Spring, Spring Boot e i servizi REST

Introduzione alle applicazioni web

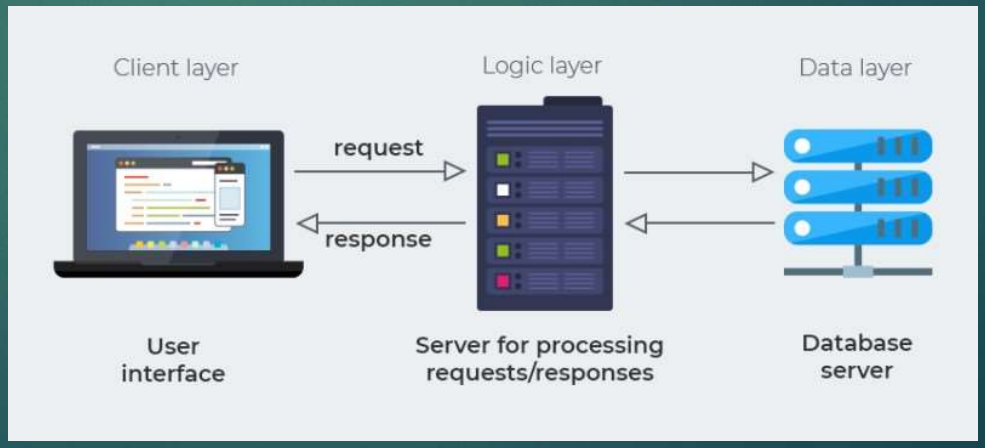
Le applicazioni distribuite

- Le applicazioni distribuite sono applicazioni in cui i vari componenti che compongono il sistema si trovano su macchine distinte.
  - I vari processi cooperano scambiandosi dati e messaggi
- Un'applicazione distribuita è strutturata solitamente con un'architettura multi layer
  - Ogni layer è costituito da un set di componenti
  - Ogni set di componenti realizza un set di funzionalità

Architettura multi layer

- L'architettura multilivello più diffusa è l'architettura three-tier.
- Essa prevede:

1. Client Layer
2. Logic Layer
3. Data Layer



Client tier → livello presentazione

- Scopo principale: visualizzare le informazioni e raccogliere dati dall'utente → interazione con l'utente
- Questo tier di **livello superiore**, ad esempio, può essere eseguito su un browser web, come applicazione desktop o come GUI (finestra grafica).
- Generalmente, i tier presentazione web sono sviluppati utilizzando HTML, CSS e JavaScript.

Logic tier → livello applicativo

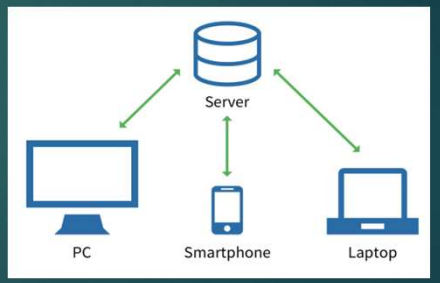
- Scopo principale: recuperare informazioni dal client tier, elaborarle e metterle in relazione con il data tier.
- Questo tier di **livello intermedio** contiene la **logica di business**, un insieme specifico di regole e comportamenti che il sistema fornisce
  - Il tier applicativo può anche aggiungere, eliminare o modificare i dati nel data tier .
- Il tier applicativo è tipicamente sviluppato utilizzando Python, Java, Perl, PHP o Ruby e comunica con il data tier tramite librerie(API)

Data tier → livello persistenza

- Scopo principale: archiviare e gestire i dati persistenti dell'applicazione, solitamente attraverso un database
- Questo tier è il **livello più basso** e interagisce solo con il livello applicativo.
- Questo livello si realizza con **database relazionali** come PostgreSQL, MySQL, MariaDB, Oracle, DB2, Informix o Microsoft SQL Server o **database NoSQL** come Cassandra, CouchDB o MongoDB.

Applicazione client - server

- Un sistema client-server è un contesto **applicativo di rete** dove un programma detto CLIENT richiede servizi ad un altro programma detto SERVER
- Il programma cliente e il programma server girano tipicamente su macchine distinte collegate in rete.
  - Il Client potrebbe usare differenti dispositivi per collegarsi al Server



- Regole generali:

1. Il server eroga servizi (su richiesta) per i client
2. I client si collegano ai server e richiedono servizi
3. Client e server devono aver concordato un protocollo di comunicazione e scambio dati

Le applicazioni web

- Le applicazioni web sono particolari **applicazioni distribuite** che usano l'architettura **client-server**

- Caratteristiche
  - Utilizzano il protocollo TCP/IP
  - Si dividono in 2 famiglie principali:
    - Applicazioni a pagine → il client è l'utente che, attraverso il browser, chiede un servizio al server e riceve in risposta una **pagina web**
    - Applicazioni a servizi → il client è un programma che chiede un servizio al server e riceve in risposta **dati**

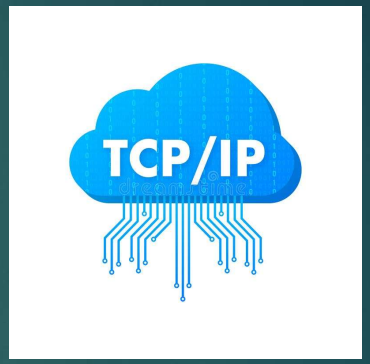
Caratteristiche di un'applicazione a servizi

1. Il client è un programma (come il server)
   - Applicazione Android
   - Applicazione IOS
   - Browser che esegue codice Javascript in una pagina web
2. Client e server si scambiano SOLO dati
3. Il sistema client-server è interoperabile
   - Client e Server possono essere scritti con infrastruttura tecnologica e linguaggi indipendenti
4. La realizzazione segue uno di questi 2 stili principali:
   - SOAP (stile classico)
   - REST (nuovo stile)

Il protocollo HTTP

Protocollo TCP/IP

- Un protocollo è un insieme di regole per dialogare.
- Il protocollo TCP/IP è il protocollo utilizzato sul web
- Le figure base sono Client e Server
- Principio base: Il client invoca il server (mai il contrario)



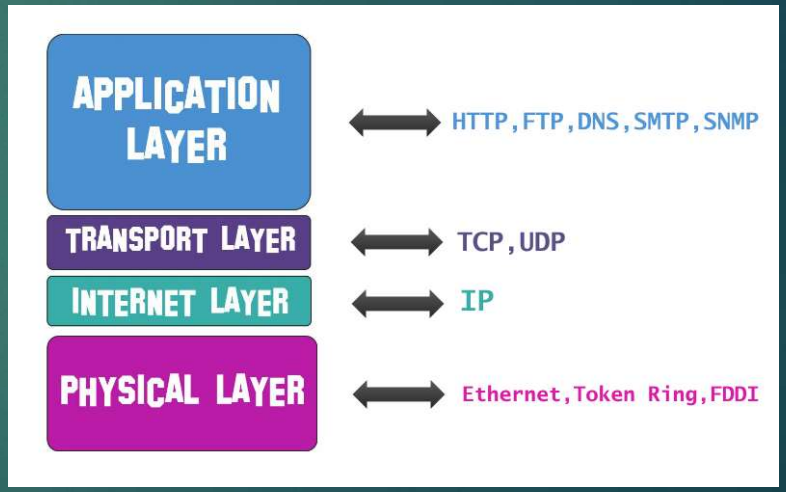
- Il protocollo TCP/IP è una famiglia di protocolli composta da 4 livelli
  - Ogni livello si occupa di un aspetto e prevede un set di regole
- I livelli sono: applicazione, trasporto, rete, fisico

Livelli del protocollo

- Il dialogo avviene correttamente solo se per ciascuno dei 4 livelli viene utilizzato lo stesso sotto protocollo.

