

Spring

Riccardo Cattaneo



Web Service

Il Web è nato negli anni '90 come una piattaforma per la condivisione di documenti distribuiti su diverse macchine e tra loro interconnessi. Il tutto è nato e si è evoluto grazie alla standardizzazione di alcuni semplici concetti:

URI – meccanismo per individuare risorse in una rete

HTTP – protocollo semplice e leggero per richiedere una risorsa ad una macchina

HTML – linguaggio per la rappresentazione dei contenuti

Questa semplice idea iniziale si è evoluta nel corso degli anni non tanto nei concetti di base, quanto nel modo di intenderli e di utilizzarli.

Al testo si sono aggiunti contenuti multimediali, i documenti sono generati dinamicamente e non pubblicati come pagine statiche, e poi il Web esce dalla visione esclusivamente ipertestuale per diventare un contenitore di applicazioni software interoperabili: una piattaforma applicativa distribuita.

Questa prospettiva ha dato origine, intorno all'anno 2000, al concetto di **Web Service**: un sistema software progettato per supportare un'interazione tra applicazioni, utilizzando le tecnologie e gli standard Web. Il meccanismo dei Web Service consente di far interagire in maniera trasparente applicazioni sviluppate con linguaggi di programmazione diversi, che girano su sistemi operativi eterogenei.

Questo meccanismo consente di realizzare porzioni di funzionalità in maniera indipendente e su piattaforme potenzialmente incompatibili facendo interagire i vari pezzi tramite tecnologie Web e creando un'architettura facilmente componibile.



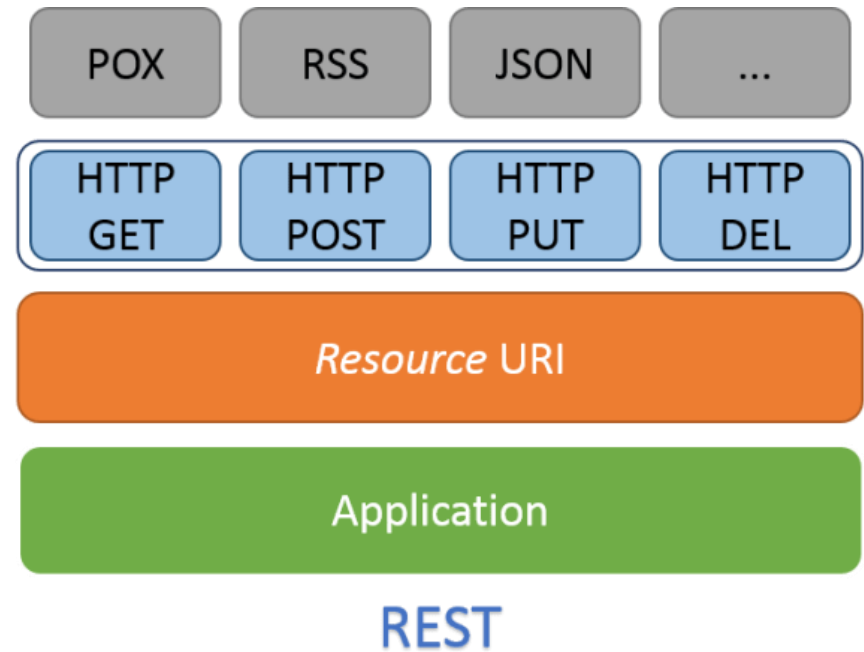
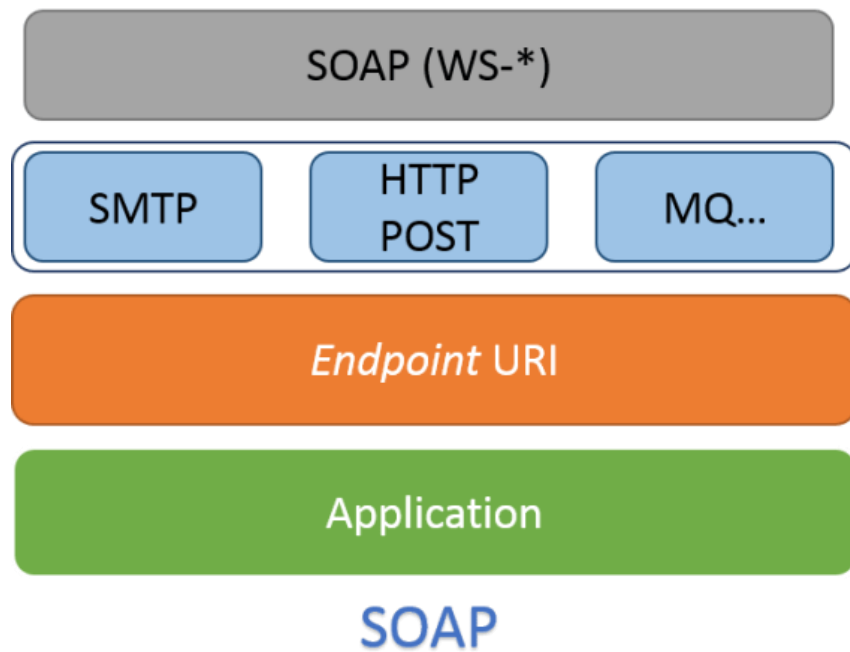
SOAP vs REST

