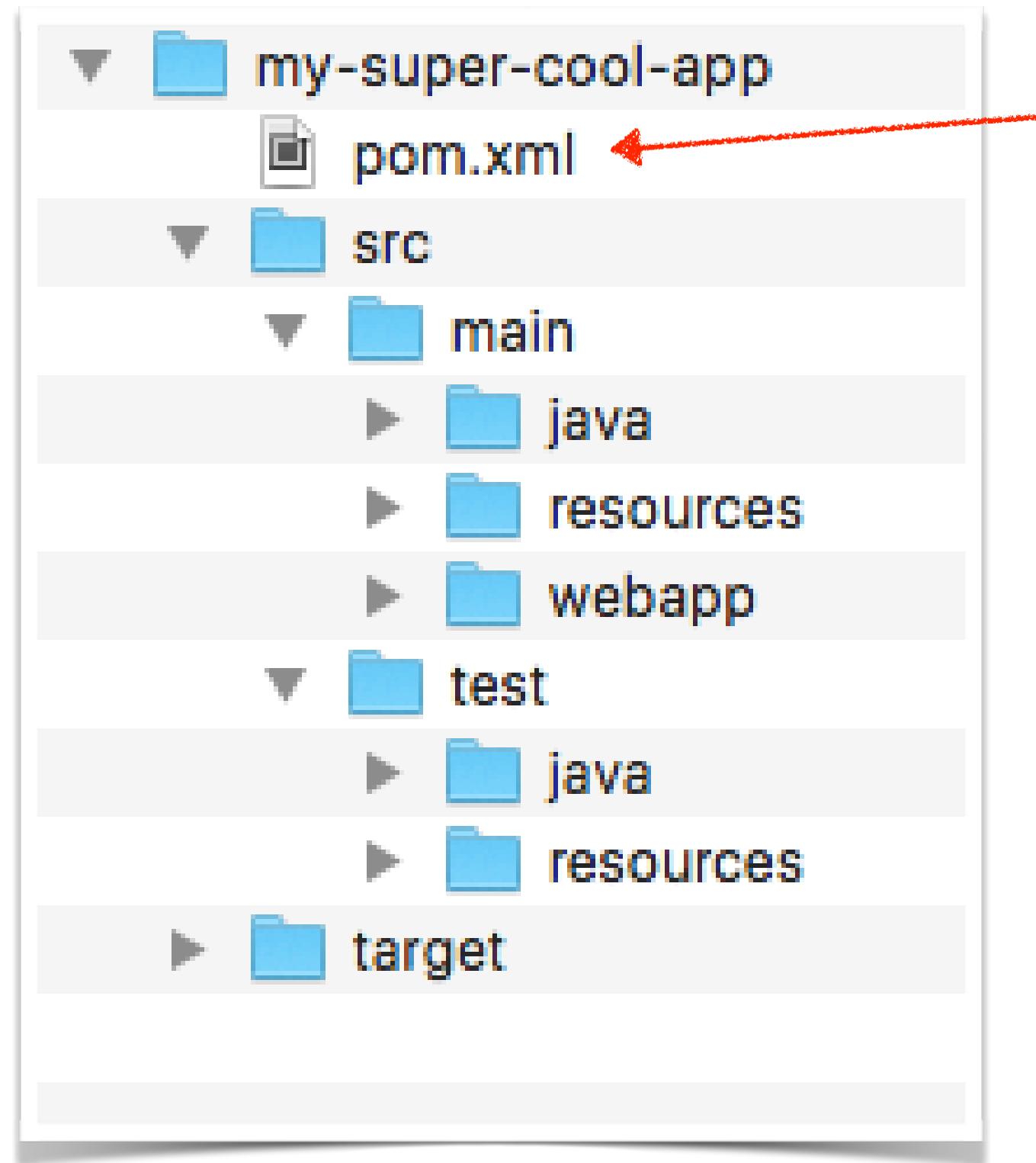
Compilazione ed esecuzione del progetto



Niente più problemi relativi a build path



Struttura di directory standard



File POM

File di configurazione situato nella radice del progetto Maven

POM File Structure



**Nome del progetto, versione
ecc, Tipo di file di output:
JAR, WAR, ...**

Elenco delle dipendenze

**Attività personalizzate
aggiuntive da eseguire: ad
esempio, unit test**

Coordinate del progetto

- ▶ Le coordinate del progetto identificano in modo univoco un progetto
- ▶ Simile alle coordinate GPS: latitudine / longitudine
- ▶ Informazioni precise per trovare una casa (città, strada, numero civico)

```
<groupId>it.examples</groupId>
<artifactId>example</artifactId>
<version>1.0.FINAL</version>
```

Coordinate progetto - Elementi

Name	Description
Group ID	Nome della società, del gruppo o dell'organizzazione. La convenzione consiste nell'utilizzare il nome di dominio inverso: <code>it.examples</code>
Artifact ID	Nome del progetto: example
Version	Una versione specifica come: 1.0, 1.6, 2.0 ... Se il progetto è in fase di sviluppo attivo, allora: 1.0-SNAPSHOT

```
<groupId>it.examples</groupId>
<artifactId>example</artifactId>
<version>1.0.FINAL</version>
```

Esempio di coordinate di progetto

```
<groupId>it.examples</groupId>
<artifactId>example</artifactId>
<version>1.0.RELEASE</version>
```

```
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>5.0.0.RELEASE</version>
```

```
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>5.2.11.Final</version>
```

Aggiunta di dipendenze

```
<project ...>
...
<dependencies>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.0.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.2.11.Final</version>
    </dependency>
    ...
</dependencies>
</project>
```

Maven Archetypes



Gli archetypes possono essere utilizzati per creare nuovi progetti Maven



Un archetype contiene i file modello per un determinato progetto Maven

Da considerare come una raccolta di "file iniziali" per un progetto

archetype := starter project

Archetypes comuni

Archetype Artifact ID	Description
maven-archetype-quickstart	Un archetype per generare un progetto Maven di esempio.
maven-archetype-webapp	Un archetype per generare un progetto Maven / Webapp di esempio.

<http://maven.apache.org/archetypes>

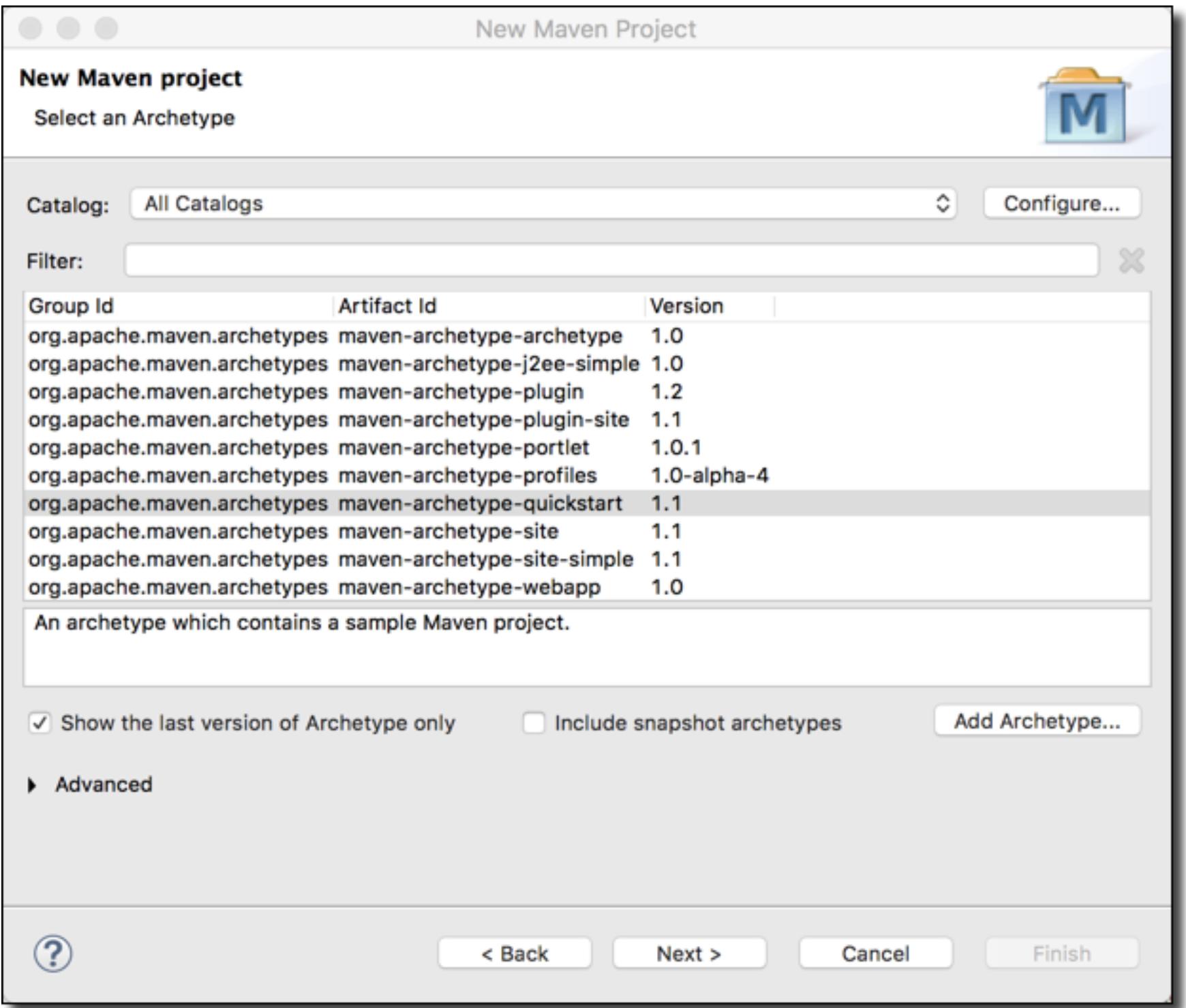
Archetypes

È possibile creare nuovi progetti utilizzando Maven Archetypes (progetto di avvio)

Dalla riga di comando con Maven

Da un IDE

Eclipse, IntelliJ, NetBeans etc...

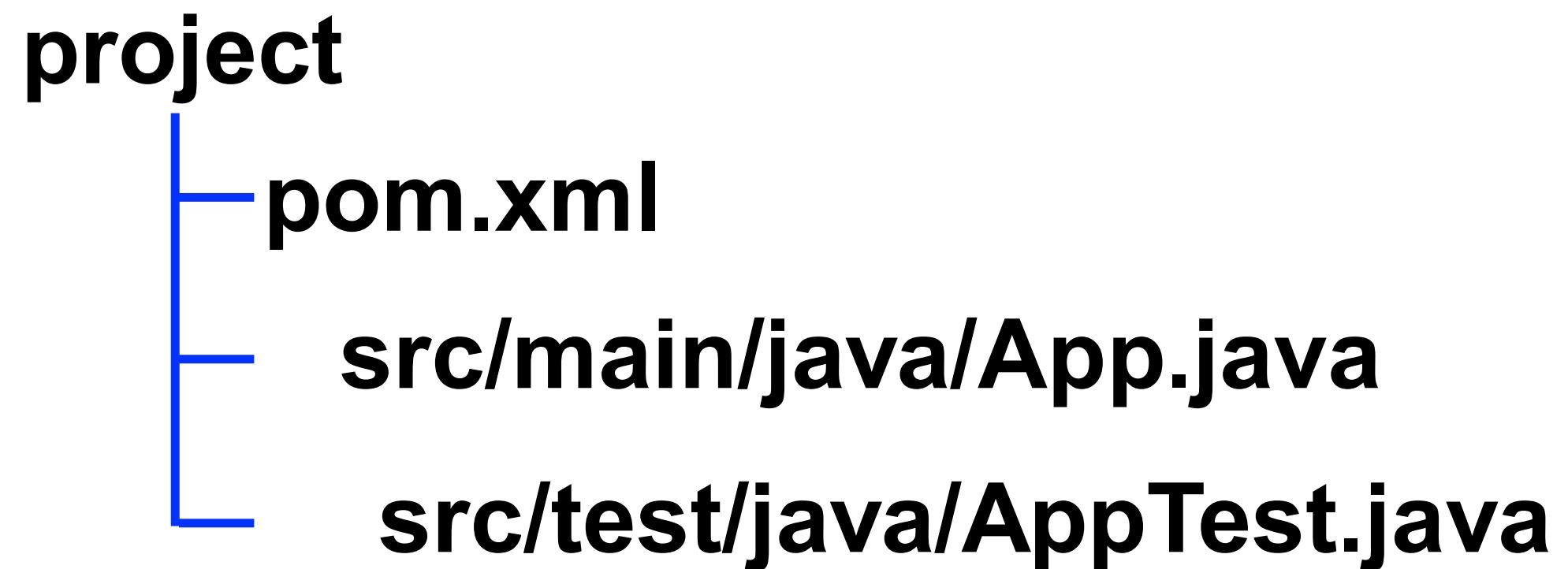


Lista di Archetypes - Eclipse

File > New Maven Project

Quickstart Archetype

- maven-archetype-quickstart contiene un progetto Maven di esempio

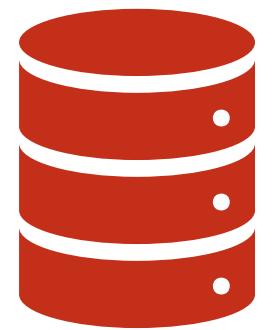


Webapp Archetype

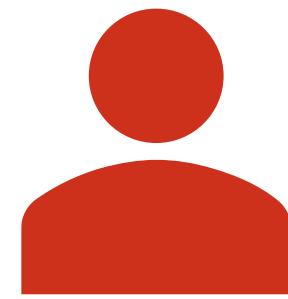
- maven-archetype-webapp contiene un progetto webapp Maven di esempio



Tipi di repository



Local Repository



Central Repository

Local Repository

- ▶ Situato sul computer dello sviluppatore
 - ▶ MS Windows: c:\Users\<users-home-dir>\.m2\repository
 - ▶ Mac e Linux: ~/.m2/repository
- ▶ Maven cercherà prima questa repository local...
- ▶ Prima di andare al repository centrale di Maven

Central Repository



Per impostazione predefinita, Maven cercherà nell'archivio centrale di Maven (remoto)