- Livello fisico → protocollo nativo
- Livello rete → IP
- Livello trasporto → TCP e UDP
- Livello applicazione →

  vari protocolli specifici per varie esigenze
  - HTTP → hyper text
  - FTP → file di grandi dimensioni
  - SMTP → mail
  - ecc



Livello application

- La realizzazione di un'applicazione web coinvolge solamente il livello più alto del protocollo (application)

- Per le applicazioni a pagine è previsto il **protocollo HTTP**
- Per le applicazioni a servizi non esisteva un protocollo specifico e si è deciso di utilizzare ancora **HTTP**

Protocollo HTTP: descrizione

- Il server offre servizi ed è in attesa del client
- Il client chiede un servizio indicando:
  - URL → protocollo, ip, porta e path verso la risorsa da agganciare

    `http://IP:port/path/resource`

  - VERB → sono parole codificate dal protocollo che servono a indicare il tipo di azione che si vuole compiere sulla risorsa indicata dalla URL

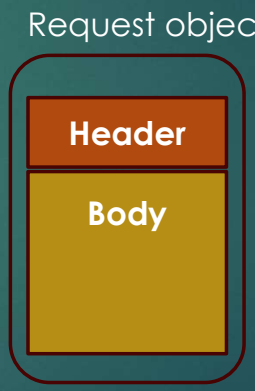
    In ambito REST si considerano solo le seguenti:

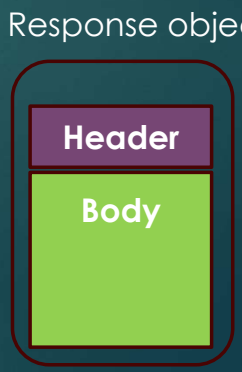
    - **GET** → lettura
    - **POST** → inserimento
    - **PUT** → modifica totale
    - **PATCH** → modifica parziale
    - **DELETE** → cancellazione

Trasporto dei dati

- Request e Response hanno un header e un body
  - Le informazioni dell'header sono meta dati e sono strutturate
  - Il body invece non segue regole (in passato si usava XML, in ambito REST si usa il formato **JSON**)

- Il client ha 3 modi di inviare dati al server:
  - Utilizzando la query string che si accoda alla URL
    - `http://ip:port/path/resource?var1=aaa&var2=bbb`
  - Inserendoli direttamente nella URL
    - `http://ip:port/path/resource/aaa/bbb`
  - Inserendoli nel body della richiesta
- Il server invece ha un solo modo di inviare dati:
  - Inserendoli nel body della risposta

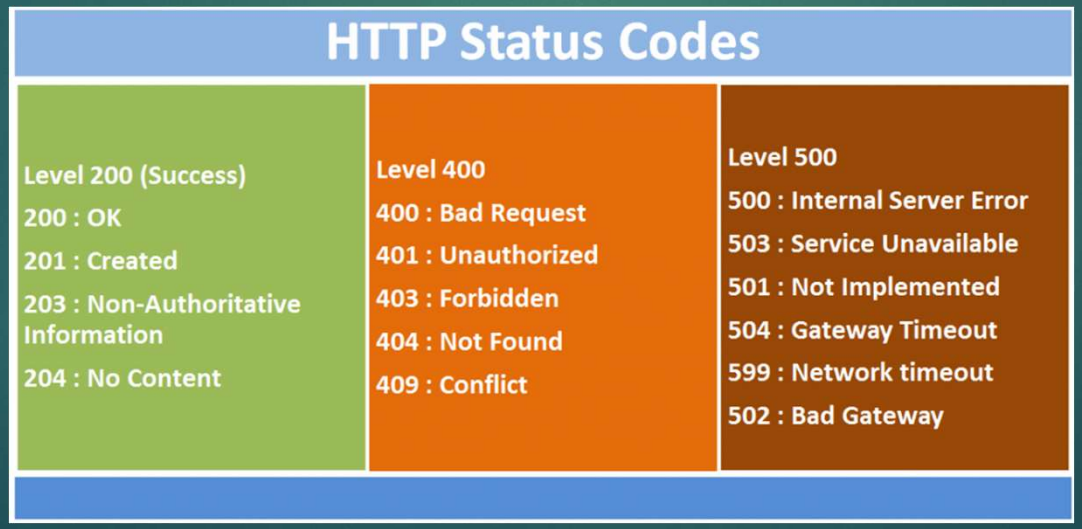




Il formato JSON si utilizza per l'invio di dati strutturati (tipo oggetti)

Status code

- Il server è tenuto a restituire uno status code per ogni richiesta col quale indicherà l'esisto dell'operazione.
- I codici vanno per gruppi di 100



Il formato json 1/2

- Il formato JSON è simile all'XML, ma molto più leggero

Questo è un oggetto rappresentato in formato JSON

Le proprietà sono racchiuse da {…} e separate da virgole.

Sono indicate con

nome (stringa) **:** **valore** (in base al tipo di variabile)

Il valore può essere numerico, testo oppure un oggetto strutturato in json

```
{
    "idUtente": 444,
    "nome": "nome4",
    "cognome": "cognome4",
    "mail": "mail4@xx.it",
    "telefono": "444.444"
}
```

- Sia client che server possono inviare dati in formato JSON e devono specificarlo nel content-type dell'header (della richiesta/risposta) in questo modo:

```
content-type='application/JSON'
```

Il formato json 2/2

- Se il client esegue una richiesta per la quale si aspetta dati in formato JSON dovrà dichiarare nell'header della richiesta:

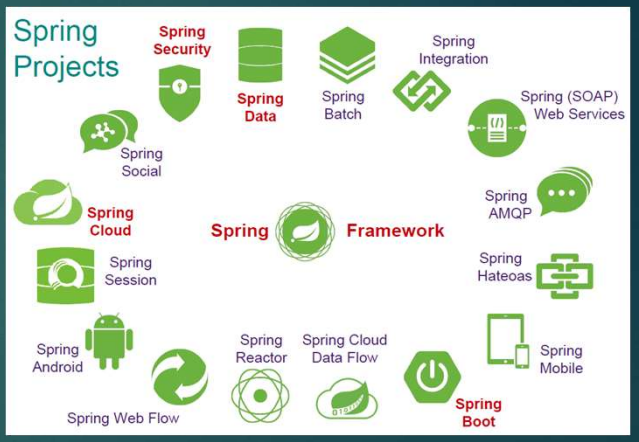
```
accept = 'application/JSON'
```

- Di conseguenza il server dovrà rispondere inviando dati in questo formato.
  - Se il client indica diverse opzioni di formato e il server le supporta, allora può sceglierne una tra quelle indicate

Spring framework

Spring

- Spring è uno dei framework più famosi utilizzati in java.
- E' composto da una serie di progetti che aiutano nella gestione delle principali problematiche nello sviluppo di applicazioni:
  - **Spring Web Flow** per applicazioni web a pagine o a servizi
  - **Spring Data** per la gestione della persistenza
  - **Spring Mobile**, estensione di spring web per applicazioni mobile
  - **Spring Integration** per l'integrazione di applicazioni aziendali
  - **Spring Security** per gestire autenticazioni e autorizzazioni



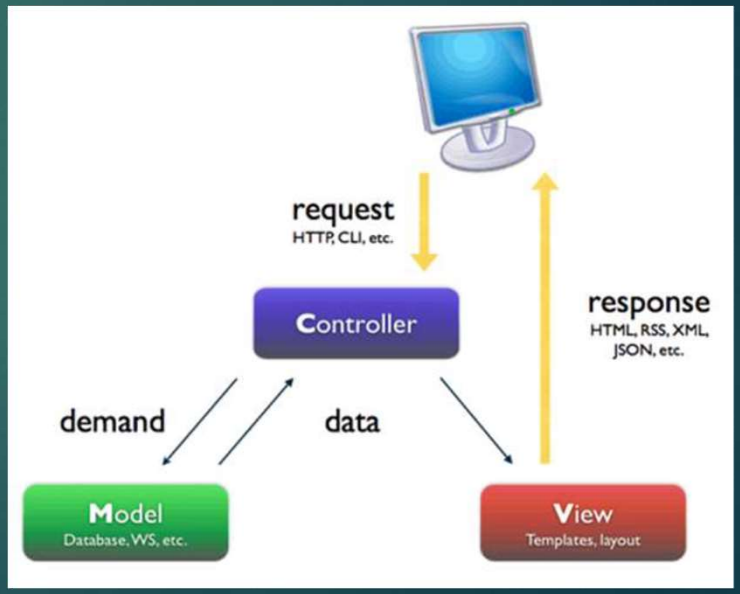
Spring web

- Spring web è il modulo di Spring che offre il supporto alla realizzazione di applicazioni web con architettura MVC