Allo stato attuale esistono due approcci alla creazione di Web Service:

Un approccio è basato sul protocollo standard **SOAP** (Simple Object Access Protocol), per lo scambio di messaggi per l'invocazione di servizi remoti, si prefigge di riprodurre in ambito Web un approccio a chiamate remote, Remote Procedure Call.

Un secondo approccio è ispirato ai principi architetturali tipici del Web e si concentra sulla descrizione di risorse, sul modo di individuarle nel Web e sul modo di trasferirle da una macchina all'altra.

Questo è l'approccio che analizzeremo in questo corso e che prende il nome di **REST** (REpresentational State Transfer).



Molti sistemi esistenti aderiscono ancora a SOAP, mentre REST, il cui avvento è successivo, è spesso considerato come un'alternativa più rapida negli scenari web.

REST è un insieme di linee guida con un'implementazione flessibile, mentre SOAP è un protocollo con requisiti specifici, come la messaggistica XML.

La differenza principale sta nel fatto che SOAP (Simple Object Access Protocol), come suggerisce l'acronimo, è un Protocollo a se stante, mentre REST (REpresentational State Transfer) è uno schema architetturale e quindi può essere implementato con sistemi differenti.

SOAP - E' uno standard con delle regole ben precise ed uno schema molto rigido, basato sull'XML. Lo scambio di dati avviene tramite degli "endpoint" che funzionano in base ad un "contratto" sulla modalità di scambio dati e di comunicazione.

REST - Rest è un'architettura, e quindi può essere implementata in diversi modi. Oggi si usa molto spesso JSON per lo scambio dati, perché è fruibile facilmente sia dai client che dai server, oltre ad essere molto più leggero e semplice da usare.

JSON

JSON acronimo di JavaScript Object Notation, è un formato adatto all'interscambio di dati fra applicazioni client/server. È basato sul linguaggio JavaScript ma ne è indipendente. Questa è la sintassi :

```
{  
    "nome": "Mario",  
    "cognome": "Rossi",  
    "active": true,  
    "eta": 42,  
    "compleanno": {  
        "day": 1,  
        "month": 1,  
        "year": 2000  
    },  
    "lingua": [ "it", "en" ]  
}
```

I tipi di dati supportati da questo formato sono:

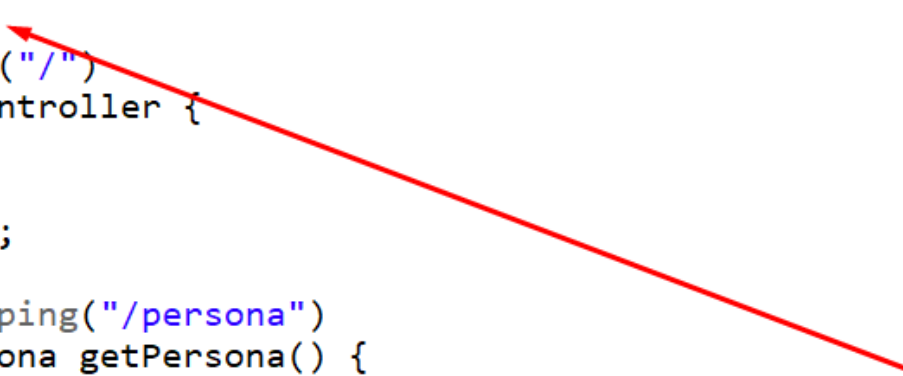
- booleani (true e false);
- interi, numero in virgola mobile;
- stringhe racchiuse da doppi apici ("");
- array (sequenze ordinate di valori, separati da virgole e racchiusi in parentesi quadre []);
- array associativi (sequenze coppie chiave-valore separate da virgole racchiuse in parentesi graffe);
- null.

Modificheremo adesso la nostra applicazione per far si che esponga microservizi rest. Il giro del Model View Controller ce l'abbiamo (anche se il nostro repository non fa ancora chiamate reali al database). Andiamo a modificare ora il nostro controller per far si che esponga dei microservizi.

Cosa dobbiamo fare ? Andiamo sul nostro Controller, eliminiamo l'Annotation `@Controller` e sostituirla con **`@RestController`**

- > demo [boot]
- ✓ demo-mvc [boot]
 - ✓ src/main/java
 - com.example.demo
 - Controller.java
 - DemoMvcApplication.java
 - Repository.java
 - Service.java
 - com.example.demo.entity
 - > src/main/resources
 - > src/test/java
 - > JRE System Library [JavaSE-11]
 - > Maven Dependencies
 - > src
 - target
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml
 - > test-scope [boot]

```
1 package com.example.demo;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5
6
7
8
9 @RestController
10 @RequestMapping("/")
11 public class Controller {
12
13     @Autowired
14     Service ser;
15
16     @RequestMapping("/persona")
17     public Persona getPersona() {
18
19         Persona p = ser.getPersonaService();
20
21         System.out.println(p.getCognome());
22         System.out.println(p.getNome());
23
24         return p;
25     }
26
27 }
```



Quindi, quando il client farà una richiesta (request), la request utilizzerà dei comandi chiamati «verbi» che sono :

- **GET** (recupra dati)
- **POST** (inserire dei dati)
- **PUT** (aggiorna i dati)
- **DELETE** (elimina i dati)

Ora che andremo a creare dei metodi, questi metodi avranno delle annotazioni che andranno ad identificare uno dei 4 verbi visti in precedenza. I verbi comunicano al Server cosa deve fare.

GET

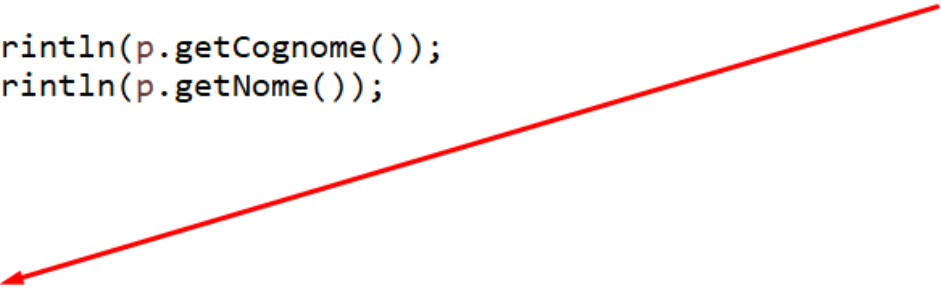
1 - Proviamo subito a creare un **metodo GET** annotandolo con `@GetMapping`, il metodo lo chiamiamo `getListPersone()`

- demo [boot]
- demo-mvc [boot]
 - src/main/java
 - com.example.demo
 - Controller.java
 - DemoMvcApplication.java
 - Repository.java
 - Service.java
 - com.example.demo.entity
 - src/main/resources
 - src/test/java
 - JRE System Library [JavaSE-11]
 - Maven Dependencies
 - src
 - target
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml
 - test-scope [boot]