<https://repo.maven.apache.org/maven2/>



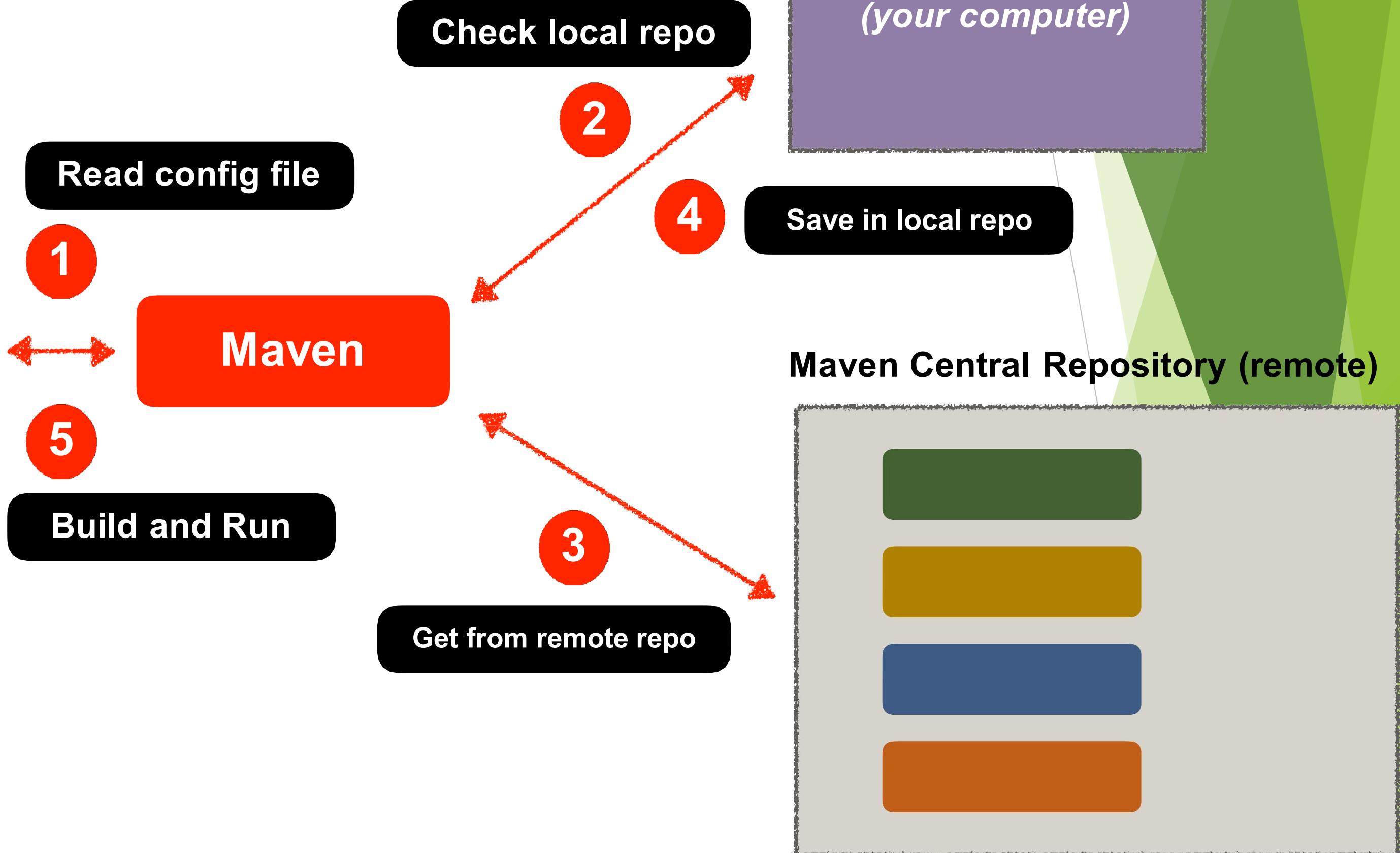
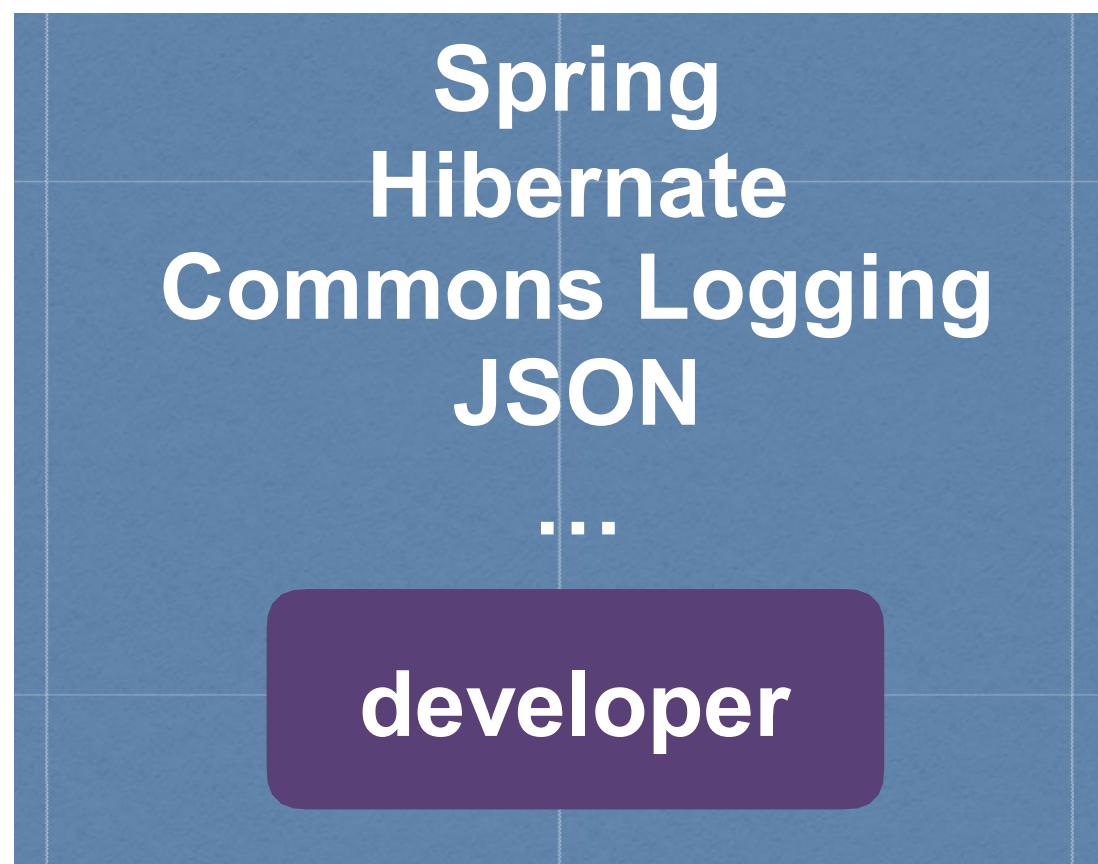
Richiede una connessione Internet



Una volta scaricati, i file vengono memorizzati nel repository locale

Maven - Come funziona

Project Config file



SPRING FRAMEWORK OVERVIEW

SPRING WEBSITE - OFFICIAL

www.spring.io

PERCHE' SPRING?

Per semplificare lo sviluppo JEE

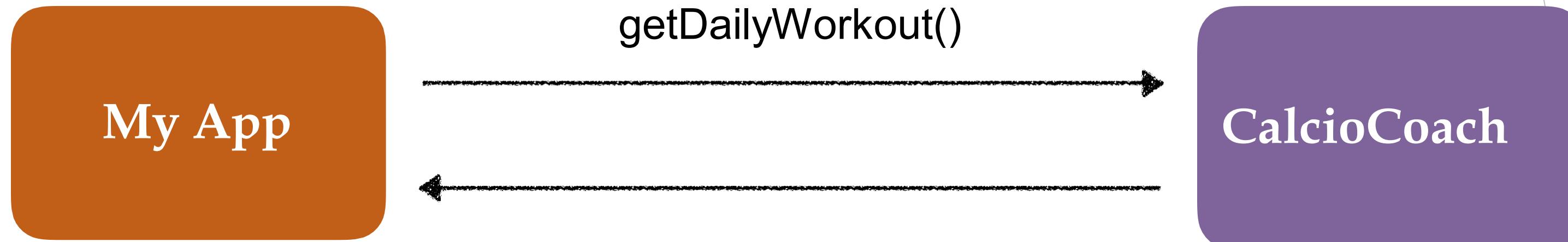
Obiettivi di Spring

- Sviluppo leggero con POJO Java (Plain-Old-Java-Objects)
- Iniezione di dipendenza per promuovere loose coupling
- Programmazione dichiarativa con programmazione orientata agli aspetti (AOP, Aspect-Oriented-Programming)
- Ridurre al minimo il codice Java boilerplate

Inversion of Control (IoC)

**Esternalizzare la costruzione e la
gestione degli oggetti.**

Scenario



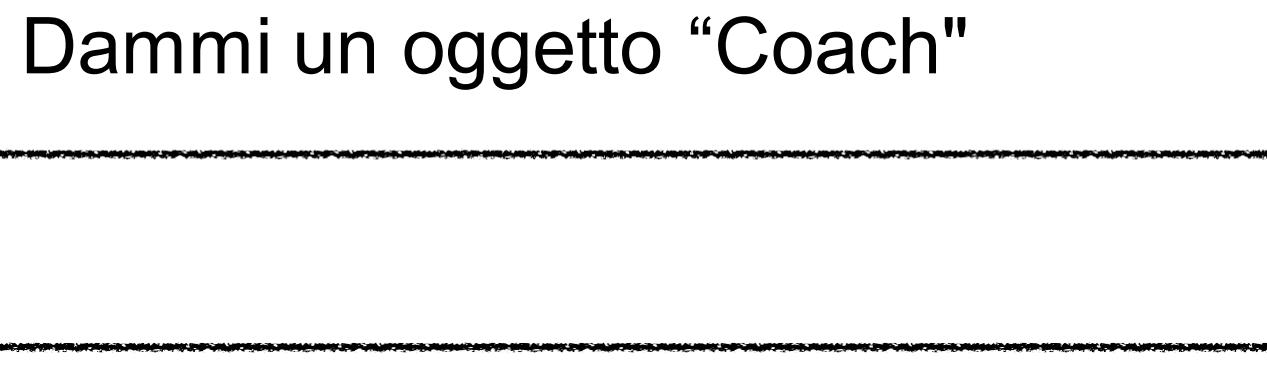
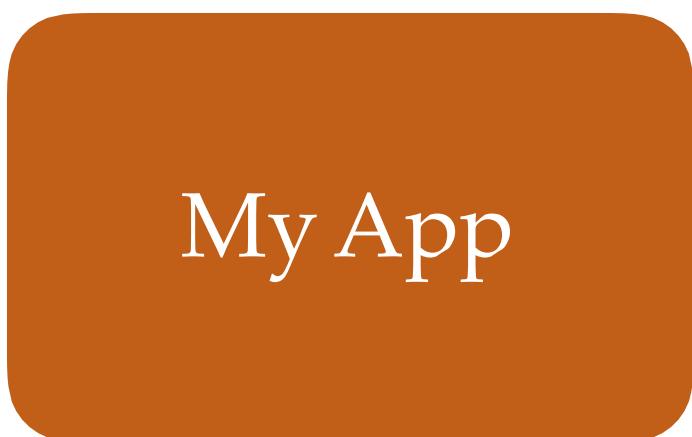
- L'app deve essere configurabile
- Si deve poter cambiare facilmente l'allenatore per un altro sport
 - Hockey, Cricket, Tennis, Baseball etc ...



Codice

- **MyApp.java**: metodo main
- **CalcioCoach.java**
- **Coach.java**: interfaccia
- **TrackCoach.java**

Soluzione ideale



```
public static void main(String[] args) {  
  
    // create the object  
    Coach theCoach = new TrackCoach();  
  
    // use the object  
    System.out.println(theCoach.getDailyWorkout());  
}
```



configuration

Dammi un oggetto "Coach"

Spring Container

- Funzioni primarie
 - Creare e gestire oggetti (*Inversion of Control*)
 - Iniettare le dipendenze dell'oggetto (*Dependency Injection*)

Spring
Object
Factory



Configurazione del container

- XML configuration file (*legacy, ma la maggior parte delle applicazioni usano ancora questo*)
- Java Annotations (*modern*)
- Java Source Code (*modern*)

IoC: come si procede?

Step-By-Step

1. Configurazione dei beans
2. Creazione di un oggetto container
3. Recupero dei beans dal container

Step 1: configurazione

File: applicationContext.xml

```
<beans ... >

    <bean id="myCoach"
          class="it.examples.BaseballCoach">
        </bean>

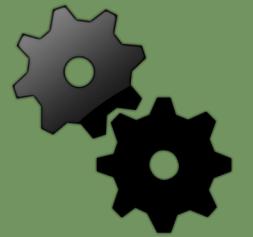
    </beans>
```

Step 2: creazione del container

- Il container è genericamente noto come **ApplicationContext**
- Implementazioni possibili:
 - ClassPathXmlApplicationContext
 - AnnotationConfigApplicationContext
 - GenericWebApplicationContext
 - altre ...

Spring

Object
Factory



Step 2: creazione del container

```
ClassPathXmlApplicationContext context =  
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

Step 3: recupero dei beans dal container

```
// create a spring container  
ClassPathXmlApplicationContext context =  
    new ClassPathXmlApplicationContext("applicationContext.xml");  
  
// retrieve bean from spring container  
Coach theCoach = context.getBean("myCoach", Coach.class);
```

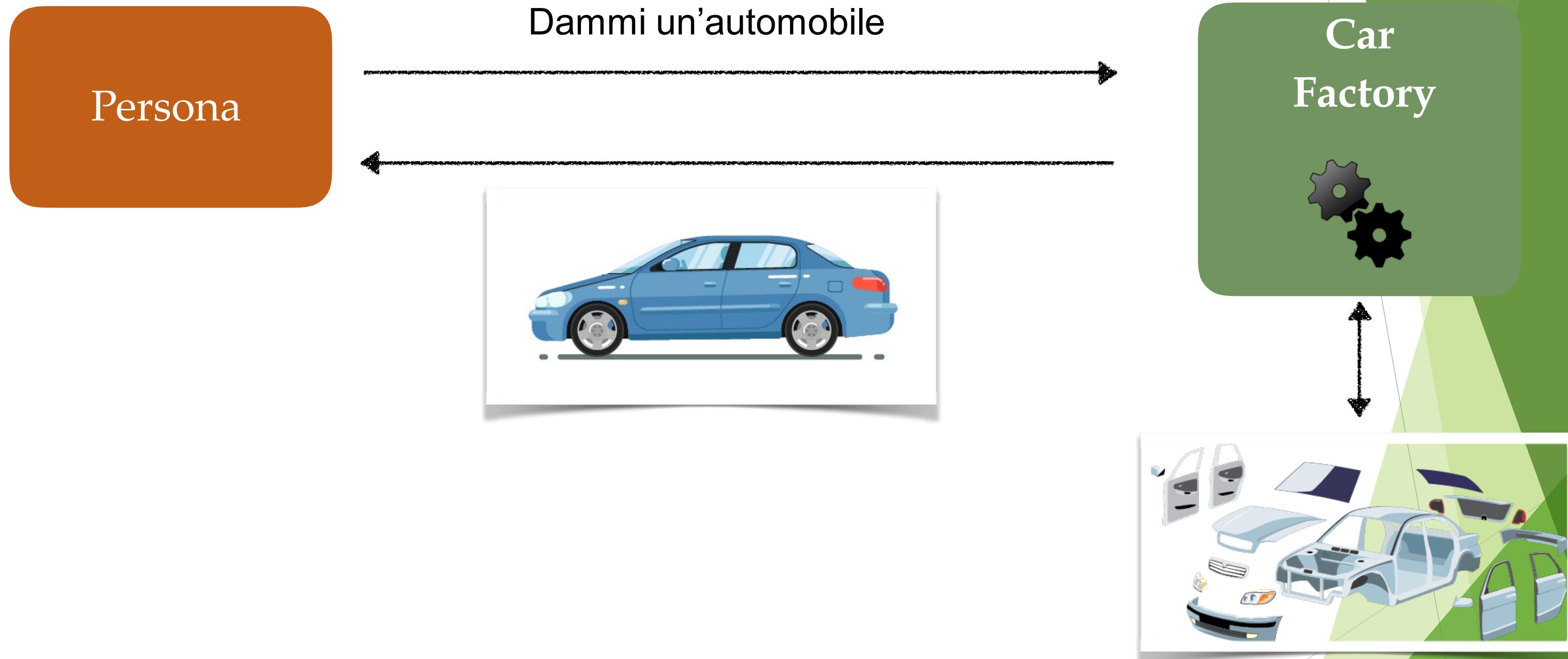
File: applicationContext.xml