- Il pattern MVC è un pattern architetturale

che propone di scomporre il sistema in 3 livelli:

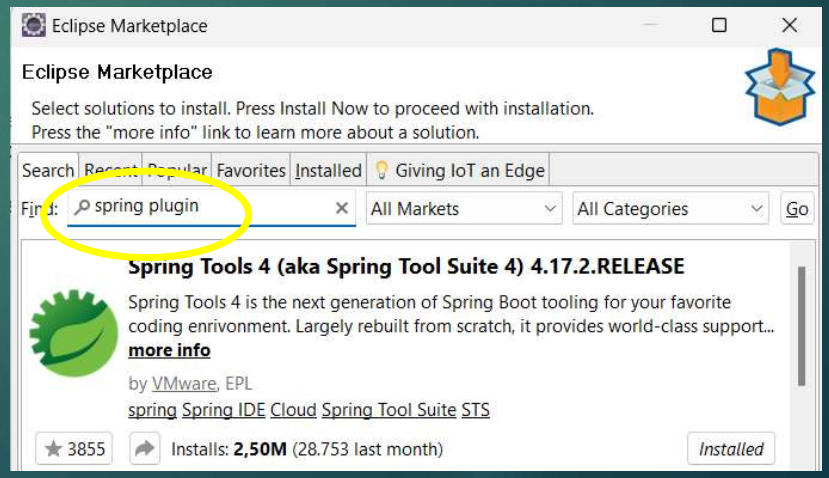
- Model → logica business
- View → strato presentazione
- Controller → coordinamento Model/View



Spring Boot

- Per utilizzare Spring web è necessario scaricare le librerie relative a questo modulo
- Le librerie si possono ottenere in questi modi:
  - scaricarle a mano → scelta sconsigliata, bisogna anche scaricare tutte le sottodipendenze
  - creare un progetto **Maven** → scelta consigliata, perché gestisce tutte le sottodipendenze
  - creare un progetto **Spring Boot** → scelta ottimale, perché configura Maven e lo attiva

- Spring Boot è un tool che aiuta nella configurazione di un progetto spring.
- Se si lavora con Eclipse è possibile scaricare il plugin per la creazione di progetti **Spring Boot**.



Applicazioni a servizi con Spring

Applicazione a servizi con Spring

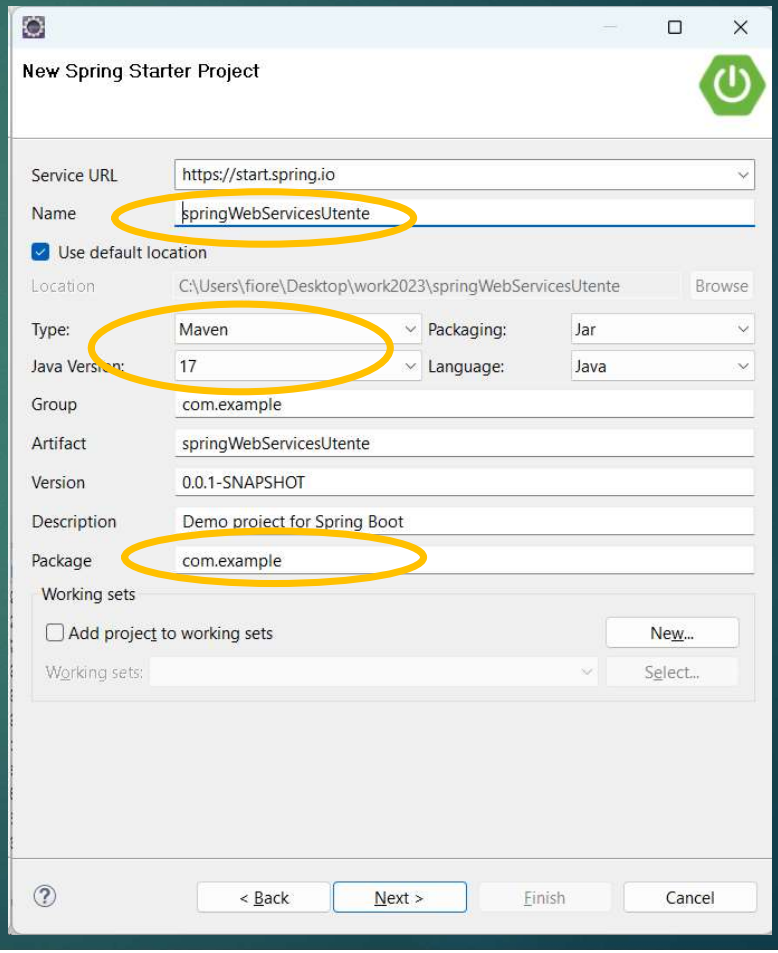
- Per realizzare un'applicazione a servizi è utile il supporto del modulo Spring WEB del framework Spring
- Normalmente ci si avvale anche di Spring Boot (il tool che semplifica e gestisce tutti gli aspetti configurativi)

- Sebbene un'applicazione a servizi sia un'applicazione web NON devo creare un progetto Java EE, ma un Java Project (Java SE)

- Se si dispone del plug-in di Eclipse per Spring, si può partire creando uno **Spring Starter Project**
  - Viceversa si può usare un wizard via web al link: https://start.spring.io/

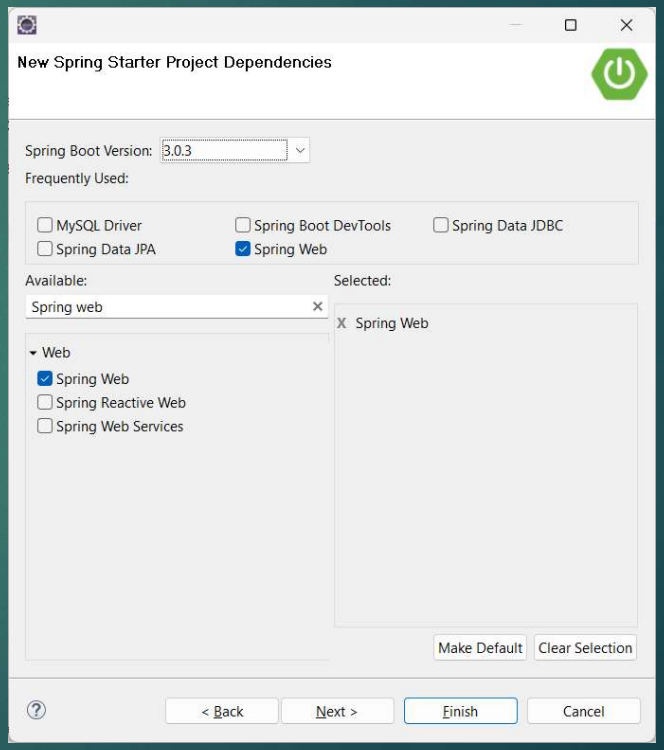
Creazione del progetto

- Imposto
  - Name = **springWebServicesUtente**
  - Type = Maven
  - Java version = 17
  - Package = **com.example**
- NEXT