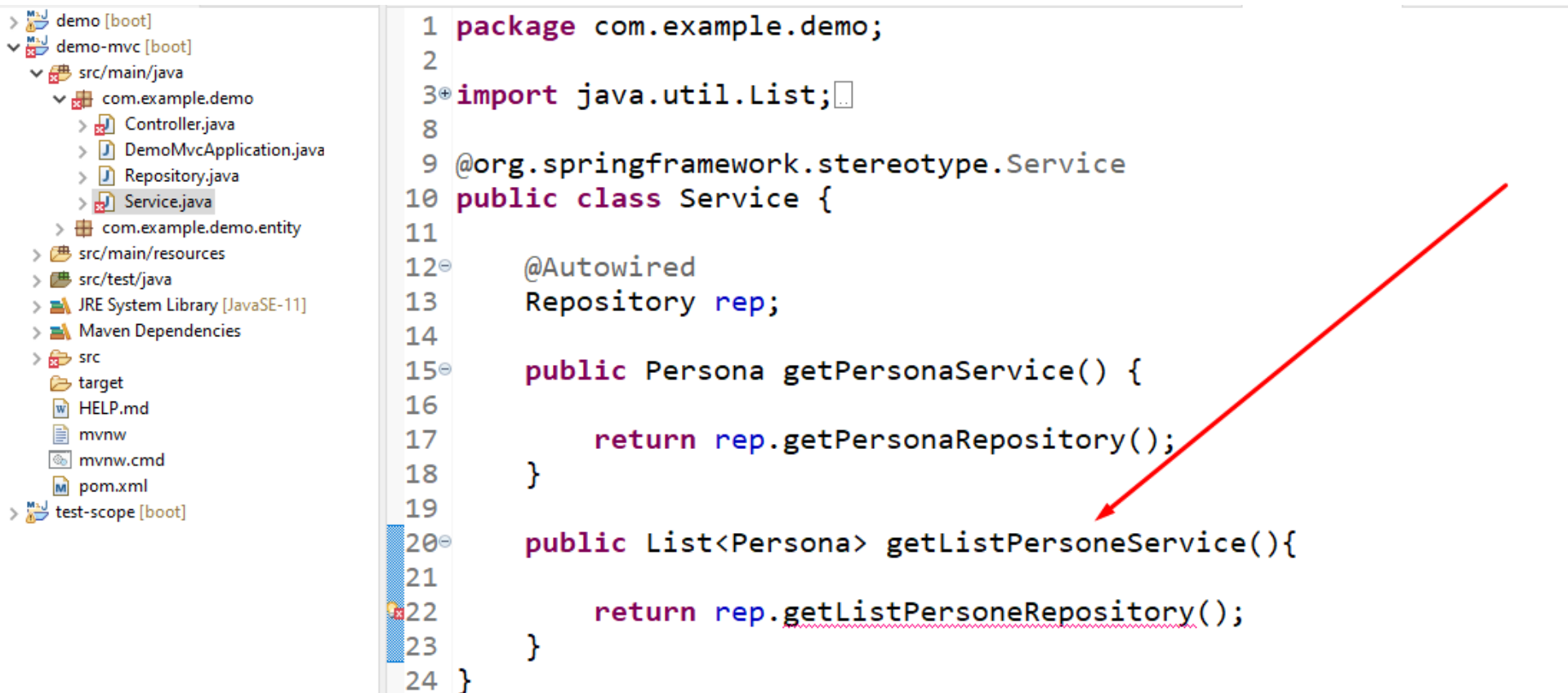
```

11
12 @RestController
13 @RequestMapping("/")
14 public class Controller {
15
16     @Autowired
17     Service ser;
18
19     @RequestMapping("/persona")
20     public Persona getPersona() {
21
22         Persona p = ser.getPersonaService();
23
24         System.out.println(p.getCognome());
25         System.out.println(p.getNome());
26
27         return p;
28     }
29
30     @GetMapping
31     public List<Persona> getListPersone(){
32
33         // chiamata al service
34
35     }
36
37 }