```
<bean id="myCoach"  
      class="it.examples.BaseballCoach">  
</bean>
```

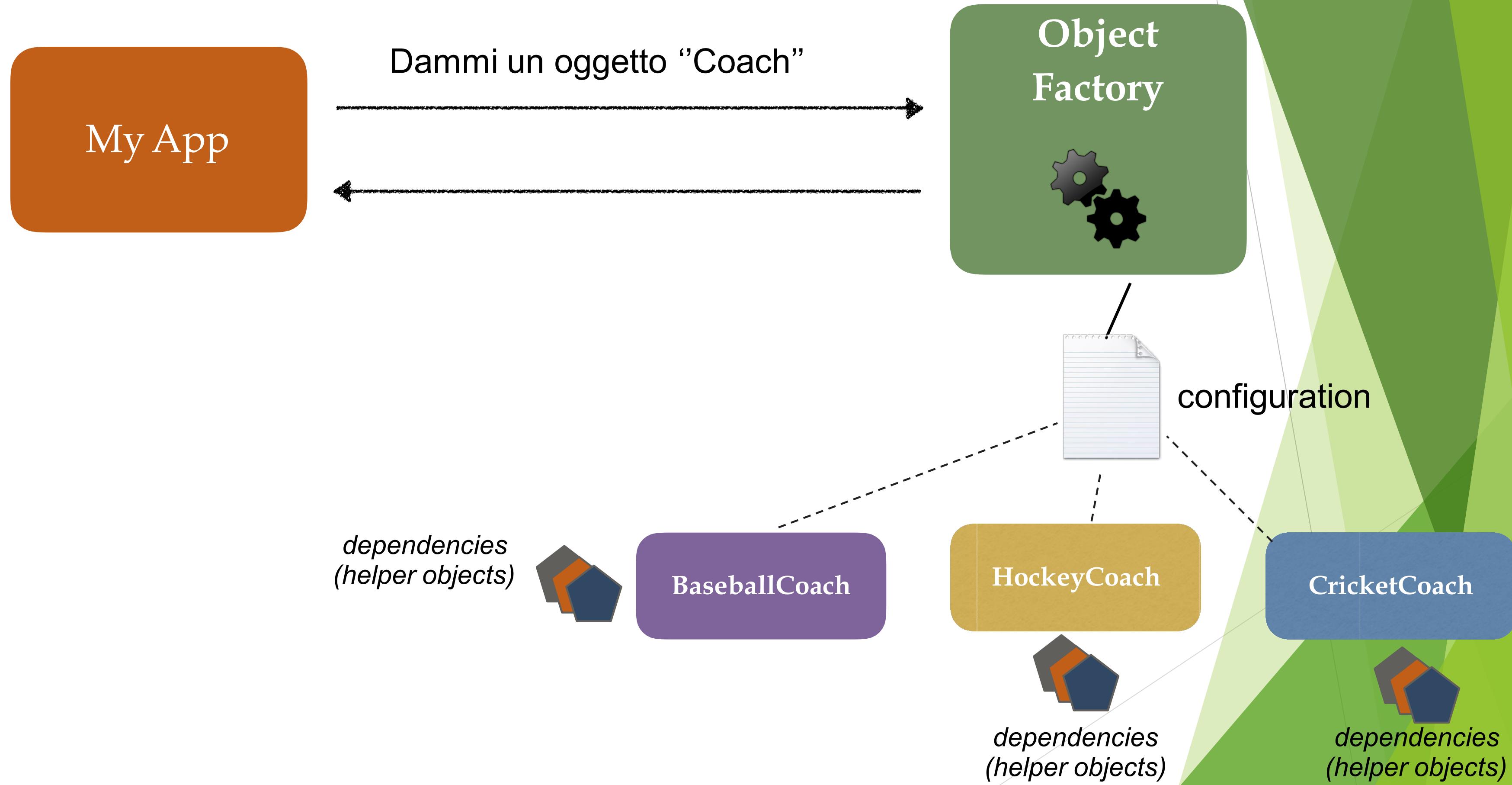
Dependency Injection

- ▶ Il client delega a Spring la responsabilità di fornire le relative dipendenze.

Car Factory

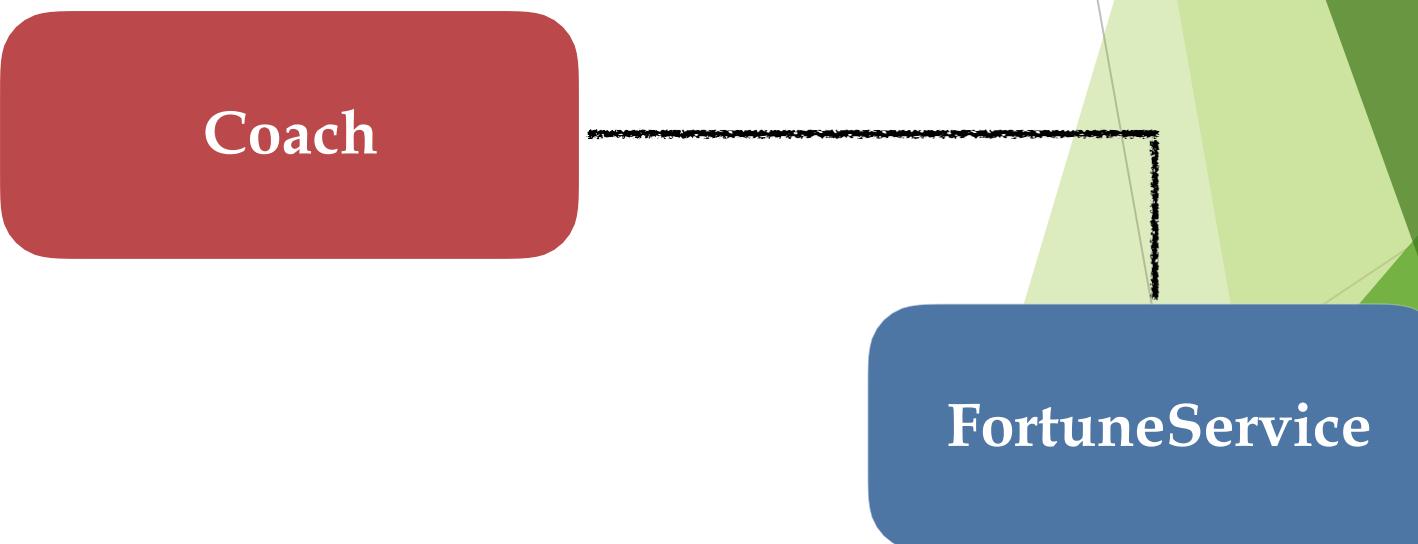


Spring Container



Esempio

- Il nostro Coach fornisce già allenamenti quotidiani
- Da adesso deve fornire anche frasi quotidiane sulla «fortuna»
- FortuneService: è una *dependency*



Tipi di injections

- Ci sono due tipi di iniezione con Spring
 - Constructor Injection
 - Setter Injection

Constructor Injection: come si procede?

1. Definire l'interfaccia della dipendenza e la relativa classe
2. Creare un costruttore nella classe per le iniezioni
3. Configurare l'inserimento delle dipendenze nel file di configurazione di Spring

Step-By-Step

Step 1: definire interfaccia e classe

File: FortuneService.java

```
public interface FortuneService {  
  
    public String getFortune();  
  
}
```

File: HappyFortuneService.java

```
public class HappyFortuneService implements FortuneService {  
  
    public String getFortune() {  
        return "Today is your lucky day!";  
    }  
}
```

Step 2: creazione di un costruttore per l'injection

File: BaseballCoach.java

```
public class BaseballCoach implements Coach {  
  
    private FortuneService fortuneService;  
  
    public BaseballCoach(FortuneService theFortuneService) {  
        fortuneService = theFortuneService;  
    }  
    ...  
}
```

Step 3: configurazione della dipendenza nel config file

File: applicationContext.xml

```
<bean id="myFortuneService"
      class="it.examples.HappyFortuneService">
</bean>

<bean id="myCoach"
      class="it.examples.BaseballCoach">
    <constructor-arg ref="myFortuneService" />
</bean>
```

Come si comporta Spring?

File: applicationContext.xml

```
<bean id="myFortuneService"
      class="it.examples.HappyFortuneService">
</bean>

<bean id="myCoach"
      class="it.examples.BaseballCoach">
    <constructor-arg ref="myFortuneService" />
</bean>
```

Come si comporta Spring?

```
<bean id="myFortuneService"
      class="it.examples.HappyFortuneService">
</bean>
```

```
<bean id="myCoach"
      class="it.examples.BaseballCoach">
    <constructor-arg ref="myFortuneService" />
</bean>
```

Spring Framework

```
HappyFortuneService myFortuneService =
    new HappyFortuneService();
```

```
BaseballCoach myCoach =
    new BaseballCoach(myFortuneService);
```

Setter Injection: come si procede?

1. Creare metodi setter nella classe per le iniezioni
2. Configurare l'inserimento delle dipendenze nel file di configurazione di Spring

Step-By-Step

Step1: creazione metodi setter

File: CricketCoach.java

```
public class CricketCoach implements Coach {  
  
    private FortuneService fortuneService;  
  
    public CricketCoach() {  
    }  
  
    public void setFortuneService(FortuneService fortuneService) {  
        this.fortuneService = fortuneService;  
    }  
    ...  
}
```

Step 2: configurazione delle dipendenze

File: applicationContext.xml

```
<bean id="myFortuneService"
      class="it.examples.HappyFortuneService">
</bean>

<bean id="myCricketCoach"
      class="it.examples.CricketCoach">

    <property name="fortuneService" ref="myFortuneService" />

</bean>
```

```
<property name="fortuneService" ref="myFortuneService" />
```

```
public void setFortuneService(...)
```

```
<property name="bestAthlete" ref="..."/>
```

```
public void setBestAthlete(...)
```

Come si comporta Spring?

```
<bean id="myFortuneService"
      class="it.examples.HappyFortuneService">
</bean>
```

Spring Framework

```
HappyFortuneService myFortuneService =
    new HappyFortuneService();
```

```
<bean id="myCricketCoach"
      class="it.examples.CricketCoach">

    <property name="fortuneService" ref="myFortuneService" />
</bean>
```

Spring Framework

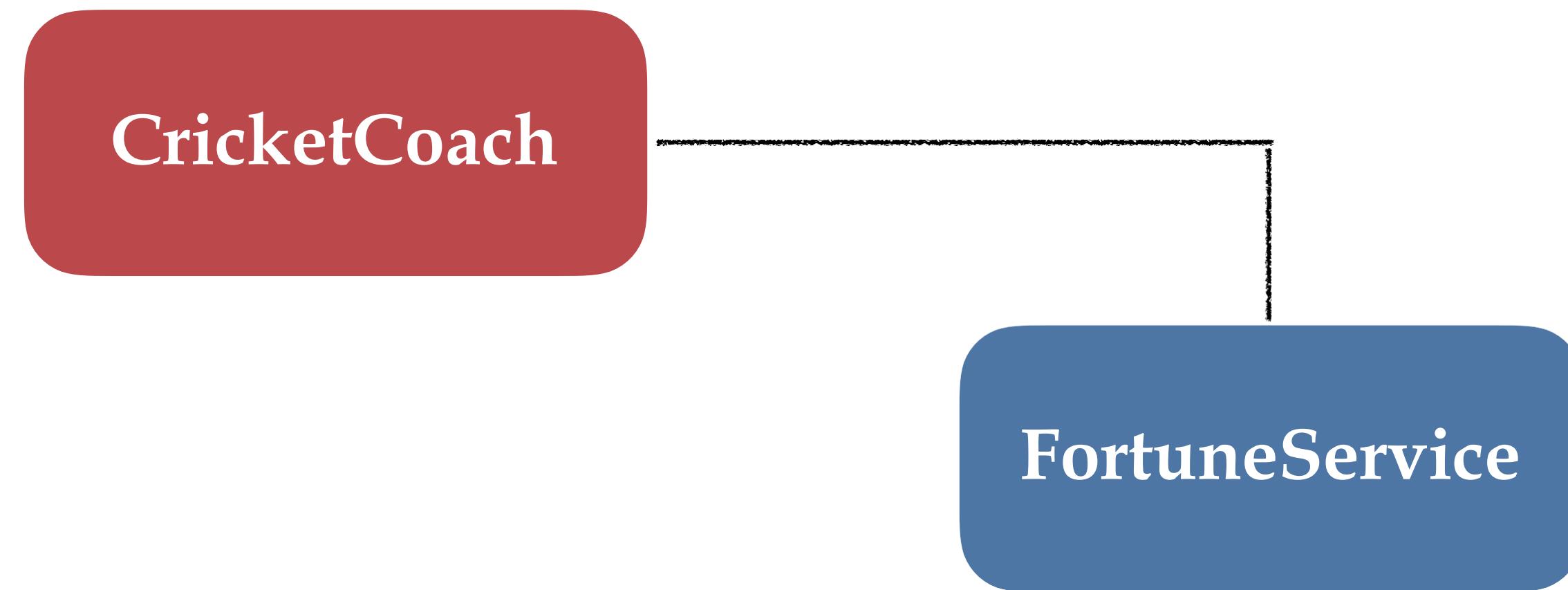
```
CricketCoach myCricketCoach =
    new CricketCoach();
```

```
myCricketCoach.setFortuneService(myFortuneService);
```

Iniezione di valori

emailAddress: test@test.com

team: TeamScatena



Iniezione di valori: come si procede?

1. Creare metodi setter nella classe per le iniezioni
2. Configurare l'inserimento dei valori nel file di configurazione di Spring

Step-By-Step

Step1: creazione metodi setter

File: CricketCoach.java

```
public class CricketCoach implements Coach {  
  
    private String emailAddress;  
    private String team;  
  
    public void setEmailAddress(String emailAddress) ...  
  
    public void setTeam(String team) ...  
  
    ...  
}
```

Step 2: configurazione dell'iniezione

File: applicationContext.xml

```
<bean id="myCricketCoach"
      class="it.examples.CricketCoach">

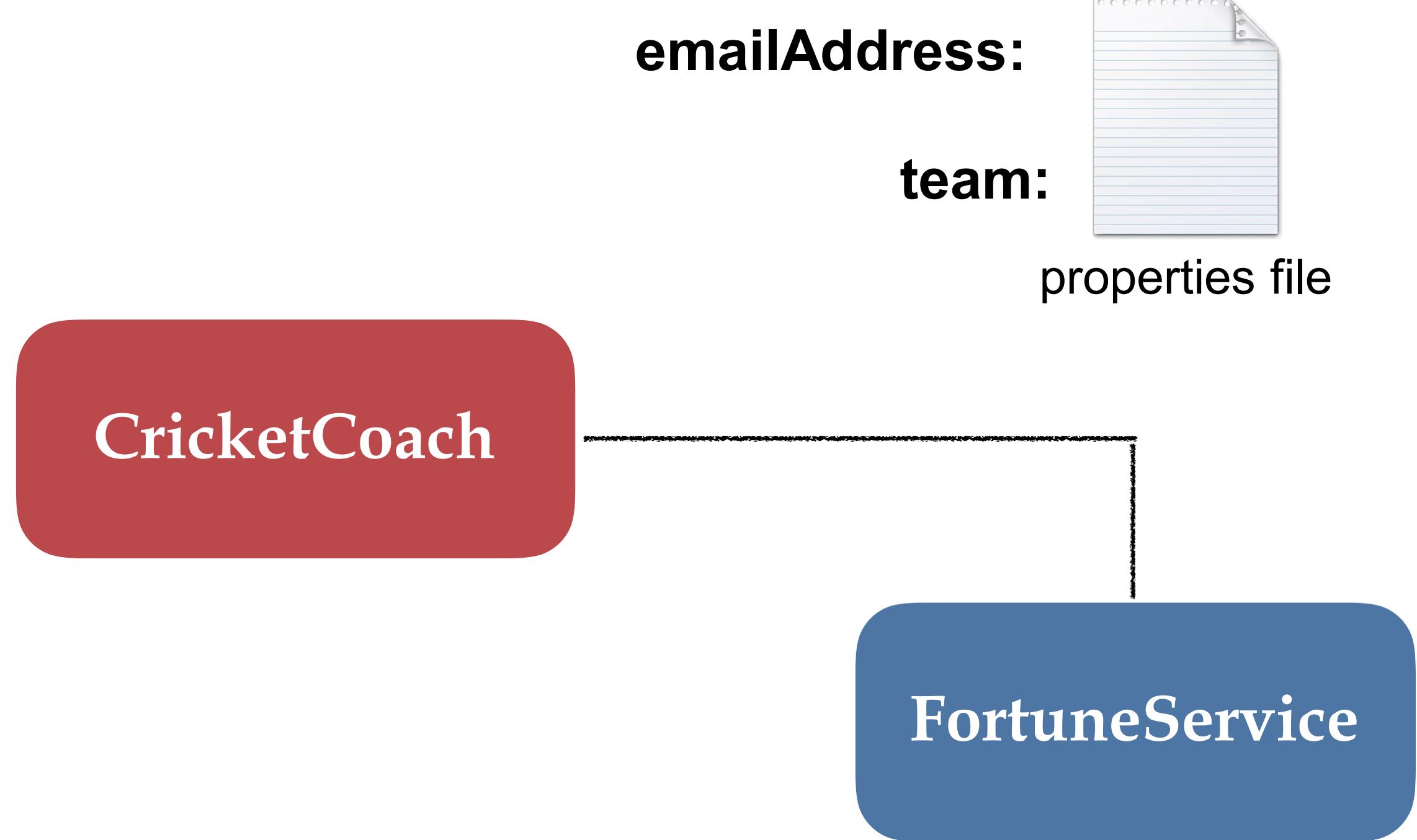
    <property name="fortuneService" ref="myFortuneService" />

    <property name="emailAddress" value="test@test.it" />

    <property name="team" value="TeamScatena" />

</bean>
```

Lettura dei valori da un file .properties



Lettura dei valori da un file .properties: come si procede?

1. Creare file properties
2. Caricare il file properties nel file di configurazione di Spring
3. Inserire i valori di riferimento dal file properties

Step-By-Step

Step 1: creazione file properties

File: sport.properties

```
foo.email=test@test.it  
foo.team=TeamScatena
```

Step 2: caricamento del properties nel file di configurazione

File: applicationContext.xml

```
<context:property-placeholder location="classpath:sport.properties"/>
```

Step 3: inserimento dei valori presi dal properties

File: applicationContext.xml

```
<bean id="myCricketCoach"
      class="it.examples.CricketCoach">
    ...
    <property name="emailAddress" value="${foo.email}" />
    <property name="team" value="${foo.team}" />
</bean>
```

foo.email=test@test.it

foo.team=TeamScatena

Bean Scopes

- Lo scope si riferisce al ciclo di vita di un bean
- Quanto tempo «vive» un bean?
- Quante istanze vengono create?
- Come viene condiviso il bean?

Default Scope: Singleton