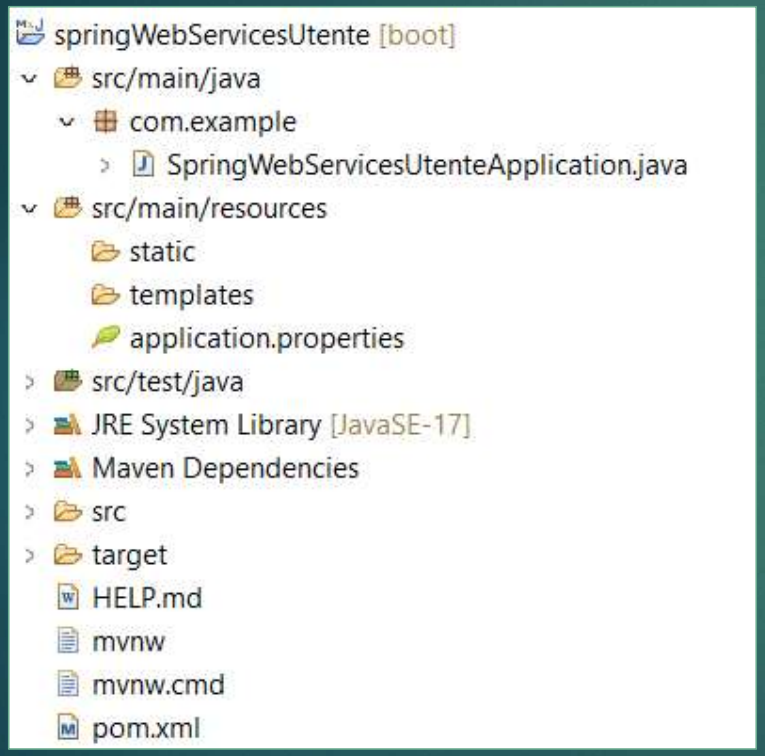


Creazione del progetto (2)

- Carico le dipendenze.
- Cerco Spring Web e lo seleziono
- FINISH



Struttura del progetto



- In src/main/java abbiamo la classe starter: SpringWebServicesUtenteApplication
- In src/main/resources abbiamo il file di configurazione: application.properties
- Infine sotto la root il file pom.xml per la gestione delle librerie

- NOTA: La directory src/main/webapp tipica della web application NON è presente (perché non produciamo pagine ma dati)

Configurare il pom.xml

- Creando il progetto vengono già caricate le dipendenze per
  - `spring-boot-starter-web`
  - `spring-boot-starter-test`

Per default potremmo produrre dati in **formato JSON, se si volesse supportare anche il formato XML si deve aggiungere anche questa dipendenza:**

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Configurare il properties

- La ServletController di Spring non è visibile nel progetto, ma bisogna impostare la sua URL, che va inserita nel file di configurazione di Spring → **application.properties**

```
# configuro l'url della servlet controller di Spring
spring.mvc.servlet.path=/spring-utenti
```

NON servono altre configurazioni!

Una classe Controller

Queste configurazioni non tengono ancora in considerazione lo stile REST

- La classi Controller sono semplici classi java e non Servlet
- Vanno annotate con `@RestController` in modo che la ServletController di Spring le possa riconoscere e caricare (le controller verranno istanziate allo startup del server)
- Devono anche specificare un path di invocazione con l'annotation

```
@RequestMapping(path = "")
```

```
@RestController
@RequestMapping(path = "/gestioneutenti")
public class UtenteController {
        // metodi della controller
}
```

Notare che la classe non ha dipendenze esplicite da Spring se non fosse per le annotation

Metodi della Controller

- Le Controller possono avere diversi metodi (non seguono particolari interfacce)
- Ogni metodo della Controller userà una annotation per specificare il VERBO
  - **@GetMapping**, **@PostMapping**, **@PutMapping**, **@PatchMapping**, **@DeleteMapping**
- La proprietà **`path`** definisce la URL che inizia con / (slash) e può anche essere parametrica utilizzando **`{nomeVar}`**
- Se il metodo produce e/o consuma dati lo si indica con **`produces`** e/o con **`consumes`**

```
@GetMapping(path="",   produces ="", consumes ="")
```

segue esempio →

- L'URL composto dal **path della Controller + il path del metodo** serve a Spring per identificare univocamente l'oggetto Controller sul quale agire e il metodo da invocare

Metodi della Controller

Queste configurazioni non tengono ancora in considerazione lo stile REST

```
@GetMapping(path="/cerca/{idUtente}", produces = "application/json")
public Utente cercaUtente(@PathVariable Integer idUtente) {
   DAOUtente d = new DAOUtente();
   return d.selectById(idUtente);
}
```

legge

```
@PostMapping(path="/registra", consumes = "application/json")
public void registra(@RequestBody Utente utente) {
    DAOUtente d = new DAOUtente();
    d.insert(utente);
}
```

scrive

- Con **`{idUtente}`** si intende che la URL contiene una variabile che coincide con l'argomento del metodo (che andrà quindi annotato con **`@PathVariable`**)
- Con **`produces`** stiamo indicando che il ritorno del metodo sarà un Utente strutturato in formato application/json
- Con **`consumes`** stiamo indicando che il metodo si aspetta un oggetto in formato json. L'oggetto sarà quello annotato con **`@RequestBody`**

Architettura completa

Scomporre il Model

- Il pattern MVC afferma che i componenti controller dovrebbero fare il coordinamento dei componenti model e NON contenere logica di business.

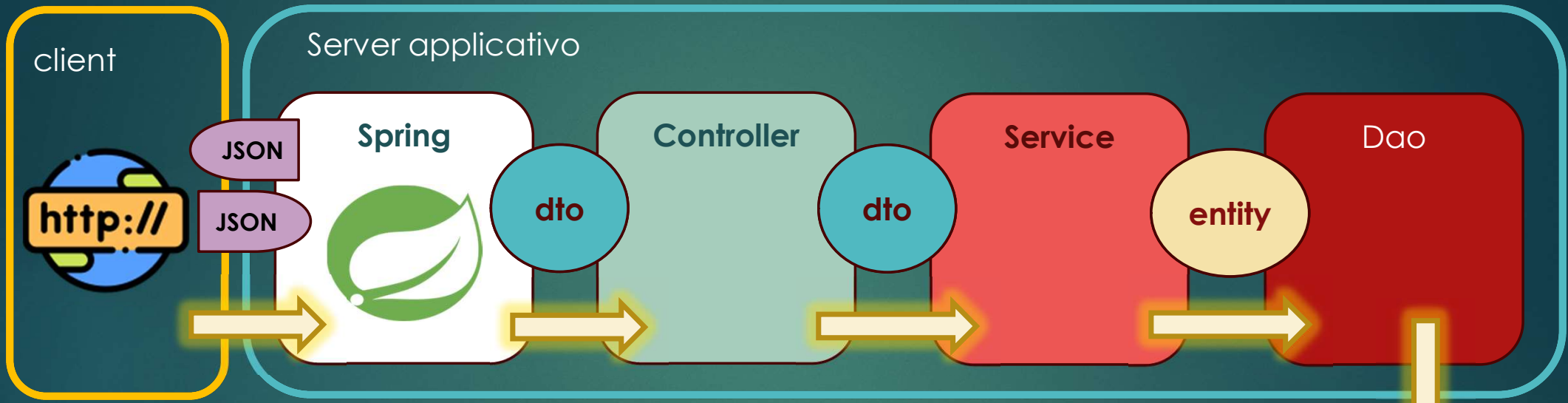
- Tipicamente si usa scomporre il **Model** in 3 layer principali:
  - Layer service → classi che contengono la logica di business ad alto livello
  - Layer dao → classi che contengono la logica a basso livello e quindi i meccanismi per l'accesso ai dati persistenti
    - Queste classi dialogano col database o con gli ORM, motori di persistenza
  - Layer entity → classi che modellano le entità del dominio e i dati salvati nelle tabelle del database

Le classi per trasferire i dati

- Le classi controller scambiano dati con il client.
  - Questi oggetti vengono trasformati da Spring da java in JSON e viceversa.
- Le classi **DTO** sono le classi che rappresentano gli oggetti da inviare al client (o da ricevere dal client)→ **Data Transfer Object**
  - I DTO sono simili alle classi entity ma possono raggruppare anche dati di diversi entity (es. studente con gli esami superati).
- Perché non usare direttamente gli entity?

1. gli entity sono mappati sulle tabelle
2. gli entity sono disaccoppiati dallo strato controller
3. gli entity potrebbero essere collegati ad un ORM (es. Hibernate) e non devono essere 'toccati' fuori dal Model (fuori da service e dao)

Architettura



- Il client invia la richiesta e Spring invoca le controller
- Le controller invocano i service e scambiano dto
- I service invocano i dao e scambiano entity
- I dao scambiano dati con database

Dependency Injection

Inversion of Control

- Una delle più famose caratteristiche di Spring framework è quella di fornire un'implementazione dell'IoC principle.