```



2 – Andiamo ora a creare il nostro Service, quindi un nuovo metodo che chiama il repository e torna una lista di Persone :



```
1 package com.example.demo;
2
3 import java.util.List;
4
5 @org.springframework.stereotype.Service
6 public class Service {
7
8     @Autowired
9     Repository rep;
10
11     public Persona getPersonaService() {
12
13         return rep.getPersonaRepository();
14     }
15
16     public List<Persona> getListPersoneService(){
17
18         return rep.getListPersoneRepository();
19     }
20 }
21
22
23
24 }
```

The screenshot shows an IDE with a project structure on the left and a Java code file on the right. The project structure includes a package `com.example.demo` with files `Controller.java`, `DemoMvcApplication.java`, `Repository.java`, and `Service.java`. The code file `Service.java` contains the following code:

```
1 package com.example.demo;
2
3 import java.util.List;
4
5 @org.springframework.stereotype.Service
6 public class Service {
7
8     @Autowired
9     Repository rep;
10
11     public Persona getPersonaService() {
12
13         return rep.getPersonaRepository();
14     }
15
16     public List<Persona> getListPersoneService(){
17
18         return rep.getListPersoneRepository();
19     }
20 }
21
22
23
24 }
```


A red arrow points from the text "torna una lista di Persone" to the method name `getListPersoneService` in the code.

3 – Andiamo ora a creare il nostro Repository che ci restituirà una lista di persone :

```
20 public List<Persona> getListPersoneRepository(){  
21  
22     ArrayList<Persona> listaPersone = new ArrayList<Persona>();  
23  
24     Persona p1 = new Persona();  
25     p1.setNome("Mario");  
26     p1.setCognome("Rossi");  
27  
28     Persona p2 = new Persona();  
29     p2.setNome("Luigi");  
30     p2.setCognome("Bianchi");  
31  
32     listaPersone.add(p1);  
33     listaPersone.add(p2);  
34  
35     return listaPersone;  
36 }
```

4 – Per completare il giro ora torniamo sul nostro controller ed andiamo ad includere la chiamata al Service che mancava, e modifichiamo il GetMapping in modo da poterlo richiamare.

```
30 @GetMapping("/listapersone")  
31 public List<Persona> getListPersone(){  
32  
33     List<Persona> listaP = ser.getListPersoneService();// chiamata al service  
34  
35     return listaP;  
36  
37 }  
38
```



5 – Come prova finale facciamo partire la nostra applicazione ed apriamo il browser all'indirizzo del servizio :

<http://localhost:8080/listapersona>

Come possiamo vedere il servizio ci risponde con una lista (un array) , abbiamo esposto in questo momento un servizio Rest.

← → ↻ ⓘ localhost:8080/listapersone

📁 Progetti 📁 Immagini 📁 Java 📁 Preferiti 📺 How to Create Onli... 🌐 Bando Voucher Digi...

```
[{"nome":"Mario","cognome":"Rossi"}, {"nome":"Luigi","cognome":"Bianchi"}]
```

POST

1 – Andiamo subito a creare un nuovo metodo all'interno del nostro Controller e annotiamolo con l'annotazione `@PostMapping`, il metodo lo chiameremo `inserisciPersona()` e prenderà in input e tornerà un oggetto `Persona` :

```
40 @PostMapping
41 public Persona inserisciPersona(Persona p) {
42
43     return ser.inserisciPersonaService(p);
44 }
45
```

2 - Andiamo ora a creare il nostro Service, quindi un nuovo metodo che chiama il repository ed inserisce una persona :

```
24  
25 public Persona inserisciPersonaService(Persona p) {  
26  
27     return rep.inserisciPersonaRepository(p);  
28 }  
29 }  
30 |
```


3 - Andiamo ora a creare il nostro Repository che inserirà una nuova persona nel database :

```
public Persona inserisciPersonaRepository(Persona p) {  
    // insert nel database  
    return p;  
}
```

4 – Per completare il giro ora torniamo sul nostro controller ed andiamo ad includere la chiamata al Service che mancava, e modifichiamo il PostMapping in modo da poterlo richiamare.

Come ultima cosa, per far funzionare tutto, aggiungiamo una nuova annotazione che serve ai parametri in ingresso per essere utilizzati da Spring : **@RequestBody**

@RequestBody

Questa annotation riceve SOLO oggetti in formato JSON (di solito questa metodologia è usata solo per le chiamate REST).