```
<beans ...>

<bean id="myCoach"
      class="it.examples.TrackCoach"
```

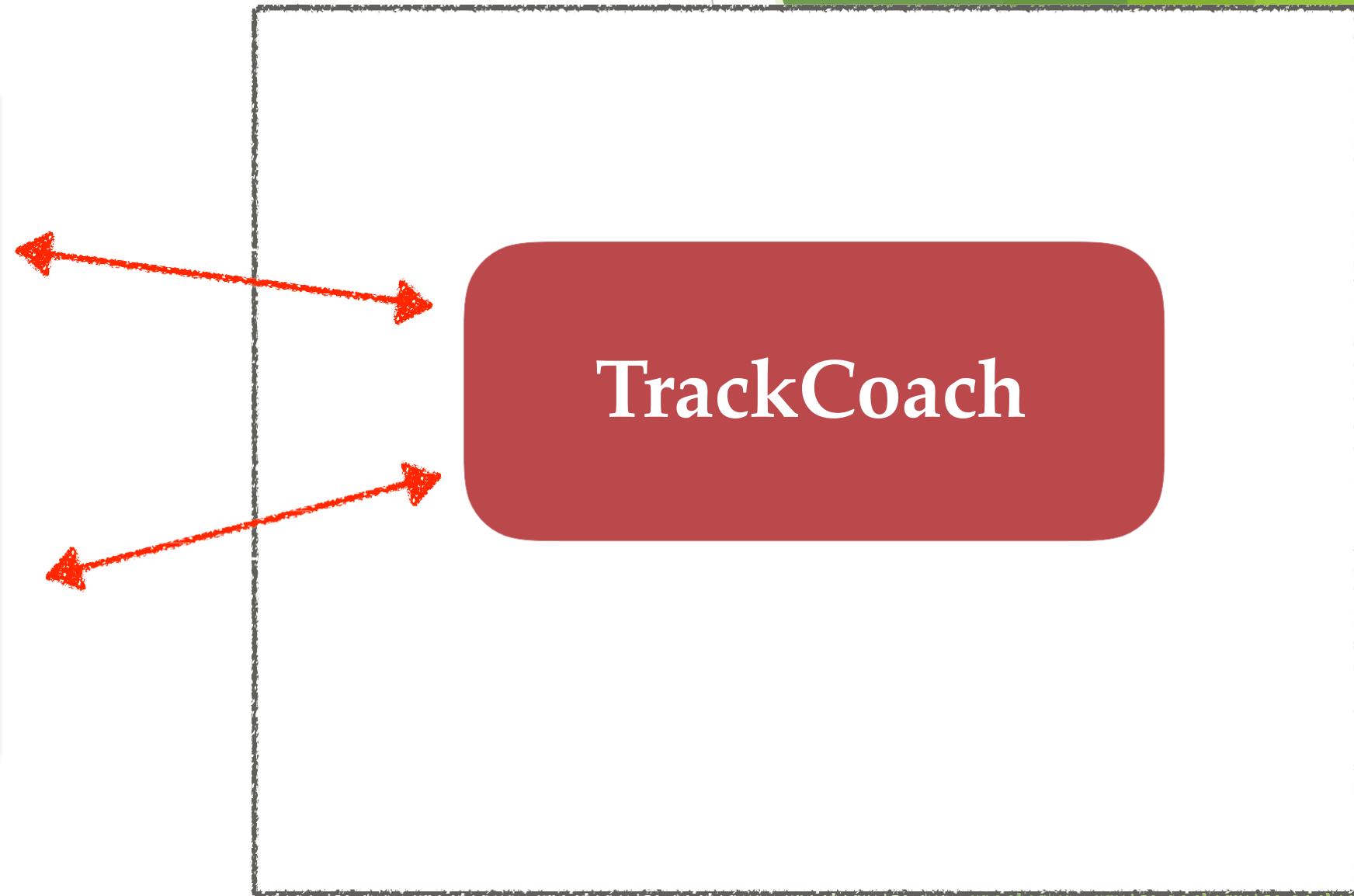
Cos'è un singleton?

- Di default, container crea una sola istanza del bean
- Viene memorizzato nella cache di memoria
- Tutte le richieste per un determinato bean restituiranno un riferimento allo stesso bean condiviso

Cos'è un Singleton?

Spring

```
Coach theCoach = context.getBean("myCoach", Coach.class);  
...  
Coach alphaCoach = context.getBean("myCoach", Coach.class);
```



Esplicitare lo scope di un bean

```
<beans ... >

<bean id="myCoach"
      class="it.examples.TrackCoach"
      scope="singleton"
```

Lista di scopes

Scope	Description
singleton	Singola istanza condivisa del bean
prototype	Crea una nuova istanza del bean per ogni richiesta

Prototype Scope

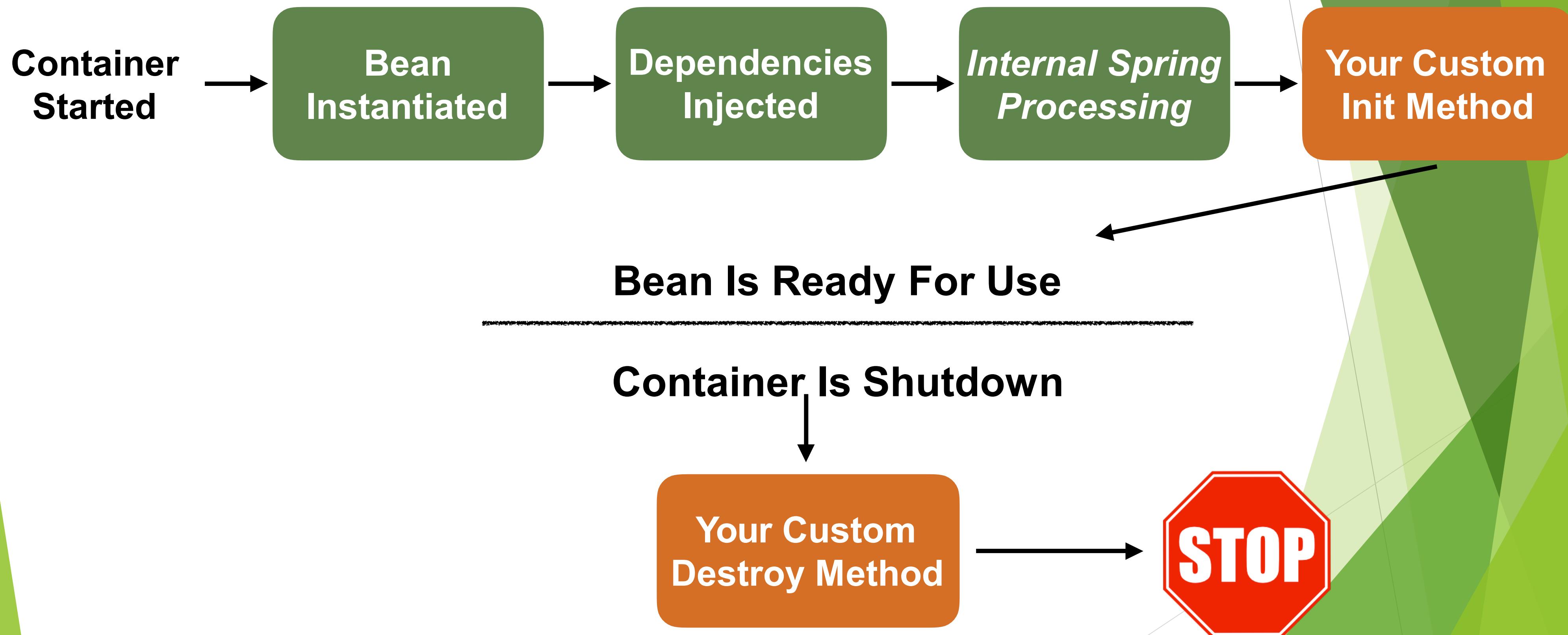
Prototype scope: nuovo oggetto per ogni richiesta

```
<beans ... >

    <bean id="myCoach"
          class="it.examples.TrackCoach"
          scope="prototype">
        ...
    </bean>

</beans>
```

Bean Lifecycle



Bean Lifecycle: metodi

- È possibile aggiungere codice personalizzato durante l'inizializzazione di bean
 - Per eventuali chiamate alla logica di business o per configurare la connessione alle risorse dati
- È possibile aggiungere codice personalizzato durante la distruzione del fano
 - Per eventuali chiamate alla logica di business o per chiudere le connessioni alle risorse dati

Init: configurazione

```
<beans ... >

<bean id="myCoach"
      class="it.examples.TrackCoach"
      init-method="doMyStartupStuff">
    ...
</bean>

</beans>
```

Destroy: configurazione

```
<beans ...>

<bean id="myCoach"
      class="it.examples.TrackCoach"
      init-method="doMyStartupStuff"
      destroy-method="doMyCleanupStuff">
  ...
</bean>

</beans>
```

Come si procede?

1. Definire i metodi custom per init e destroy
2. Configurare i nomi dei metodi definiti nel file di configurazione

Step-By-Step

Che cosa sono le annotazioni Java?

- Etichette/marcatori speciali aggiunti alle classi Java
- Forniscono metadati relativi alla classe
- Processate in fase di compilazione o in fase di esecuzione per un'elaborazione «speciale»

Boot

Color: Silver

Style: Jewel

Code: 1460

SKU: 10072090

Size US: 8

Size UK: 6

Perché configurare con le annotazioni?

- La configurazione XML può essere troppo onerosa
- Le annotazioni riducono al minimo la configurazione XML

Scanning delle classi

- Spring eseguirà uno scan delle classi Java alla ricerca di annotazioni speciali
- In seguito registrerà automaticamente i bean all'interno del container

Come si procede?

1. Abilitare lo scanning dei componenti nel file di config
2. Aggiungere l'annotazione `@Component` alle classi Java
3. Recuperare i bean dal context

Step-By-Step

Step 1: abilitare scanning dei componenti

```
<beans ... >  
  
<context:component-scan base-package="it.examples" />  
  
</beans>
```

Step 2: aggiungere annotazione @Component nelle classi

```
@Component("thatSillyCoach")
public class TennisCoach implements Coach {

    @Override
    public String getDailyWorkout() {
        return "Practice your backhand volley";
    }

}
```

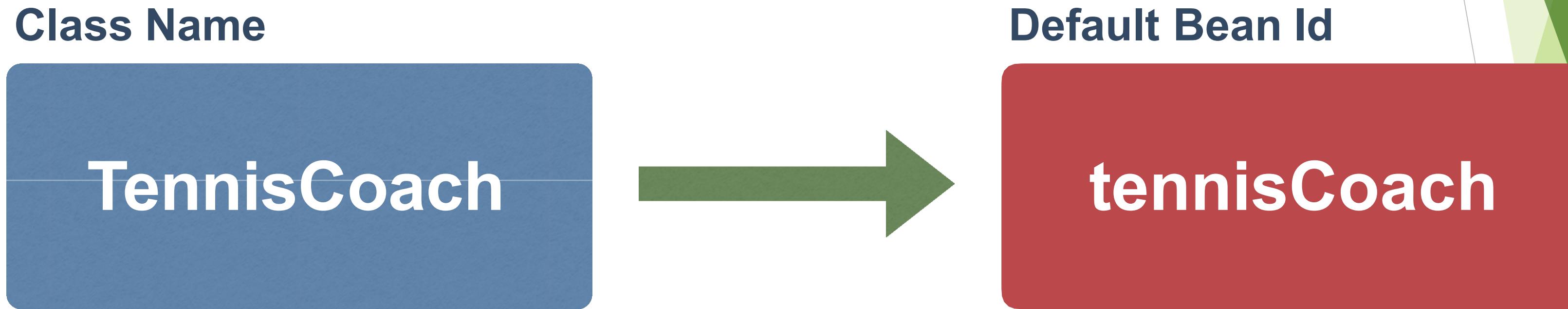
Step 3: recuperare i bean dal container

- Stesso codice di prima...

```
Coach theCoach = context.getBean("thatSillyCoach", Coach.class);
```

Spring supporta anche ID di default

- Default bean id: il nome della classe, in camel-case



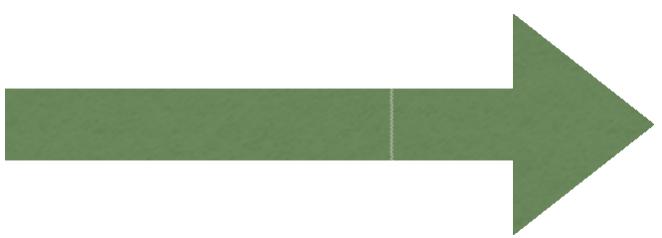
Esempio

```
@Component  
public class TennisCoach implements Coach {
```

Class Name



Default Bean Id



Esempio

```
@Component  
public class TennisCoach implements Coach {
```

```
// get the bean from spring container  
Coach theCoach = context.getBean("tennisCoach", Coach.class);
```

Esempio

```
@Component  
public class TennisCoach implements Coach {
```

Class Name

HappyFortuneService

Default Bean Id

happyFortuneService



Cos'è l'AutoWiring?

Coach

FortuneService

- Per implementare dependency injection, Spring può usare il meccanismo di autowiring
- Spring, in caso di autowiring, cerca un bean che abbia lo stesso tipo dell'oggetto che si sta cercando di iniettare.
- Una volta trovato, lo inietta automaticamente.

Esempio di autowiring

Coach

FortuneService

- Inserimento di FortuneService con autowiring nell'implementazione di un Coach
- Spring eseguirà lo scan delle classi @Component
- C'è una classe che implementa FortuneService?
- In quel caso, allora, la inietta.

Tipi di autowiring

- Constructor Injection
- Setter Injection
- Field Injections

Constructor Injection: come si procede?

1. Definire l'interfaccia e la classe della dipendenza
2. Creare un costruttore nella classe in cui bisogna eseguire l'injection
3. Configurare l'iniezione attraverso l'annotazione @Autowired

Step-By-Step

Step 1: definire l'interfaccia e la classe della dipendenza

File: FortuneService.java

```
public interface FortuneService {  
  
    public String getFortune();  
  
}
```

File: HappyFortuneService.java

```
@Component  
public class HappyFortuneService implements FortuneService {  
  
    public String getFortune() {  
        return "Today is your lucky day!";  
    }  
}
```

Step 2: creare un costruttore per l'injection

File: TennisCoach.java

```
@Component  
public class TennisCoach implements Coach {  
  
    private FortuneService fortuneService;  
  
    public TennisCoach(FortuneService theFortuneService) {  
        fortuneService = theFortuneService;  
    }  
    ...  
}
```

Coach

FortuneService

Step 3: configurare l'injection attraverso @Autowired

File: TennisCoach.java

```
@Component  
public class TennisCoach implements Coach {  
  
    private FortuneService fortuneService;  
  
    @Autowired  
    public TennisCoach(FortuneService theFortuneService) {  
        fortuneService = theFortuneService;  
    }  
    ...  
}
```



Setter Injection: come si procede?

1. Creare metodi setter nella classe per le iniezioni
2. Configurare l'inserimento delle dipendenze con @Autowire

Step-By-Step

Step1: creare metodi setter

File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

    private FortuneService fortuneService;

    public TennisCoach() {
    }

    public void setFortuneService(FortuneService fortuneService) {
        this.fortuneService = fortuneService;
    }
    ...
}
```

Step 2: configurare dipendenza con @Autowired

File: TennisCoach.java

```
@Component
public class TennisCoach implements Coach {

    private FortuneService fortuneService;

    public TennisCoach() {
    }

    @Autowired
    public void setFortuneService(FortuneService fortuneService) {
        this.fortuneService = fortuneService;
    }
    ...
}
```

Field Injection: come si procede?