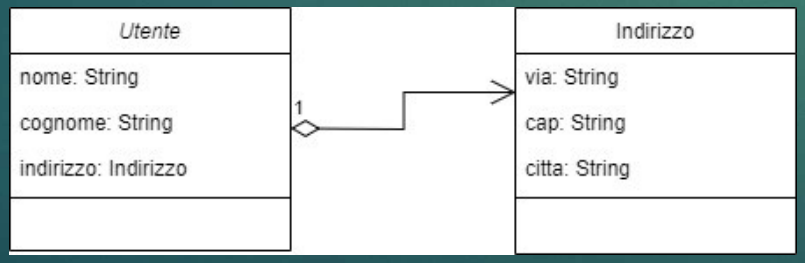
- Il principio IoC – Inversion of Control - è un pattern per cui un componente di livello applicativo **riceve il controllo** da un componente appartenente a una libreria riusabile (solitamente un framework)

La libreria è passiva rispetto al mio componente che è passivo rispetto al framework

Dependency Injection (1/2)

- La Dependency Injection è una forma di IoC
- Consiste nella capacità di un framework (detto Container) di creare oggetti, in inversione di controllo, creando e settando anche gli oggetti di cui sono composti (detti dipendenze).

- Esempio: Un utente ha un nome, un cognome e un indirizzo (che è composto da una via, un cap e una città) → creo 2 classi → Utente ha una dipendenza da Indirizzo



Per costruire un utente devo costruire un indirizzo e poi settarlo nell'utente!

Dependency Injection (2/2)

- Un IoC Container è responsabile di creare oggetti e iniettare le dipendenze, secondo le configurazioni impostate dal programmatore.
- Si può richiedere l'injection di un oggetto in un altro in 3 modalità:
  - via setter method (*relativo all'attributo da iniettare*)
  - via attributo privato (*da iniettare*)
  - via costruttore (*che imposta l'attributo da iniettare*)

- Il modulo **Spring Core** di Spring framework offre un IoC Container, implementato da `org.springframework.context.ApplicationContext`
- La tecnica di injection che useremo sarà quella **via attributo privato**

DI con Spring

- Per impostare una dipendenza si usa l'annotation

  `@Autowired`

- E' possibile utilizzarla per annotare
  - il metodo set

```
@Autowired
public void setResidenza(Indirizzo residenza) {this.residenza = residenza;}
```

  - l'attributo della classe

```
@Autowired
private Indirizzo residenza;
```

  - il costruttore

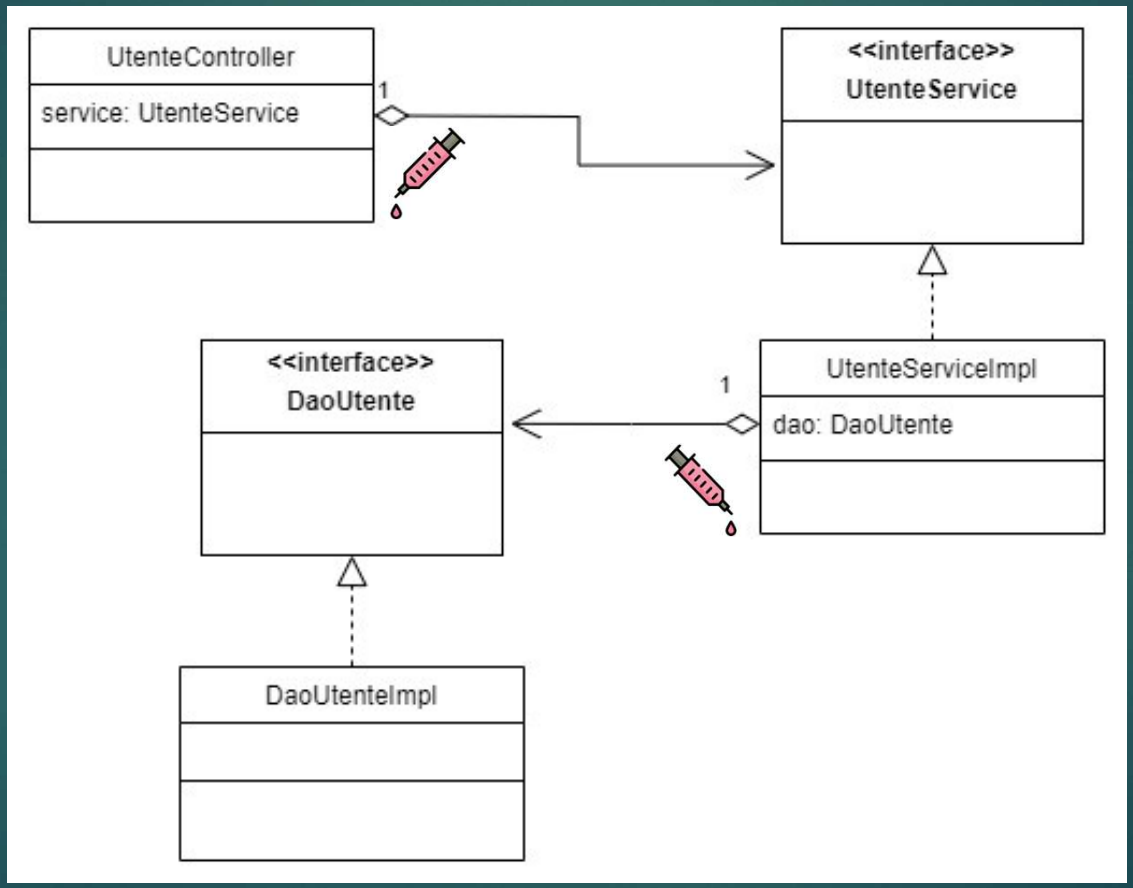
```
@Autowired
public Utente(@Value("mario") String nome, @Value("rossi") String cognome,
                    Indirizzo residenza) {
    this.nome = nome;
    this.cognome = cognome;
    this.residenza = residenza;
}
```

Dove usare l'injection?

- Nell'architettura controller/service/dao abbiamo le seguenti dipendenze:
  - Controller usa Service → Controller dipende da Service
  - Service usa Dao → Service dipende da Dao
- L'injection con Spring prevedere che l'oggetto da iniettare abbia una classe e un'interfaccia di riferimento (perché realizza un proxy)
  - Per tutti i service dovremo avere interfaccia e implementazione
  - Per tutti i dao dovremo avere interfaccia e implementazione

segue diagramma →

Class Diagram



Annotation sugli attributi

- Nella classe **`UtenteController`** ← Classe concreta

```
@Autowired
private UtenteService service;
```
← interfaccia

- Nella classe **`UtenteServiceImpl`** ← Classe concreta

```
@Autowired
private DaoUtente dao;
```
← interfaccia

Annotation sulle classi

- Le classi controller, service e dao verranno create tutte da Spring e iniettate, secondo le dipendenze indicate
  - E' molto importante che si trovino tutte in pacchetti che derivano dal package base del progetto, dove c'è la classe di avvio (classe col metodo main)

- Per indicare la tipologia della classe concreta da iniettatare si usa specificare anche
- **`@Service`** sulla classe concreta di tipo service
- **`@Repository`** sulla classe concreta di tipo dao

Nota: si annotano solo le classi concrete, non le interfacce

- La classe Controller mantiene **`@RestController`**

Sistemi REST

Sistemi REST

- REST è uno stile architetturale formato da vincoli, linee guida e best practice che si utilizza per la realizzazione di sistemi distribuiti.