Si tratta di anteporre al tipo del parametro che vogliamo ricevere l'annotation @RequestBody in modo da indicare che tale oggetto sarà presente nel body della Request come JSON.

@RequestBody vs @RequestParam

Da non confondere il RequestBody con RequestParam. Con RequestBody sto passando in ingresso come parametro un oggetto (JSON), mentre con RequestParam viene passato il singolo elemento che può essere passato ad esempio da un form di una applicazione web.

```
@PostMapping("/inseriscipersona")
public Persona inserisciPersona(@RequestBody Persona p) {

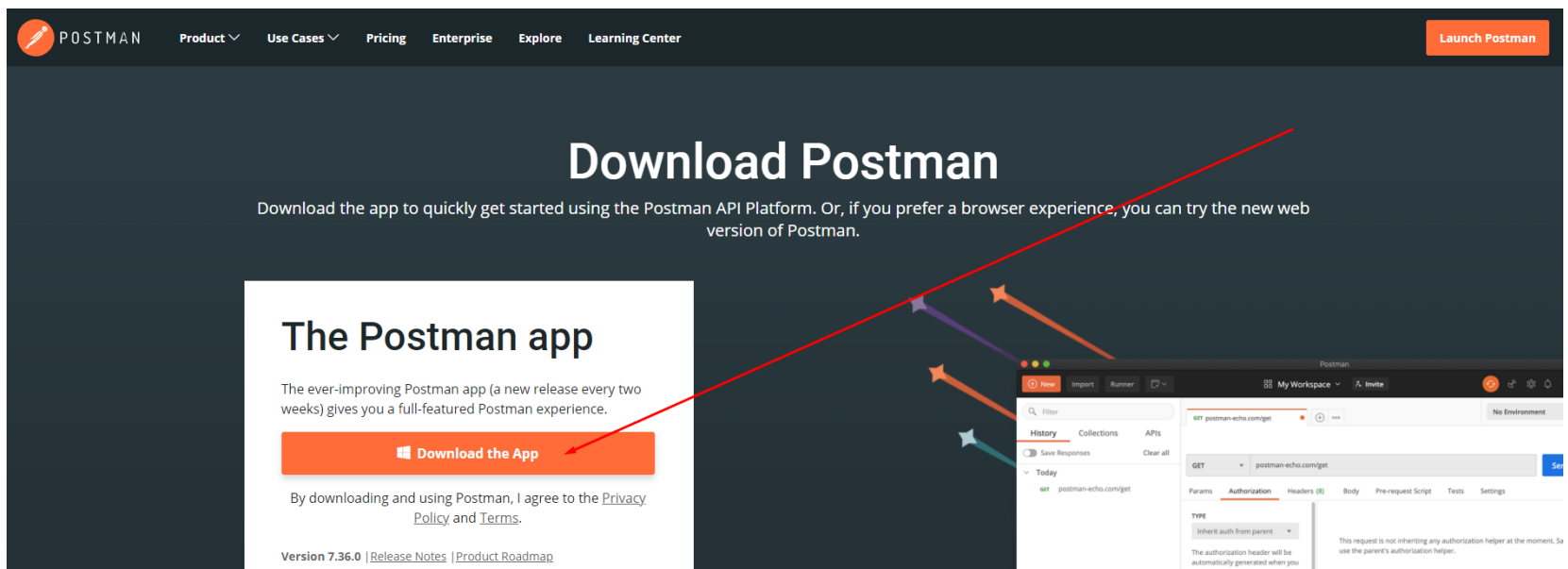
    return ser.inserisciPersonaService(p);
}
```

Andiamo ora a testare il servizio. Mentre per il get abbiamo potuto utilizzare la url, con il post e con gli altri verbi dobbiamo utilizzare un software che servirà a testare servizi rest (non posso passare attraverso la url)... in questo corso utilizzeremo il software **postman**...

Postman

Andiamo alla seguente pagina web, scarichiamo ed installiamo l'app (in automatico vi verrà proposto il download per il vostro sistema operativo):

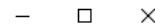
<https://www.postman.com/downloads/>





Postman

File Edit View Help



Sign in with Single Sign-On (SSO)



POSTMAN

Why Sign Up?