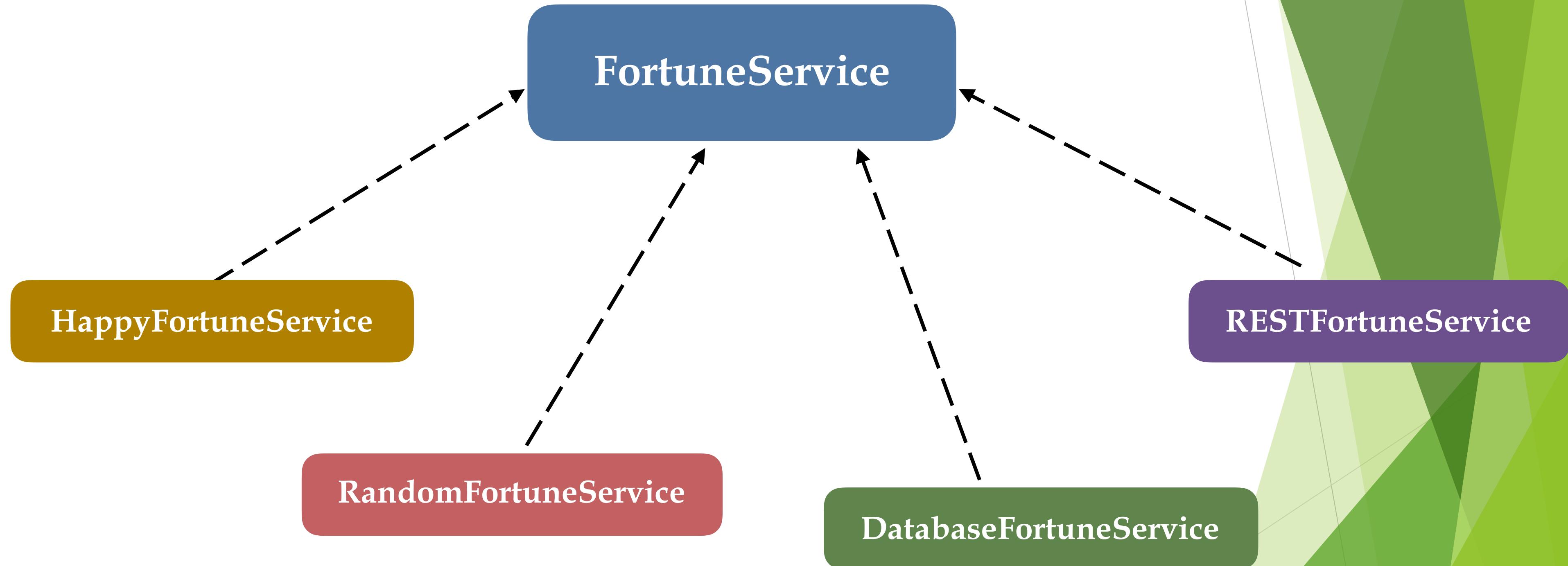
1. Configurare l'iniezione delle dipendenze con l'annotazione @Autowired

Step 1: configurazione delle dipendenze con l'annotazione @Autowired

File: TennisCoach.java

```
public class TennisCoach implements Coach {  
  
    @Autowired  
    private FortuneService fortuneService;  
  
    public TennisCoach() {  
    }  
  
    // no need for setter methods  
    ...  
}
```

Autowiring di una classe con molteplici impl.?



Si genera un'eccezione...

Error creating bean with name 'tennisCoach':
Injection of autowired dependencies failed

Caused by: [org.springframework.beans.factory.NoUniqueBeanDefinitionException](#):

No qualifying bean of type [it.examples.FortuneService] is defined: expected
single matching bean but found 4:

databaseFortuneService,happyFortuneService,randomFortuneService,RESTFortuneService

Soluzione: annotazione @Qualifier

```
@Component
public class TennisCoach implements Coach {
    @Autowired
    @Qualifier("happyFortuneService")
    private FortuneService fortuneService;
    ...
}
```

Scope: singleton

```
@Component  
@Scope("singleton")  
public class TennisCoach implements Coach {  
  
...  
}
```

Scope: prototype

```
@Component  
@Scope("prototype")  
public class TennisCoach implements Coach {  
  
    ...  
  
}
```

Lifecycle: configurazione metodo init

```
@Component
public class TennisCoach implements Coach {

    @PostConstruct
    public void doMyStartupStuff() { ... }

    ...
}
```

Lifecycle: configurazione metodo destroy

```
@Component
public class TennisCoach implements Coach {

    @PreDestroy
    public void doMyCleanupStuff() { ... }

    ...
}
```

Come si procede?

1. Definire metodi per init e destroy
2. Aggiungere annotazioni ai metodi creati:
@PostConstruct e @PreDestroy

Step-By-Step

Java Configuration

- Si configura il container Spring utilizzando direttamente codice Java, senza XML

No XML!

Come si procede?

Step-By-Step

1. Creare una classe con annotazione **@Configuration**
2. Specificare il component scanning: **@ComponentScan**
3. Leggere le configurazioni del container
4. Recuperare i beans dal container

Step 1: creare una classe con @Configuration

```
@Configuration
public class SportConfig {  

}  

}
```

Step 2: aggiungere @ComponentScan

```
@Configuration  
@ComponentScan("it.examples")  
public class SportConfig {  
  
}
```

Step 3: leggere le configurazioni

```
AnnotationConfigApplicationContext context =  
    new AnnotationConfigApplicationContext(SportConfig.class);
```

Step 3: recuperare i bean dal container

```
Coach theCoach = context.getBean("tennisCoach", Coach.class);
```

Come definire i bean con codice Java?

```
@Configuration  
public class SportConfig {  
  
    @Bean  
    public Coach swimCoach() {  
        SwimCoach mySwimCoach = new SwimCoach();  
  
        return mySwimCoach;  
    }  
  
}
```

In caso di iniezione di dipendenze?

```
@Configuration  
public class SportConfig {  
  
    @Bean  
    public FortuneService happyFortuneService() {  
        return new HappyFortuneService();  
    }  
  
    @Bean  
    public Coach swimCoach() {  
        SwimCoach mySwimCoach = new SwimCoach( happyFortuneService() );  
  
        return mySwimCoach;  
    }  
}
```

Il resto non cambia...

```
AnnotationConfigApplicationContext context =  
    new AnnotationConfigApplicationContext(SportConfig.class);
```

Il resto non cambia...

```
Coach theCoach = context.getBean("swimCoach", Coach.class);
```

Come iniettare variabili da un file .properties?

File: SportConfig.java

```
@Configuration  
{@PropertySource("classpath:sport.properties")  
public class SportConfig {  
  
    ...  
}}
```

Step 3: Reference Values from Properties File

File: **SwimCoach.java**

```
public class SwimCoach implements Coach {  
  
    @Value("${foo.email}")  
    private String email;  
  
    @Value("${foo.team}")  
    private String team;  
  
    ...  
}
```

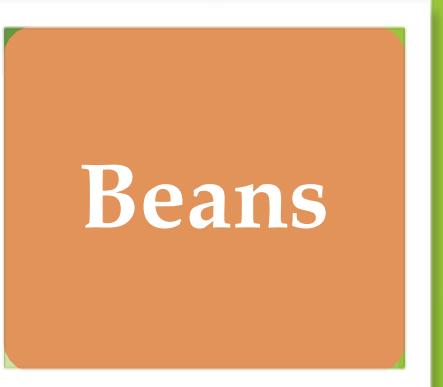
foo.email=test@test.it
foo.team=TeamScatena

The background features a large, abstract graphic composed of various shades of green. It consists of several overlapping triangles and trapezoids, creating a layered, polygonal effect. The colors range from bright lime green on the left to dark forest green on the right.

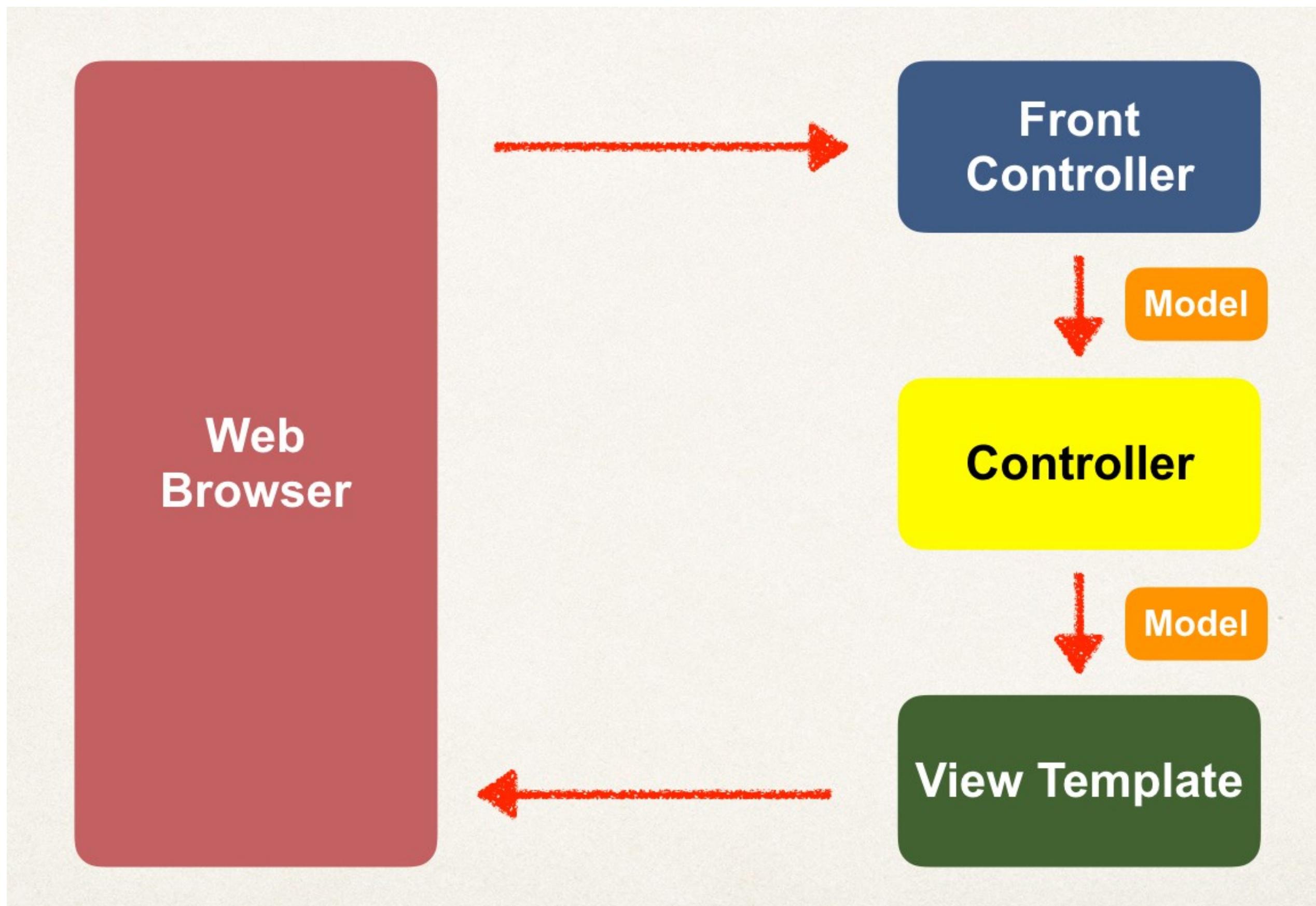
SPRING MVC

Componenti di un'app Spring MVC

- Un set di pagine web
- Una collezione di beans (controllers, services, etc...)
- Spring configuration (XML, Annotations or Java)

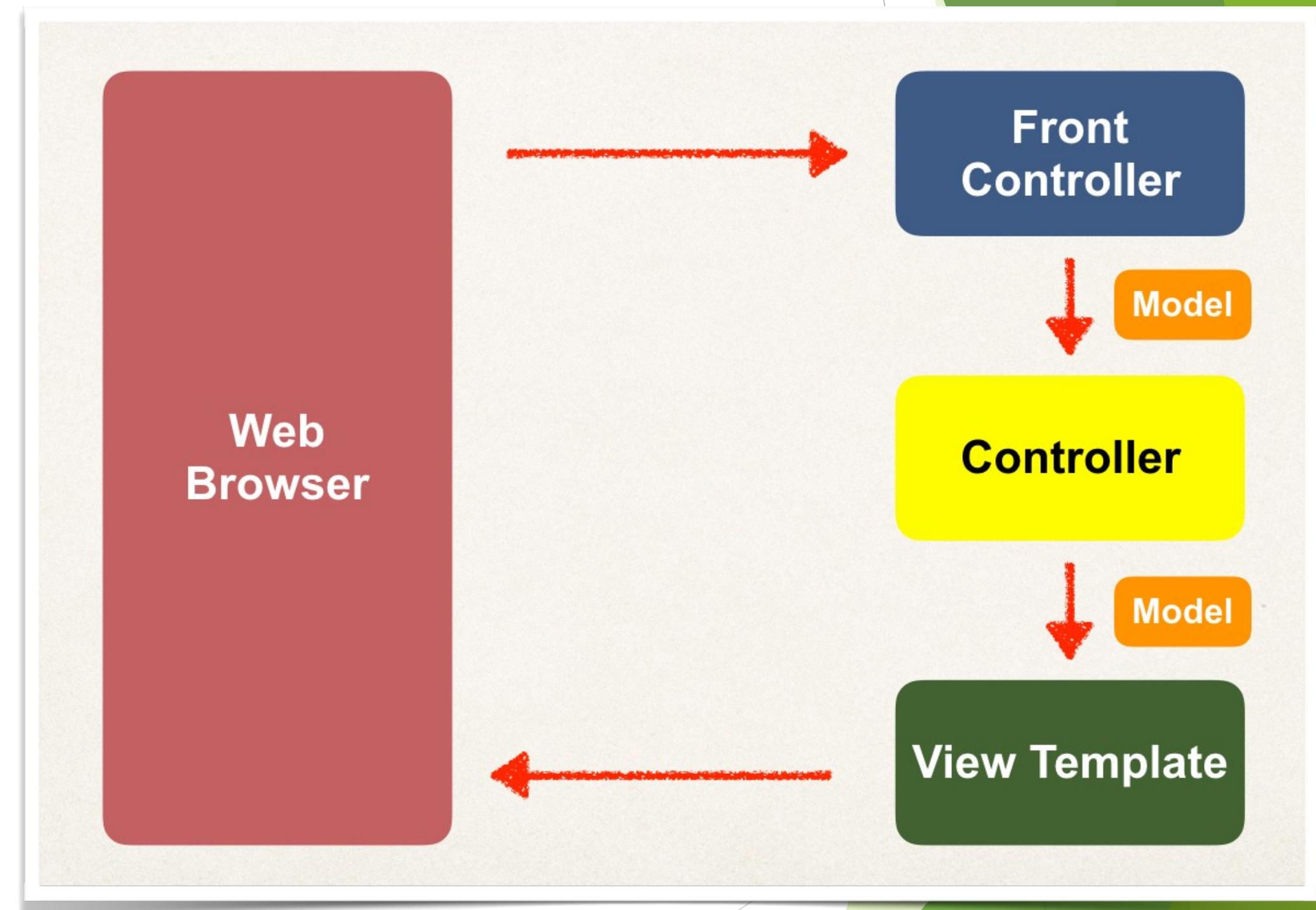


Come funziona Spring MVC?



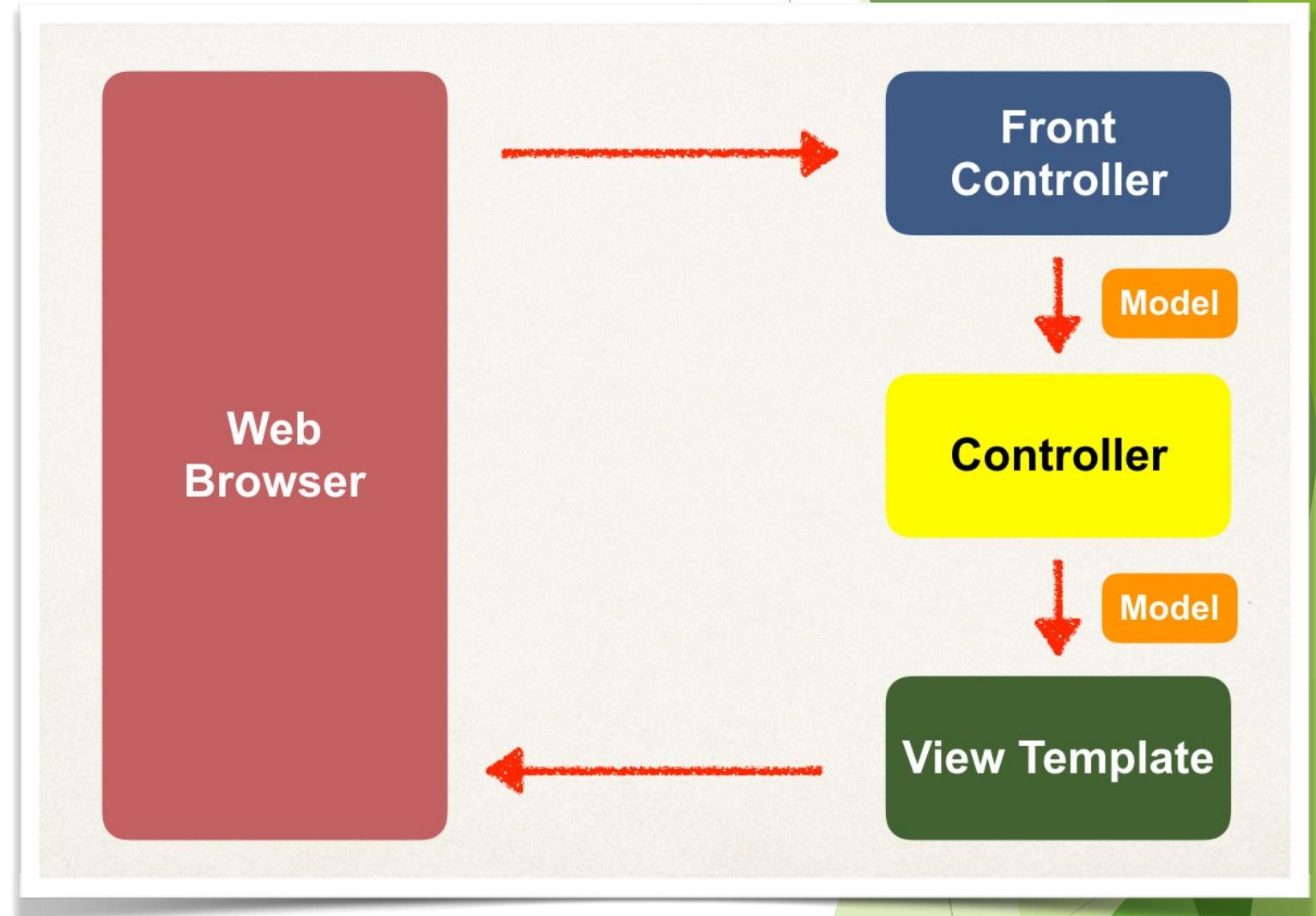
Spring MVC Front Controller

- Il front controller è conosciuto come **DispatcherServlet**
 - E' parte del framework
- Lo sviluppatore deve creare
 - Model
 - View templates
 - Controller