- **REST non è un'architettura**
- L'obiettivo è quello di regolamentare lo sviluppo di sistemi complessi che altrimenti evolverebbero in maniera "caotica"
- **Un sistema che si attiene a tali vincoli prende il nome di sistema RESTful**

- L'acronimo **REST**, **RE**presentational **S**tate **T**ransfer ("trasferimento dello stato di rappresentazione") deriva dalla tesi di dottorato di Roy **Fielding** intitolata "stili architetturali e progettazione di architetture software basate sul networking"
  - La tesi risale all'anno 2000
  - Fielding è uno dei principali autori delle specifiche del protocollo **HTTP**

Vincoli: stateless e risorse

- Lo stile REST prevede una serie di vincoli.
- Uno dei principali vincoli afferma che il sistema deve essere stateless, cioè non deve essere necessario l'uso della sessione (sebbene non sia vietato)
- Altro importante vincolo invece è quello che impone che ogni richiesta venga processata al fine di produrre una risorsa che il programma client userà.

Cos'è un architettura stateful?

- E' un'architettura dove il servizio - erogato dal server per il client - memorizza lo stato del client sul server

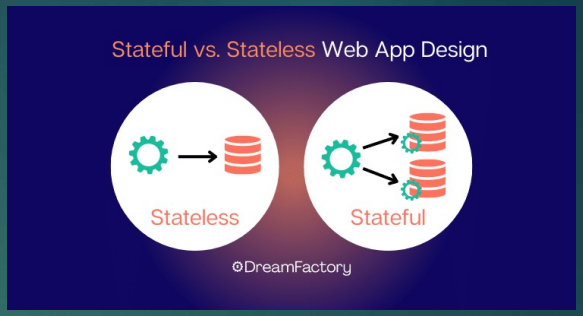
- Esempio: pizzeria con camerieri dedicati ai clienti
  - Vantaggio → il cameriere conosce dettagli del cliente e lo stato dell'ordine
  - Svantaggi →
    - modifiche all'ordine possono diventare ONEROSE se non si *accede* al cameriere assegnato
    - se ci sono molti clienti, il carico di lavoro NON si distribuisce facilmente
  - Il sistema non gestisce bene imprevisti, modifiche, aggiornamenti e non è scalabile (al crescere delle richieste il servizio scende di qualità a meno di non assumere altri camerieri) → quindi è tendenzialmente molto 'costoso'!

Cos'è un architettura stateless?

- E' un'architettura dove il servizio - erogato dal server per il client – NON memorizza lo stato del client sul server → senza stato

- Esempio: pizzeria con camerieri che accedono ad un sistema centralizzato per la gestione degli ordini.
  - Vantaggio →
    - qualunque cameriere può accedere allo stato dell'ordine di qualunque cliente ed effettuare variazioni.
    - se ci sono molti clienti, il carico di lavoro si può distribuire facilmente perché tutti i camerieri sono intercambiabili
  - Svantaggi → Il cameriere NON conosce dettagli e preferenze del cliente
  - Il sistema gestisce **molto meglio** imprevisti, modifiche, aggiornamenti e risulta scalabile (al crescere delle richieste il servizio mantiene qualità, senza dover necessariamente assumere altri camerieri) → quindi è tendenzialmente meno 'costoso'!

Perché servizi stateless?



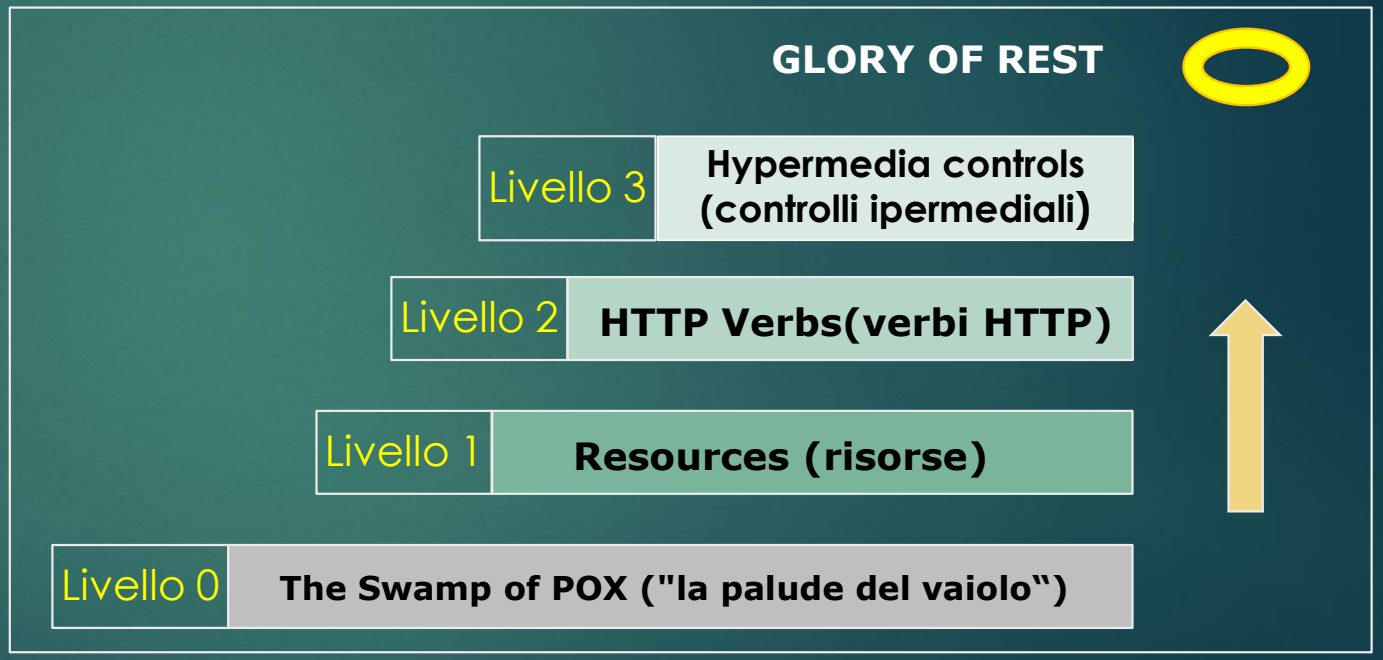
- Perché c'è un vantaggio economico!

- Nei servizi statefull le varie chiamate usavano la sessione sul server per salvare dati parziali e la **sessione è molto costosa!**

- Nei servizi REST la sessione non si usa (nel senso che non è necessaria)
- I servizi sono **stateless** e la gestione della conversazione viene spostata sul client (che è un programma su una macchina)
  - Le azioni 'su piu step' usano la memoria del client e, solo alla fine, parte la chiamata al server (come chiamata stateless)
  - Chi scrive il servizio (sul server) è totalmente disaccoppiato dal client
- Vantaggio economico + disaccoppiamento Client/Server

La scala di livelli

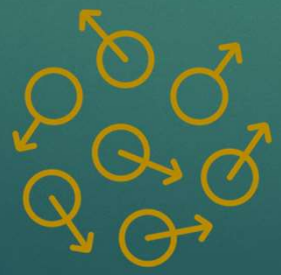
- Fielding teorizzò una scala di livelli di compatibilità allo stile REST.
- **Livello zero** (il "Far West "): sistemi a basso grado di maturità in cui tutte le risorse tecnologiche sono disponibili, e tutti gli stili sono ammessi
- **Livello tre**: stato di totale maturità. Sistemi che rispettano i vincoli e dunque possono chiamarsi **RESTful**.



I livelli 1 e 2 si realizzano sempre, mentre per il livello 3 si può valutare se/come realizzarlo

Livello 0: The Swamp of POX

- POX sta per plain old XML ma in inglese significa “vaiolo”
- I sistemi POX
  - scambiano messaggi in modo sincrono attraverso il protocollo HTTP
  - Non possiedono il concetto di risorsa (ma di funzione)
  - Non sfruttano i verbi HTTP (tipicamente agiscono solo con GET e POST)

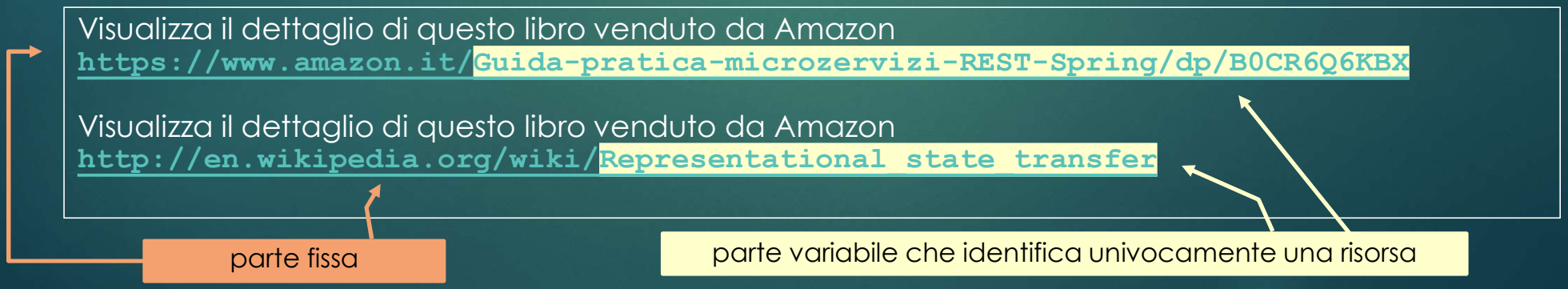


Livello 1: resources

- **Prevede di organizzare i servizi in termini di risorse**
- Una risorsa è una qualunque cosa sia accessibile nel web, ossia il cui stato sia trasferibile tra server e client
- Esempi
  - un libro venduto on-line,
  - le previsioni meteo di Londra,
  - un item oggetto di un'asta eBay,
  - i dati identificativi di un volo aereo,
  - un valore di cambio valuta,
  - i prezzi di un prodotto finanziario e le informazioni relative allo stesso,
  - la spiegazione di un termine tratta da Wikipedia

Livello 1: resources

- Il concetto di risorsa è centrale per le architetture REST
- Questa direttiva base richiede di assegnare a tutte le risorse un identificativo univoco → **URI** (Uniform Resource Identifier)
- Lo schema URI esiste già, è largamente utilizzato, non presenta problemi, ed è uno standard internazionale
- Esempio di URI reale:



Livello 1: resources

- Dal punto di vista del server, le risorse possono essere viste come **oggetti** del dominio "esposte" nel modello REST

URI	risposta
http://myUniversity.it/professors	Elenco di tutti i professori dell'università
http://myUniversity.it/professors/123	Dati del professore con id=123
http://myUniversity.it/professors/123/lectures	Lezioni tenute dal professore con id=123
http://myUniversity.it/lectures	Elenco di tutte le lezioni tenute all'università
http://myUniversity.it/lectures/ing_sw	Dati del corso di Ingegneria del software (ing_sw)
http://myUniversity.it/lectures/ing_sw/examsessions	Elenco delle sessioni di esame relative al corso di ing_sw
http://myUniversity.it/examsessions	Elenco di tutte le sessioni di esame (di tutti i corsi)
http://myUniversity.it/examsessions/110	Dati della sessione di esame con id=110

Regole dei link

- Tutte le azioni sulle risorse di un certo tipo prevederanno la URI col nome della risorsa al plurale.

Esempio: **http://myUniversity.ac.uk/professori**

- Se devo eseguire un'azione su una risorsa specifica aggiungo l'ID all'URI

Esempio: leggo tutti i dati relativi al professore con id :123
http://myUniversity.ac.uk/professori/123

- Se devo inviare parametri sciolti per l'esecuzione della richiesta, li posso aggiungere sulla query string

Esempio: modifico l'email del professore con id:123
http://myUniversity.ac.uk/professori/123?newMail=Giordano@uni.na.it

- Se devo inviare oggetti strutturati per l'esecuzione della richiesta, li aggiungo nel body (magari in formato JSON)

Esempio: aggiungo una materia d'insegnamento al professore con id:123
http://myUniversity.ac.uk/professori/123

Carico la materia nel body in formato json →

```
{
    "idMateria":55,
    "titolo":"Analisi II",
    "crediti":10
}
```

La potenza dei link

Vantaggi dei link:

- Sono comprensibili dagli essere umani
- Sono leggibili dalle macchine che sanno come elaborali (vedi ad esempio il browser)
- **Sono uno standard globale:** rappresentano link (puntatori a risorse "linkate")
  - un web services potrebbe ritornare come risposta un link che rappresenta l'invocazione ad una risorsa presente su un server ubicato in qualsiasi parte del mondo
- Sono associati all' HTTP che possiede il **Content Type Negotiation.**
  - ad esempio, se il link punta ad un video, verrà interpretato con un lettore video
- E' possibile assegnare informazioni aggiuntive nell'header HTTP

URI progressivi

- All'interno di un dominio sono spesso presenti oggetti associati ad altri per mezzo di relazioni has_a.
  - Esempio: Un professore HA un elenco di materie (che insegna)
  - Si dice che l'oggetto Materia è **correlato** al Professore.
- Quando si vuole accedere ai dati correlati di un risorsa (es. le materie insegnate da un professore) si deve seguire la regola stilistica delle URI progressive

  `http://domain/risorsaPrimaria/risorsaCorrelata`
- Per ottenere tutte le materie del professore con id:123, invoco:

  `http://universita.napoli.it/professori/123/materie`

Livello 2: verbi HTTP

- Il protocollo HTTP mette a disposizione 8 verbi con cui il client può indicare il tipo di azione che richiede al server

- Storicamente, le funzioni sono state sempre **mal sfruttate:** le applicazioni web utilizzavano quasi esclusivamente i metodi GET e POST per compiere qualsiasi tipo di operazione

- Il livello 2 REST richiede di **utilizzare i seguenti metodi HTTP** standard :

  GET, PUT, PATCH, POST, DELETE

  in modo congruente

Descrizione di GET

- Il metodo **GET** serve per leggere dati dal server
- Si usa per recuperare tutte le informazioni, sotto forma di un'entità, relative alla risorsa identificata dall'URI presente nella richiesta
- Esempio:
  - Leggere il dettaglio di un prodotto in vendita
  - Leggere tutti gli esami sostenuti da uno studente
- Se l'URI si riferisce a un processo, la risposta deve contenere un'entità che contenga a sua volta i risultati dell'esecuzione del suddetto processo invece di restituire le informazioni relative al processo stesso

Descrizione di POST

- Il metodo **POST** serve per salvare dati sul server
- Permette ai client di inserire (save) o eventualmente aggiornare (update) una risorsa.
- Esempi tipici sono:
  - invio di form compilate al lato client
  - invio di un commento a un blog
  - uploading dei file
- A differenza di GET, dove solo un URL e le intestazioni possono essere presenti, il metodo POST può anche includere la parte **body**
  - superamento del vincolo di dimensione (lunghezza della URL)

Descrizione di PUT e PATCH

- Il metodo **PUT** serve per effettuare modifiche totali alla risorsa sul server

- Permette ai client di aggiornare (update) o eventualmente inserire (save) una risorsa.
- Se la richiesta si riferisce a una risorsa già esistente → si intende come richiesta di modifica
  - l'entità inviata rappresenta una **versione aggiornata** di quella presente sul server
  - il server torna una risposta con stato **200 (OK) o 204 (no content)**
- Se la richiesta si riferisce a una risorsa NON esistente → si intende come richiesta di inserimento
  - il server deve restituire una risposta con stato **201 (created)**
- Se la risorsa non può essere creata o modificata, il server deve rispondere con un appropriato errore che riflette la natura del problema → **501 (not Implemented).**

- Il metodo **PATCH** lavora come PUT, ma serve per modificare un singolo campo di una risorsa. Pertanto ha solo la logica dell'aggiornamento (update).