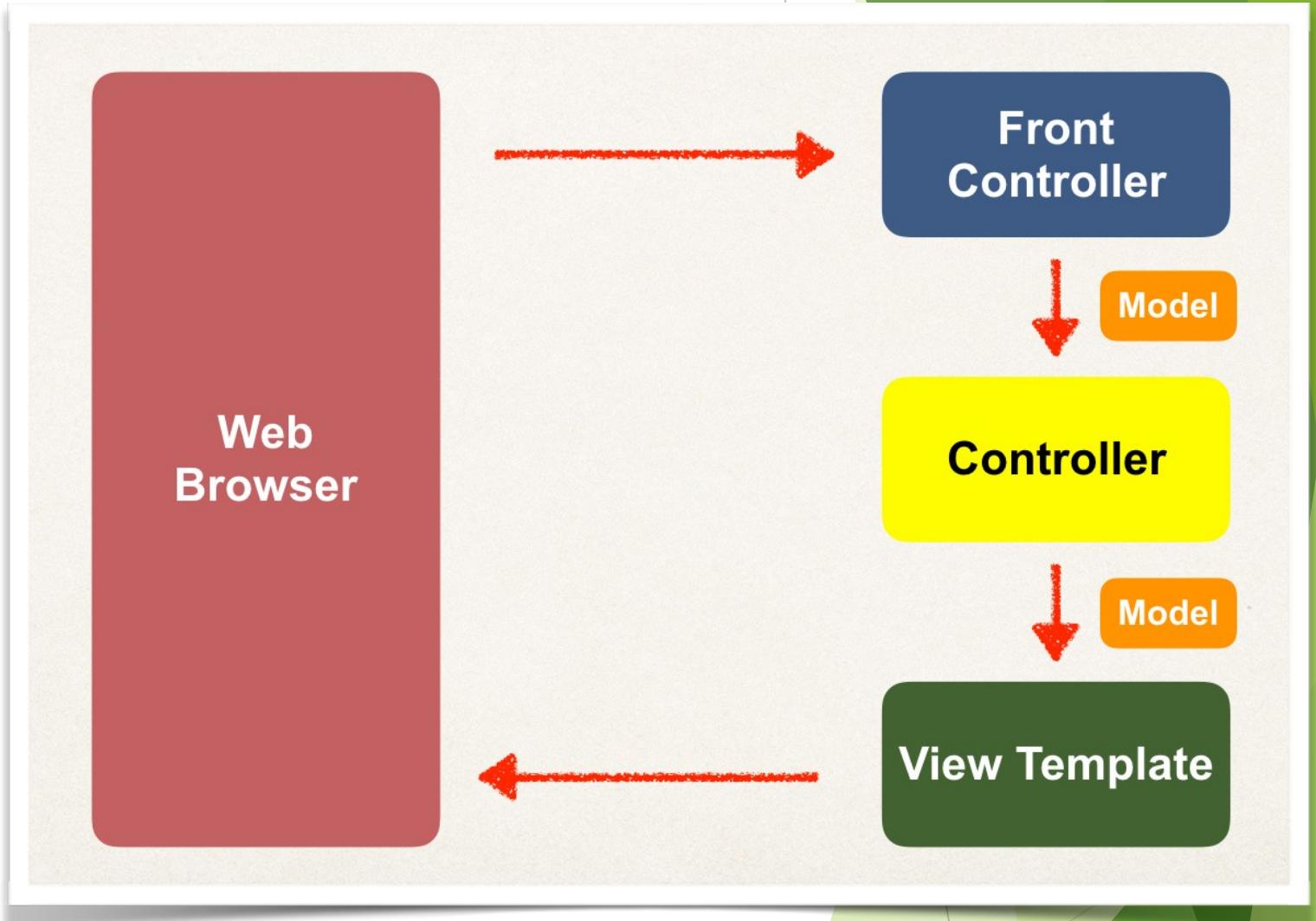
Controller

- Codice sviluppato dallo sviluppatore
- Contiene la logica di business
 - Gestisce le richieste
 - Gestisce i dati
 - Ingloba i dati nel model
- Invia i dati al motore di templating



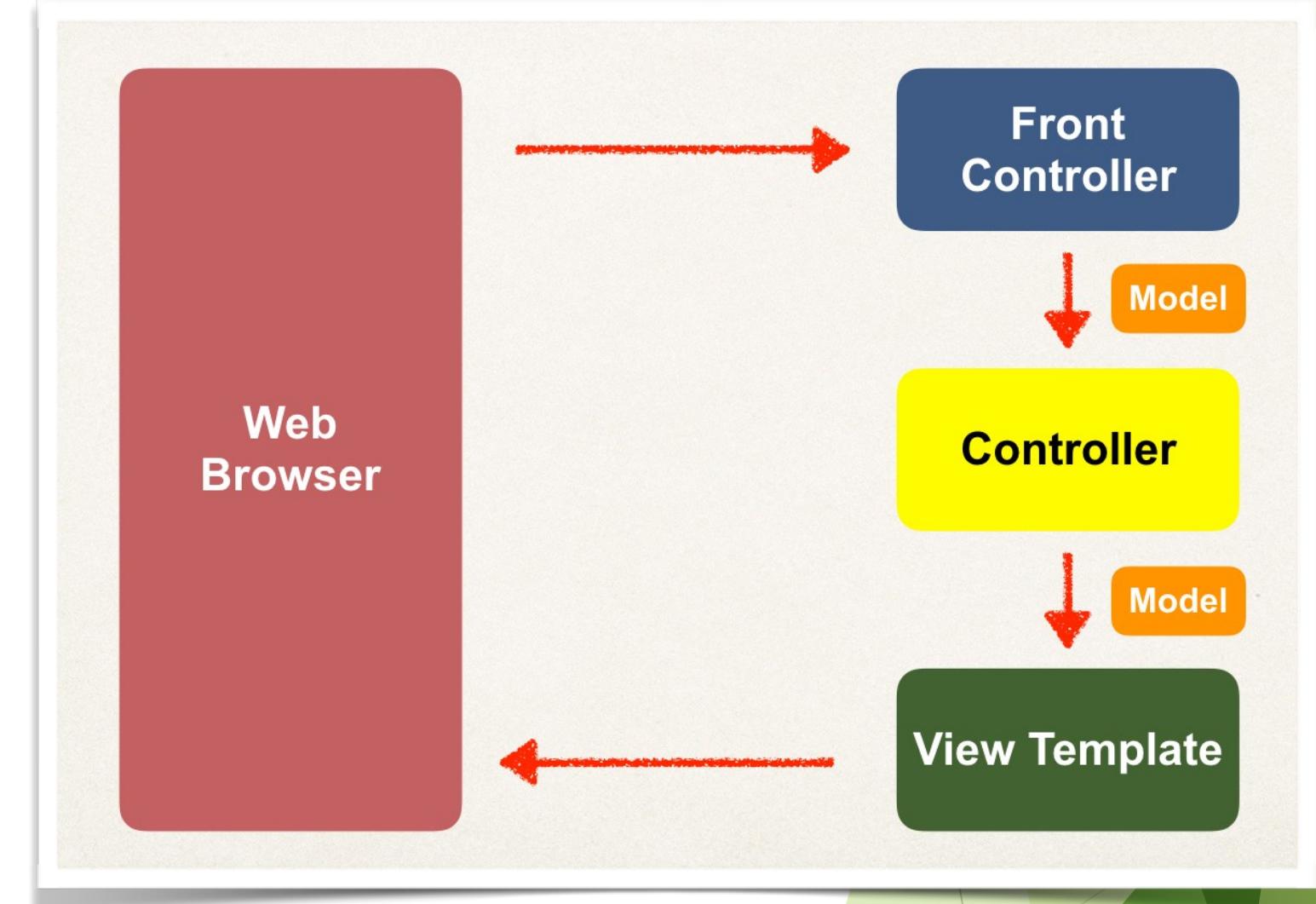
Model

- Model: contiene i dati
- Archivia/recupera i dati dal backend
 - database, web service, etc...



View template

- Spring MVC è flessibile
 - Supporta numerosi motori di template
- I più comuni sono **JSP + JSTL o Thymeleaf**
- Lo sviluppatore crea pagine web per mostrare i dati



JSP: Java Server Pages

JSTL: JSP Standard Tag Library

Configurazione Spring MVC – Parte 1

Aggiungere configurazioni in: **WEB-INF/web.xml**

1. Configurare il Dispatcher Servlet
2. Impostare il mapping degli URL verso il Dispatcher

Step-By-Step

Configurazione Spring MVC - Parte 2

Aggiungere configurazioni al file: **WEB-INF/spring-mvc.xml**

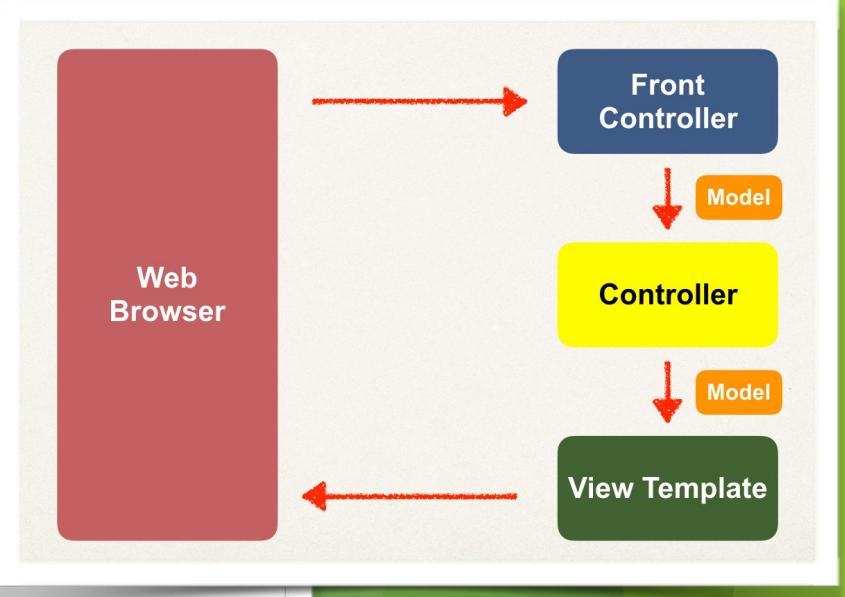
3. Aggiungere il supporto per la scansione dei componenti Spring
4. Aggiungere il supporto per la conversione, la formattazione e la validazione
5. Configurare Spring MVC View Resolver

Step-By-Step

Step 1: Configurare Spring DispatcherServlet

File: web.xml

```
<web-app>  
  
    <servlet>  
        <servlet-name>dispatcher</servlet-name>  
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
  
        <init-param>  
            <param-name>contextConfigLocation</param-name>  
            <param-value>/WEB-INF/spring-mvc.xml</param-value>  
        </init-param>  
  
        <load-on-startup>1</load-on-startup>  
    </servlet>  
  
</web-app>
```



Step 2: impostare il mapping degli URL verso Dispatcher

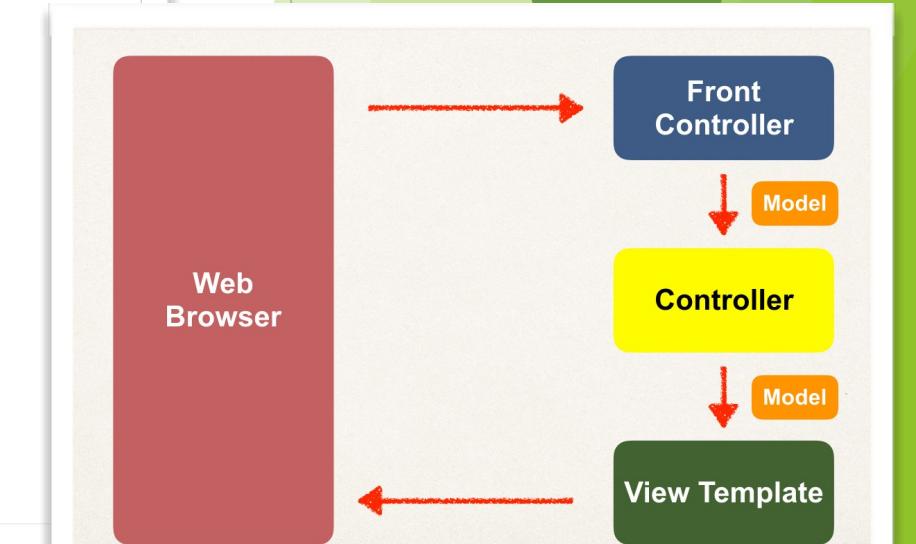
File: web.xml

```
<web-app>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        ...
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```



Step 3: aggiungere componentscanning

File: spring-mvc.xml

```
<beans>  
  
    <context:component-scan base-package="it.examples" />  
  
</beans>
```

Step 4: Aggiungere il supporto per la conversione, la formattazione e la validazione

File: spring-mvc.xml

```
<beans>

    <context:component-scan base-package="it.examples" />

    <mvc:annotation-driven/>

</beans>
```

Step 5: configurare il View Resolver

File: spring-mvc.xml

```
<beans>

    <context:component-scan base-package="it.examples" />

    <mvc:annotation-driven/>

    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/view/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