NB: PUT e POST si assomigliano ma dopo chiariremo che differiscono per idempotenza

Descrizione di DELETE

- Il metodo **DELETE** serve per eliminare una risorsa sul server.
- Il server dovrebbe restituire
  - **stato 200 (OK)** → si elimina la risorsa o si sposta in una posizione non più accessibile
  - **stato 202 (accepted)** → l'azione non è stata ancora compiuta ma comunque è stata accettata
  - **204 (no content)** → si elimina la risorsa ma la risposta non include un'entità
- NB: Il client non ha garanzia che l'operazione sia stata eseguita, anche se il codice di stato restituito dal server indica che l'azione è stata completata con successo

Livello 2: sicurezza e idempotenza

- Nell'ambito del livello 2, ci sono altri 2 criteri che caratterizzano i metodi HTTP utilizzati in ambito REST.
- Criterio del metodo sicuro → deve essere garantito dalle richieste GET
- Criterio di idempotenza → deve essere garantito per tutte richieste, tranne che per quelle POST

Metodi sicuri

- Un metodo è definito **sicuro** se, a seguito della relativa chiamata, lo stato del server non cambia.
  - **L'unica funzione HTTP sicura deve essere GET**
- **Il metodo GET dovrebbe essere realizzato in modo da essere sempre sicuro**
  - E' sbagliato (secondo le regole REST) utilizzare GET per modifiche sul server
  - Ad esempio non è possibile alterare lo stato del server richiedendo più volte (con GET) l'ottenimento dei dati di un professore universitario
- PUT, PATCH, DELETE e POST eseguono modifiche → non sono sicuri per definizione
- NOTA: qui non si sta parlando di come mettere in sicurezza le API Rest ma solo di quali metodi/funzioni HTTP vanno considerate sicure

Metodi idempotenti

- Il concetto di idempotenza viene dalla matematica
- Una operazione è *idempotente se il risultato di tale operazione è indipendente dal numero di volte in cui viene eseguita.*
  - l'aggiunta di zero ad un numero → è idempotente.
  - l'operazione di incremento "x++" → NON è idempotente
- Un metodo è idempotente se effettuando multiple chiamate (a parità di condizioni iniziali), si ottiene comunque lo stesso risultato.
  - Significa essenzialmente che il risultato di una richiesta eseguita con successo è indipendente dal numero di volte in cui viene eseguita
- Tutte funzioni HTTP , **tranne POST ,** sono pensate per essere idempotenti
- Lo sviluppatore deve scrivere delle API REST sicure ed idempotenti

Riepilogo

- **GET è per natura idempotente (e sicuro)**
  - Invocando una richiesta GET non si cambia mai lo stato delle risorse sul server
- **PUT, PATCH e DELETE devono essere idempotenti**
  - **PUT** → Una operazione PUT deve modificare lo stato di una risorsa sovrascrivendo il vecchio stato con uno nuovo. Di conseguenza, quando si invoca la prima volta, la risorsa effettivamente si aggiorna; se si invoca una seconda volta la risorsa si aggiorna con gli stessi dati della prima richiesta rimanendo di fatto invariata
  - **DELETE** → Una operazione DELETE deve cancellare una risorsa in base ad un certo criterio. Di conseguenza, quando si invoca la prima volta la risorsa viene trovata e cancellata; se si invoca una seconda volta la risorsa non viene trovata e quindi non cancellata (lasciando di fatto lo stato della risorsa -risorsa cancellata- invariato)
    - Si otterà la prima volta uno status 200 (OK) e dalla seconda volta in poi 204 (No Content)
- **POST è per natura NON idempotente**
  - Pensiamo, ad esempio, all'inserimento di un commento ad un post. A meno di tecniche particolari, se una richiesta POST è eseguita n volte, la relativa esecuzione genera n commenti uguali

Vantaggi della proprietà di idempotenza

- SAFE: il client può ripetere la medesima richiesta più volte senza preoccuparsi di alterare lo stato del server
- INTEGRATION: l'integrazione tra sistemi è più agevole perché semplifica la semantica dei connettori
- Esempio:
- Un client esegue una richiesta di una operazione idempotente ma NON riceve risposta in tempo utile per il sopraggiungimento del time-out.
- Invece di intraprendere complicate procedure per cercare di capire cosa sia successo sul server *(richiesta non ricevuta, richiesta ricevuta e processata con risposta non pervenuta in tempo utile, etc.),* il client può tranquillamente emettere nuovamente la medesima richiesta senza alcun rischio!

Differenze tra POST e PUT

- Formalmente, POST e PUT possono essere utilizzate entrambe per inserire o modificare:
  - **POST → save** crea una nuova risorsa, ma se esiste la aggiorna
  - **PUT → update** aggiorna una risorsa già esistente ma se non esiste la crea (save)
- Quindi quando utilizzare PUT e quando POST?
- La vera differenza tra le due operazione va ricercata nelle conseguenze del loro operare sulla risorsa ed è legata al concetto di indempotenza
  - **PUT è una funzione idempotente** → si usa per azioni con comportamento idempotente
  - **POST è una funzione NON idempotente** → si usa per azioni con comportamento non idempotente

Livello 3: hypermedia controls e HATEOAS

- Il livello 3 è caratterizzato dal vincolo noto con l'acronimo di **HATEOAS, HyperText As The Engine Of Application State** → ipertesto come motore dello stato delle applicazioni.

- Il termine **ipermedia** (hypermedia) si riferisce all'insieme di elementi grafici, audio, video, testo e collegamenti ipertestuali che si usano per creare un genere non-lineare di media di informazione
- Un client interagisce con un'applicazione di rete interamente attraverso ipermedia forniti dinamicamente dal server.
  - Un client REST pertanto non ha bisogno di conoscere preliminarmente come interagire con una particolare applicazione o con un server; tutto quello di cui ha bisogno è un punto di accesso all'ipermedia.

- Al fine di rispettare il vincolo **HATEOAS**, **il server deve restituire nella risposta anche (o soltanto) l'insieme delle possibili azioni richiedibili.**
  - Segue esempio → →

HATEOAS

- Questa è la risposta ad una richiesta per le sessioni di esame
- Le sessioni sono 2 ma non contengono dati bensì link per eseguire le successive operazioni a partire dalla sessione data.

```
HTTP/1.1 200 OK ...
<examSession lecture="ING_SW_ARCH01”>
  <session self="http://myUniversity.ac.uk/examsessions/110"
           date="04-11-2011">
    <room self="http://myUniversity.ac.uk/buildings/senate_house_01/222" />
    <link rel="http://myUniversity.ac.uk/examSessions/reserve"
        uri ="/examsessions/110" />
  </session>
  <session self="http://myUniversity.ac.uk/examsessions/211"
           date="18-11-2011">
    <room self="http://myUniversity.ac.uk/buildings/senate_house_01/115" />
    <link rel="http://myUniversity.ac.uk/examSessions/reserve"
         uri ="/examsessions/211" />
  </session>
</examSession >
```

link alla risorsa `sessionediesame con id:110`

link all'**azione** di prenotazione della sessione 110

link alla risorsa `sessionediesame con id:211`

link alla risorsa aula di esame relativa alla sessione 211

Vantaggi di HATEOAS

- Il vincolo **HATEOAS** serve a disaccoppiare client e server in modo da consentire al server di evolvere autonomamente le proprie funzionalità.

- Inoltre la tecnica di restituire solo i collegamenti alle ulteriori risorse collegate alla risorsa richiesta consente di avere un minor carico sulla rete
  - Infatti se il client chiede una risorsa che rappresenta un oggetto complesso, non ottiene tutto l'oggetto ma una sorta di contenitore con l'insieme delle operazioni che si possono eseguire sull'oggetto
  - In base alle reali esigenze, può ottenere i dati successivamente (sfruttando i link ricevuti nella risorsa)

Vantaggi uso di REST

- La separazione delle responsabilità dettata dalle linee guida REST
  - semplifica l'implementazione dei componenti
  - riduce la complessità della semantica dei connettori
  - migliora le prestazioni e aumenta la scalabilità dei componenti server-side.
- REST consente l'elaborazione intermedia forzando i messaggi a essere auto-descrittivi. Infatti:
  - l'interazione tra le richieste è stateless
  - i metodi standard
  - si utilizzano i media type per indicare la semantica e lo scambio di informazioni

- Concludendo: le architetture RESTful sono basate sulla più grande infrastruttura informatica creata dall'uomo, **il Web**, e permettono a qualsiasi sistema ipermediale distribuito di possedere importanti proprietà, quali semplicità, scalabilità, portabilità, visibilità e tipicamente elevate performance.