View Resolver Configs - Explained

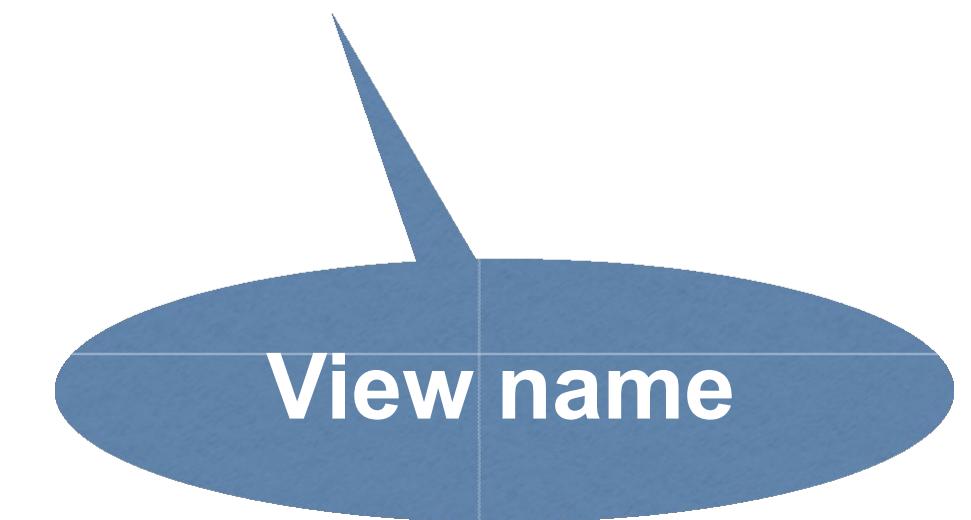
- Quando l'app restituisce il nome di una view, Spring MVC antepone il prefisso e appende il suffisso definiti in config.

```
<bean  
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"  
    <property name="prefix" value="/WEB-INF/view/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

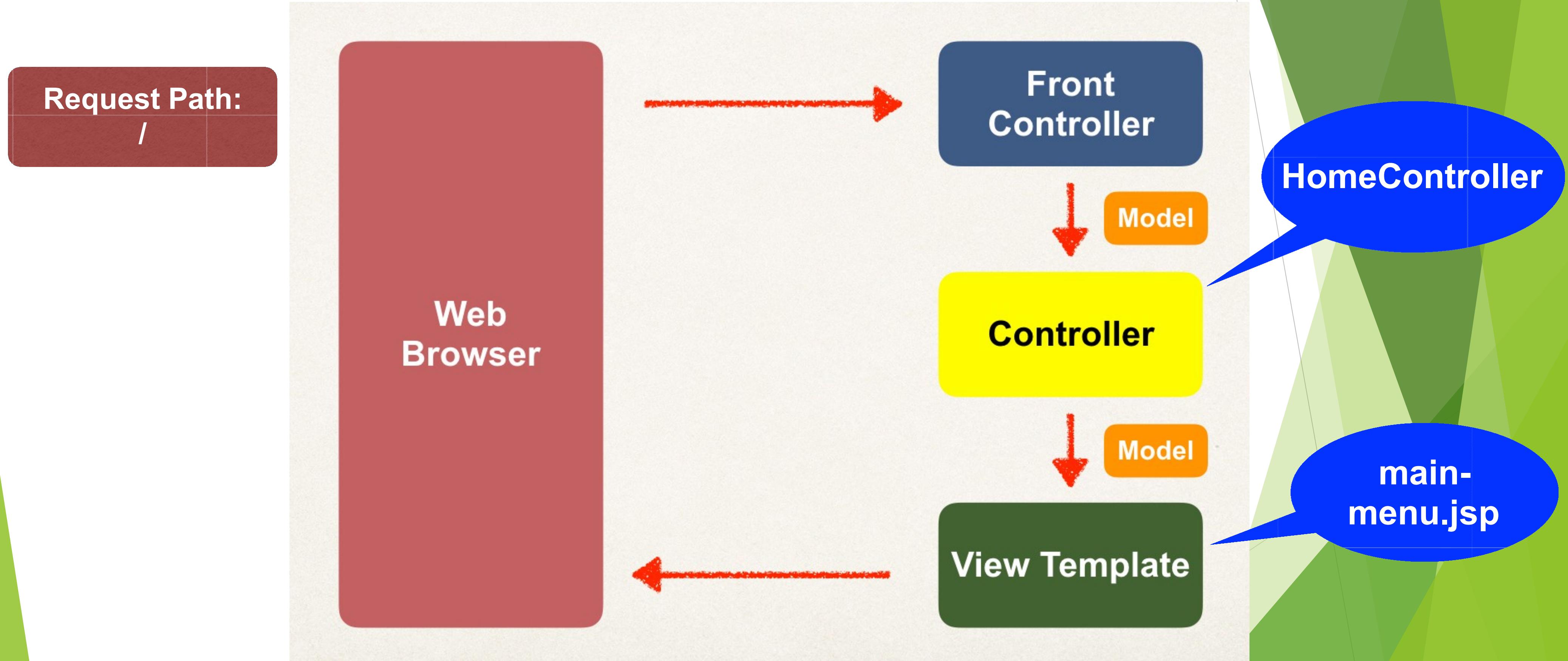
View Resolver

```
<bean  
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"  
    <property name="prefix" value="/WEB-INF/view/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

/WEB-INF/view/ show-student-list .jsp



Our First Spring MVC Example



Processo di sviluppo

Step-By-Step

1. Creare una classe Controller
2. Definire un metodo nella classe Controller
3. Aggiungere il RequestMapping al metodo
4. Restituire il nome della view
5. Sviluppare la view

Step 1: creare la classe controller

- Annotare la classe con @Controller
 - @Controller deriva da @Component

```
@Controller
public class HomeController {  
}
```

Step 2: aggiungere un metodo nel controller

```
@Controller  
public class HomeController {  
  
    public String showMyPage() {  
  
        ...  
    }  
  
}
```

Step 3: aggiungere RequestMapping

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/")  
    public String showMyPage() {  
  
        ...  
    }  
  
}
```

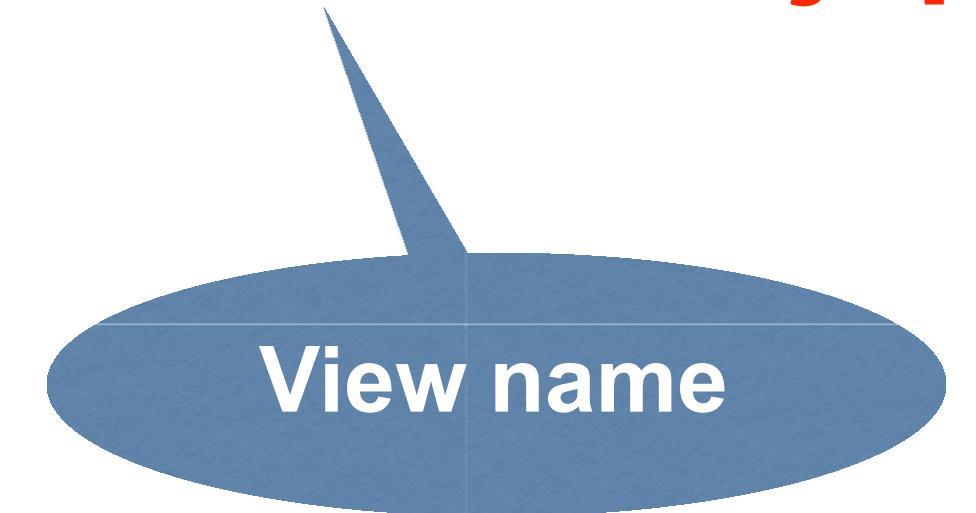
Step 4: restituire la view

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/")  
    public String showMyPage() {  
        return "main-menu";  
    }  
  
}
```

Individuazione della pagina di visualizzazione

```
<bean  
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"  
    <property name="prefix" value="/WEB-INF/view/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

/WEB-INF/view/main-menu.jsp



Step 5: sviluppare la vista

File: /WEB-INF/view/main-menu.jsp

```
<html><body>
```

```
<h2>Spring MVC Demo - Home Page</h2>
```

```
</body></html>
```

Leggere i dati da una form

helloworld-form.jsp



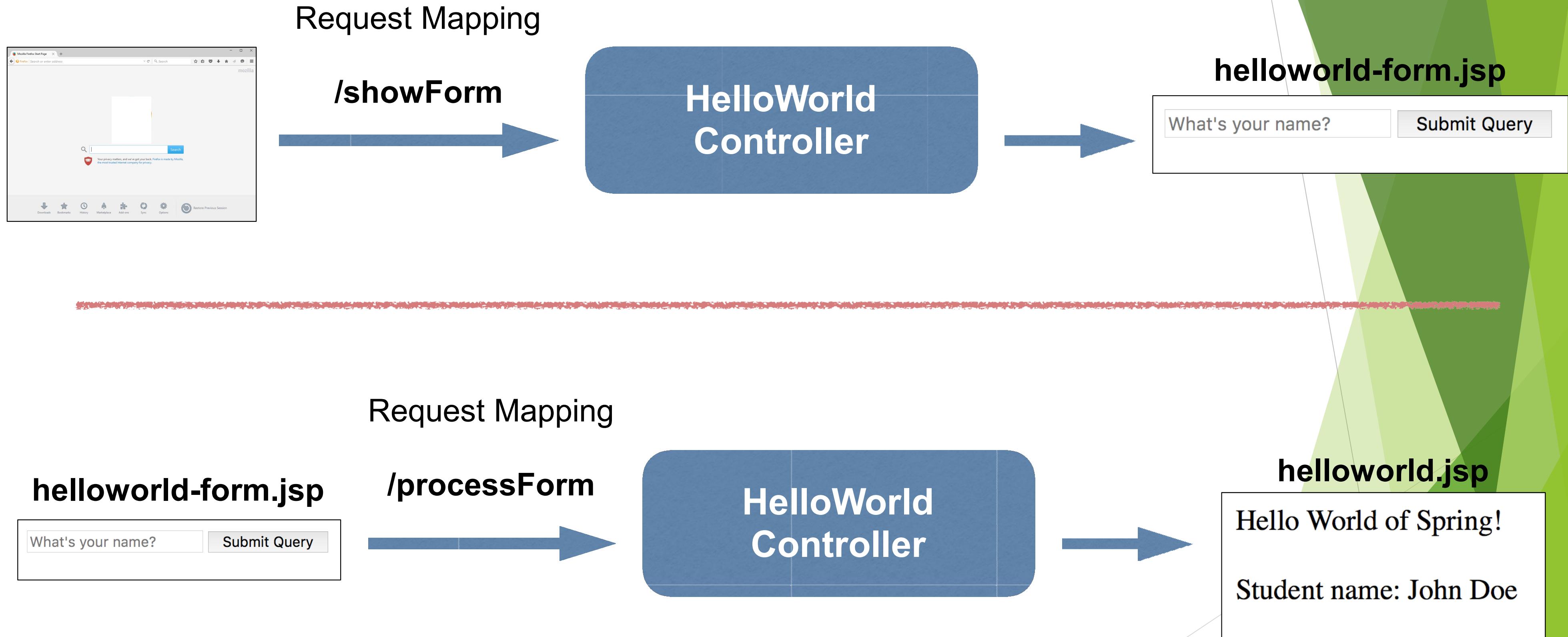
What's your name? Submit Query

helloworld.jsp

Hello World of Spring!

Student name: John Doe

Leggere i dati da una form: il flusso



Controller

```
@Controller  
public class HelloWorldController {  
  
    // need a controller method to show the initial HTML form  
  
    @RequestMapping("/showForm")  
    public String showForm() {  
        return "helloworld-form";  
    }  
  
    // need a controller method to process the HTML form  
  
    @RequestMapping("/processForm")  
    public String processForm() {  
        return "helloworld";  
    }  
}
```

Come si procede?

Step-By-Step

1. Creare il Controller

2. Mostrare il codice HTML della form

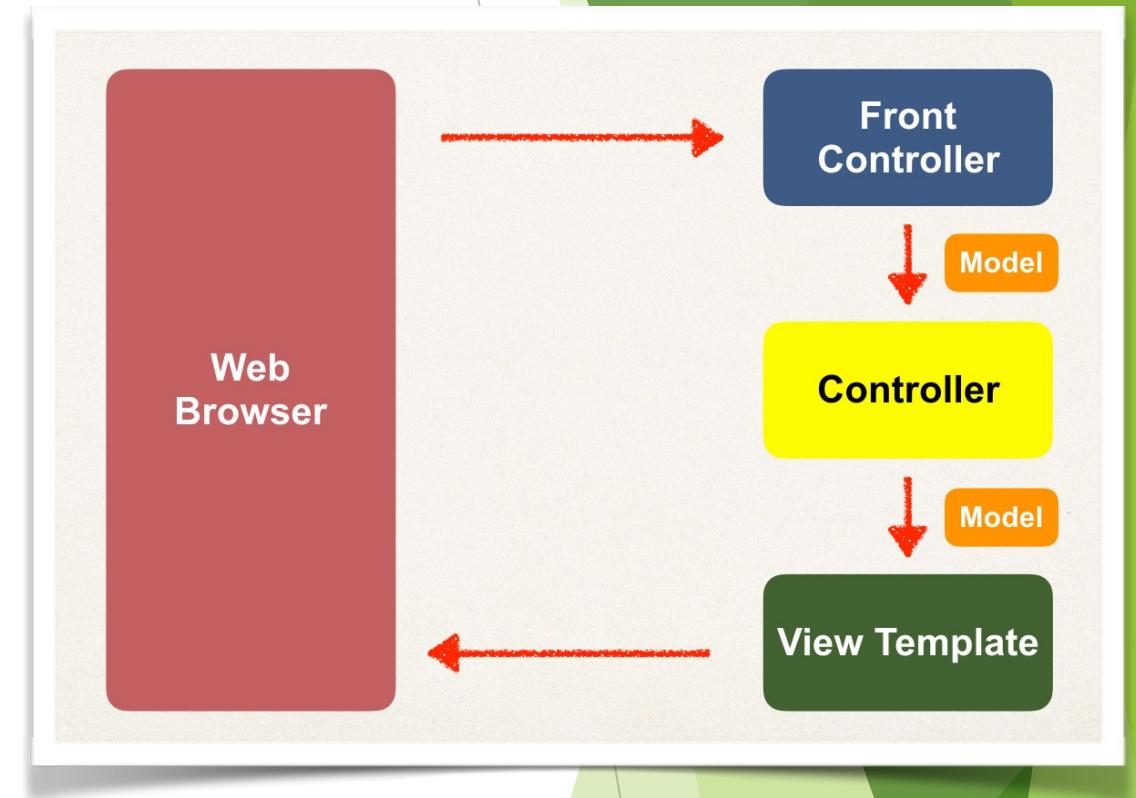
- a. Creare un metodo nel controller per mostrare la form
- b. Creare la view per visualizzare il modulo

3. Elaborare la form HTML

- a. Creare un metodo nel controller per processare la form
- b. Sviluppare una pagina per «confermare» l'elaborazione

Spring Model

- Il model è un container del dato
- Nel Controller
 - E' possibile immettere i dati nel **model**
 - strings, objects, info dal database, etc...
- Nelle view (ad esempio JSP) è possibile accedere ai dati nel model



Esempio di codice

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

    // read the request parameter from the HTML form
    String theName = request.getParameter("studentName");

    // convert the data to all caps
    theName = theName.toUpperCase();

    // create the message
    String result = "Yo! " + theName;

    // add message to the model
    model.addAttribute("message", result);

    return "helloworld";
}
```

View Template - JSP

```
<html><body>
```

Hello World of Spring!

...

The message: \${message}

```
</body></html>
```

Per aggiungere più dati al model...

```
// get the data
//
String result = ...
List<Student> theStudentList = ...
ShoppingCart theShoppingCart = ...

// add data to the model
//
model.addAttribute("message", result);

model.addAttribute("students", theStudentList);

model.addAttribute("shoppingCart", theShoppingCart);
```

Invece di usare HttpServletRequest...

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

    String theName = request.getParameter("studentName");

    ...
}
```

Eseguiamo il bind con @RequestParam

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(
    @RequestParam("studentName") String theName,
    Model model) {

    //
}
```

Controller Request Mapping... e «sub-mapping»

```
@RequestMapping("/funny")
public class FunnyController {

    @RequestMapping("/showForm")
    public String showForm() {
        ...
    }

    @RequestMapping("/processForm")
    public String process(HttpServletRequest request, Model model) {
        ...
    }
}
```

The diagram illustrates the mapping of URLs to controller methods. Two orange callout boxes point from specific annotations to their resulting URLs:

- The first callout points from the annotation `@RequestMapping("/showForm")` to the URL `/funny/showForm`.
- The second callout points from the annotation `@RequestMapping("/processForm")` to the URL `/funny/processForm`.

HTML Forms

- I form vengono usati per ricevere i dati dagli utenti

Sign In

Email Address:

Password:

Remember me

Sign In

Il form tag

- Il form tag genera HTML al posto dello sviluppatore

Form Tag	Description
form:form	main form container
form:input	text field
form:textarea	multi-line text field
form:checkbox	check box
form:radiobutton	radio buttons
form:select	drop down list
<i>more</i>	

Struttura pagina web

```
<html>
```

```
... regular html ...
```

```
... Spring MVC form tags ...
```

```
... more html ...
```

```
</html>
```

Come usare il form tag?

- Specificare il namespace all'inizio del JSP

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Come funziona il form tag?

student-form.jsp

First name:

Last name:

Student

**Student
Controller**

Student

student-confirmation.jsp

The student is confirmed: John Doe

Mostrare la form

Nel controller

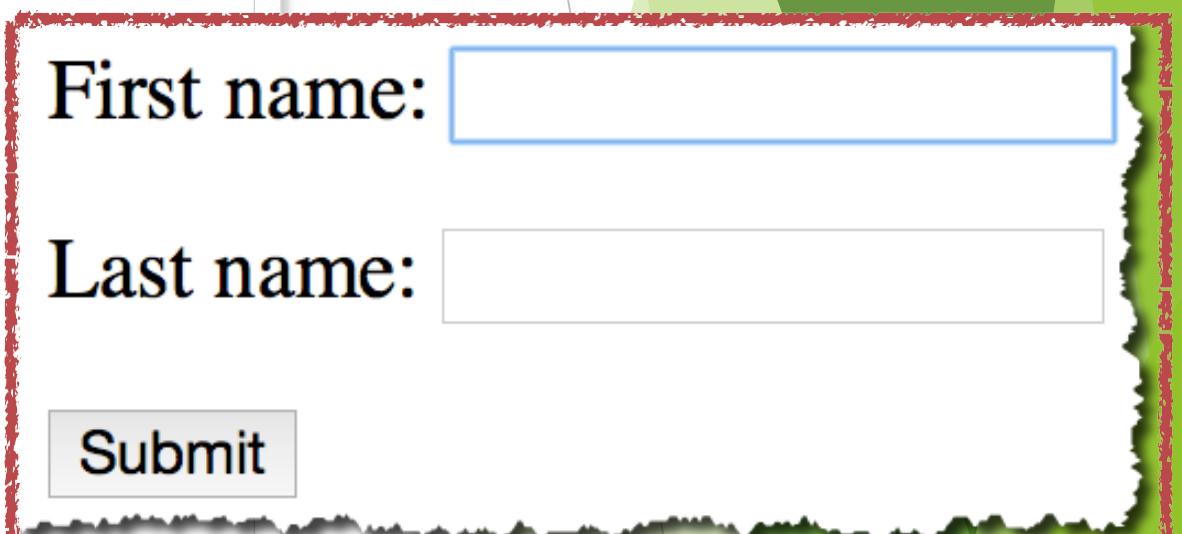
- Prima di mostrare la view, è necessario iniettare dei dati
- Si tratta in realtà di un oggetto che contiene una serie di dati da visualizzare all'interno della view e che permettono il data binding della form in fase di processamento

II ModelAttribute

```
@RequestMapping("/showForm")
public String showForm(Model theModel) {
    theModel.addAttribute("student", new Student());
    return "student-form";
}
```

Esempio di codice

```
<form:form action="processForm" modelAttribute="student">  
  
First name: <form:input path="firstName" />  
  
<br><br>  
  
Last name: <form:input path="lastName" />  
  
<br><br>  
  
<input type="submit" value="Submit" />  
  
</form:form>
```



The image shows a web page with a form. The form contains two text input fields, one for "First name" and one for "Last name", both outlined in blue. Below the inputs is a grey "Submit" button. The entire form is enclosed in a red border with a torn paper effect at the corners.

First name:

Last name:

Submit

Quando la form viene caricata...

```
<form:form action="processForm" modelAttribute="student">
```

First name: <form:input path="firstName" />

Last name: <form:input path="lastName" />

<input type="submit" value="Submit" />

```
</form:form>
```

Quando viene caricato il
form, Spring MVC
chiamerà:

student.getFirstName()

...

student.getLastName

Quando la form viene processata...

```
<form:form action="processForm" modelAttribute="student">
```

First name: <form:input path="firstName" />

Last name: <form:input path="lastName" />

<input type="submit" value="Submit" />

</form:form>

Quando il form viene inviato, Spring MVC chiamerà:

student.setFirstName(...)

...

student.setLastName(...)

Gestire la sumbit della form nel controller

```
@RequestMapping("/processForm")
public String processForm(@ModelAttribute("student") Student theStudent) {

    // log the input data
    System.out.println("theStudent: " + theStudent.getFirstName()
        + " " + theStudent.getLastName());

    return "student-confirmation";
}
```

Pagina di «conferma»

```
<html>
```

```
<body>
```

The student is confirmed: \${student.firstName} \${student.lastName}

```
</body>
```

```
</html>
```

The student is confirmed: John Doe

Development Process

Step-By-Step

1. Creare una classe Student
2. Creare un controller per Student
3. Creare HTML form
4. Creare il metodo per processare la form
5. Creare una pagina di «conferma»

HTML <select> Tag

First name:

Last name:

Country: Brazil

```
<select name="country">
  <option>Brazil</option>
  <option>France</option>
  <option>Germany</option>
  <option>India</option>
  ...
</select>
```

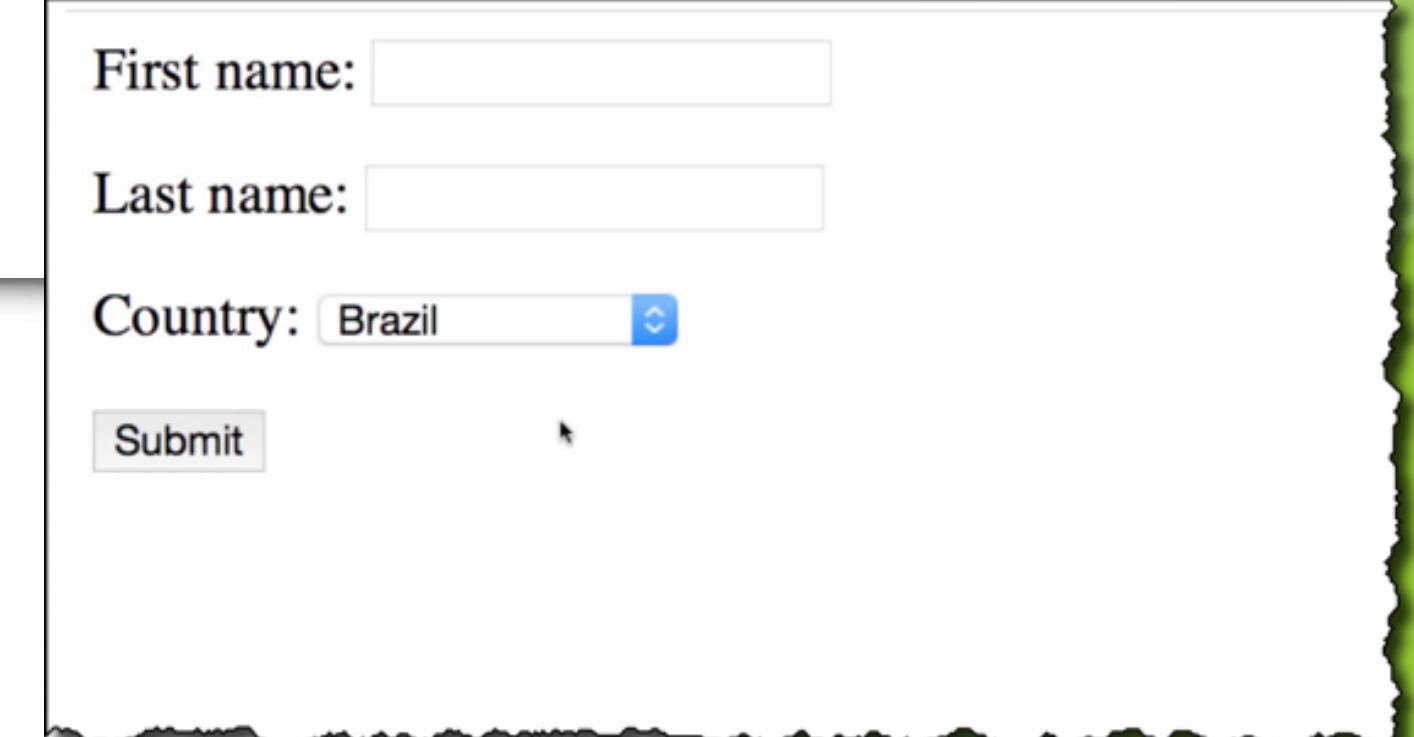
Spring MVC Tag

L'elenco a discesa è rappresentato dal tag

<form:select>

Esempio

```
<form:select path="country">  
  
    <form:option value="Brazil" label="Brazil" />  
    <form:option value="France" label="France" />  
    <form:option value="Germany" label="Germany" />  
    <form:option value="India" label="India" />  
  
</form:select>
```



First name:

Last name:

Country: ▼

Submit

Esempio radio buttons

First name:

Last name:

Favorite Programming Languages:

Java C# PHP Ruby



Spring MVC Tag

Un radio button è rappresentato dal tag

<form:radiobutton>

Code Example

```
Java <form:radio button path="favoriteLanguage" value="Java" />
C# <form:radio button path="favoriteLanguage" value="C#" />
PHP <form:radio button path="favoriteLanguage" value="PHP" />
Ruby <form:radio button path="favoriteLanguage" value="Ruby" />
```

Esempio checkbox

Operating Systems: Linux Mac OS MS Windows

Submit

Spring MVC Tag

Una checkbox è rappresentata dal tag

<form:checkbox>

Code Example

```
Linux <form:checkbox path="operatingSystems" value="Linux" />
Mac OS <form:checkbox path="operatingSystems" value="Mac OS" />
MS Windows <form:checkbox path="operatingSystems"
               value="MS Windows" />
```

Development Process

Step-By-Step

1. Aggiornare la form HTML
2. Aggiornare la classe Student
3. Aggiornare la pagina di conferma

Java Standard Bean Validation API

Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number \geq value
@Max	Must be a number \leq value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others ...	

Java Standard Bean Validation API

- E' solo una specifica, non presenta un'implementazione dei metodi di validazione
- Quindi abbiamo bisogno di integrare un'implementazione delle validation API che ci permetta di validare i campi degli oggetti: **Hibernate Validator**



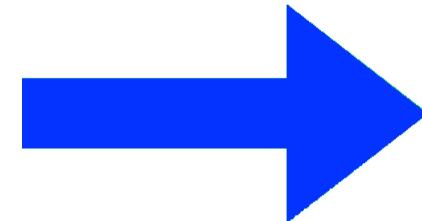
Campo richiesto

Fill out the form. Asterisk () means required.*

First name:

Last name (*):

Submit



Fill out the form. Asterisk () means required.*

First name:

Last name (*): **is required**

Submit

Come si procede?

Step-By-Step

1. Aggiungere la regola di validazione nella classe
2. Mostrare il messaggio di errore nel codice HTML
3. Eseguire la convalida da Controller

Step 1: aggiungere regola di validazione

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Customer {

    private String firstName;

    @NotNull(message="is required")
    @Size(min=1, message="is required")
    private String lastName;

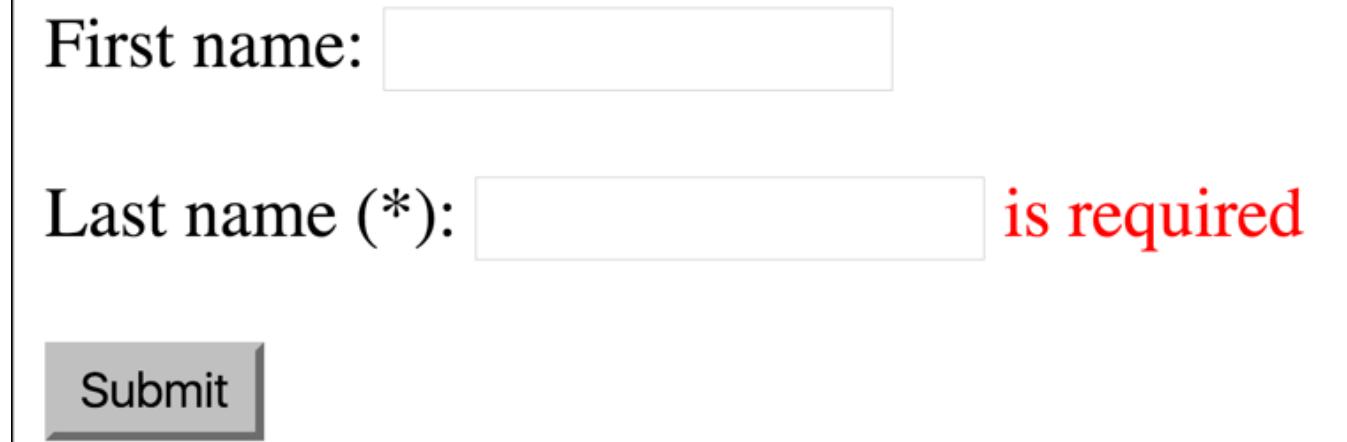
    // getter/setter methods

}
```

Step 2: mostrare errore in HTML

customer-form.jsp

```
<form:form action="processForm" modelAttribute="customer">  
  
First name: <form:input path="firstName" />  
  
<br><br>  
  
Last name (*): <form:input path="lastName" />  
<form:errors path="lastName" cssClass="error" />  
  
<br><br>  
  
<input type="submit" value="Submit" />  
  
</form:form>
```



The screenshot shows a web page with a form. The 'First name:' field is empty. The 'Last name (*):' field is also empty and has a red border, indicating it is required. A red error message 'is required' is displayed next to the field. A 'Submit' button is at the bottom of the form.

First name:

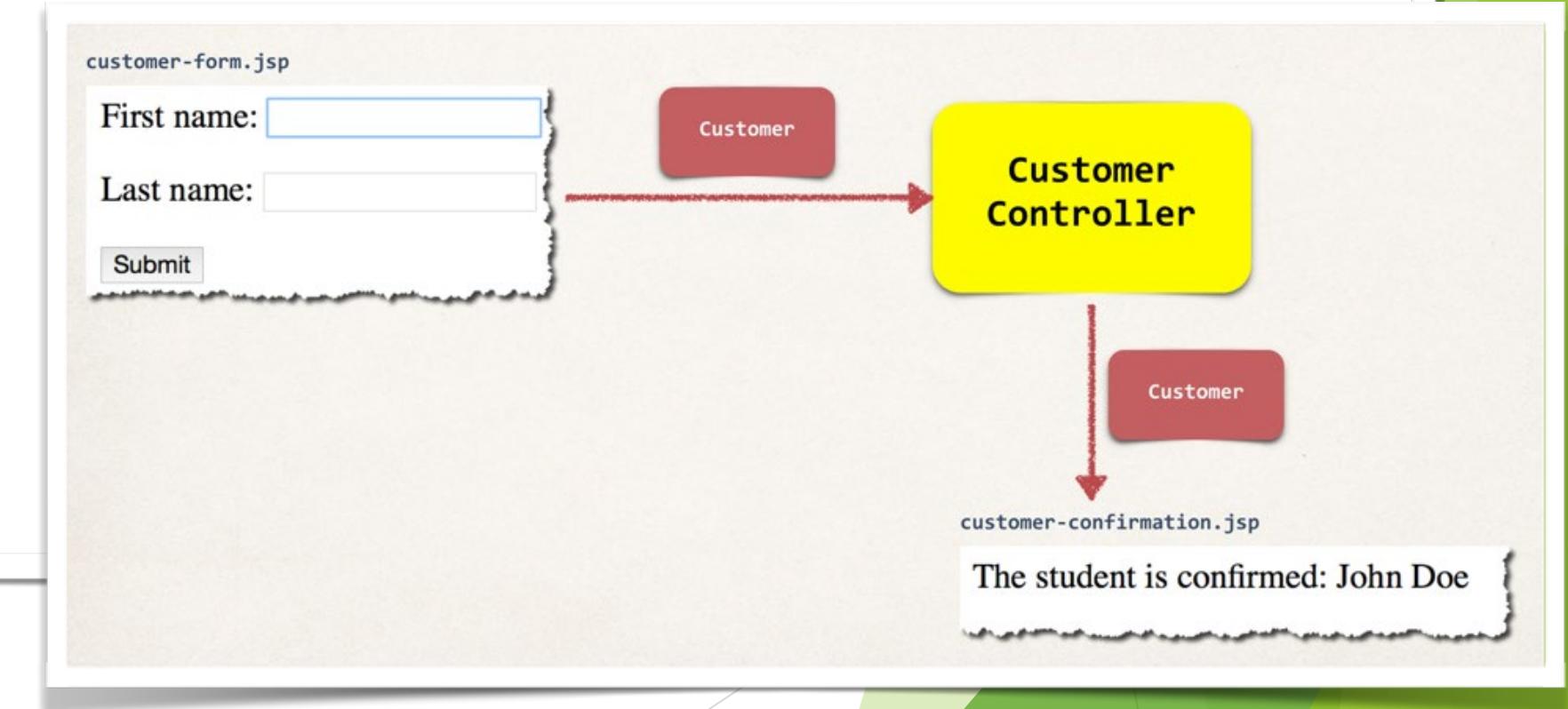
Last name (*): is required

Submit

Step 3: eseguire la convalida da controller

```
@RequestMapping("/processForm")
public String processForm(
    @Valid @ModelAttribute("customer") Customer theCustomer,
    BindingResult theBindingResult) {

    if (theBindingResult.hasErrors()) {
        return "customer-form";
    }
    else {
        return "customer-confirmation";
    }
}
```



Spazi

- Il nostro esempio precedente ha un problema con lo spazio. Se inseriamo un valore con tutti spazi bianchi, la validazione va a buon fine!
 - Ma dovrebbe fallire.
- Quindi dobbiamo eseguire un trim delle stringhe.

@InitBinder

- **@InitBinder** funziona come un pre-processore
- Pre-elabora ogni richiesta web che arriva al controller in cui è definito

@InitBinder

CustomerController.java

...

```
@InitBinder
public void initBinder(WebDataBinder dataBinder) {

    StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

    dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);

}
```

...

@Min e @Max

```
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;

public class Customer {

    @Min(value=0, message="must be greater than or equal to zero")
    @Max(value=10, message="must be less than or equal to 10")
    private int freePasses;

    // getter/setter methods

}
```

@Pattern

```
import javax.validation.constraints.Pattern;  
  
public class Customer {  
  
    @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")  
    private String postalCode;  
  
    // getter/setter methods  
  
}
```