- Organize all your API development within Postman Workspaces
- Sync your Postman data across devices
- Backup your data to the Postman cloud
- It's free!



Create Postman Account [Sign In](#) instead?

Email

Username

Password

[SHOW](#)

☐ Sign up to get product updates, news, and other marketing communications.

☒ Keep me signed in

By creating an account, I agree to the [Terms](#) and [Privacy Policy](#).

Create free account

or




Sign up with Google

[Skip signing in and take me straight to the app](#)

You haven't sent any requests

Any request you send in this workspace will appear here.

 Show me how

Good morning!

Let's start the day off right. Use Launchpad to start something new, pick up where you left off, or explore some resources to help you master Postman.

Start something new

- GET Create a request
- Create a collection
- Create an environment
- Create an API
- View More

Customize

- Dark mode
- Enable Launchpad
- More settings

What's new with Postman

Resources

Events

Last Chance to Save 50% on Premium Workshop Tickets for Postman Galaxy

Early bird pricing for Pathways tickets to Postman Galaxy—the global, virtual API conference—ends on December 31. The Pathways ticket gives you unique access to the exclusive, only-at-Postman-Galaxy hands-on workshops.

Register Now

Postman Public Workspaces: The First Massively Multiplayer API Experience

With public workspaces, API collaboration moves for the first time beyond the realm of a team. Now, people from different teams, different companies—virtually anyone, really—can work together to build software in a way that was not possible...

[Read the blog post](#)

Discover

Explore some templates and public APIs you might find useful.

Templates

APIs

Intro to writing tests - with examples

This collection contains examples of tests that you can use to automate your testing process. Includes basic test syntax, examples of API tests, and integration tests.

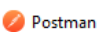
COVID19 API

Corona Virus stats REST API documentation

Working with GraphQL

Examples of working with GraphQL endpoints such as using a JSON request body, content-type, and importing queries.

[Browse More](#)



File Edit View Help

+ New Import Runner

My Workspace Invite

Filter

History Collections APIs

Save Responses

You haven't sent any requests

Any request you send in this workspace will appear here.

Show me how

Launchpad

POST Untitled Request

Untitled Request

POST

Enter request URL

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Headers

Hide auto-generated headers

KEY	VALUE
<input checked="" type="checkbox"/> Postman-Token ⓘ	<calculated when request is sent>
<input checked="" type="checkbox"/> Content-Length ⓘ	0
<input checked="" type="checkbox"/> Host ⓘ	<calculated when request is sent>
<input checked="" type="checkbox"/> User-Agent ⓘ	PostmanRuntime/7.26.8
<input checked="" type="checkbox"/> Accept ⓘ	*/*
<input checked="" type="checkbox"/> Accept-Encoding ⓘ	gzip, deflate, br
<input checked="" type="checkbox"/> Connection ⓘ	keep-alive
Key	Value

Response

Spring

Prima di poterlo utilizzare dobbiamo andare ad inserire un'impostazione che di default non è presente : impostazione del content-type

Content-Type

application/json

NewImportRunner

My Workspace

Invite

Filter

HistoryCollectionsAPIs

Save Responses

You haven't sent any requests

Any request you send in this workspace will appear here.

Show me how

LaunchpadPOST Untitled Request

Untitled Request

POSTEnter request URL

ParamsAuthorizationHeaders (8)BodyPre-request ScriptTestsSettings

HeadersHide auto-generated headers

KEY	VALUE
<input checked="" type="checkbox"/> Postman-Token ⓘ	<calculated when request is sent>
<input checked="" type="checkbox"/> Content-Length ⓘ	0
<input checked="" type="checkbox"/> Host ⓘ	<calculated when request is sent>
<input checked="" type="checkbox"/> User-Agent ⓘ	PostmanRuntime/7.26.8
<input checked="" type="checkbox"/> Accept ⓘ	*/*
<input checked="" type="checkbox"/> Accept-Encoding ⓘ	gzip, deflate, br
<input checked="" type="checkbox"/> Connection ⓘ	keep-alive
<input checked="" type="checkbox"/> Content-Type ⓘ	application/json
Key	Value

Response

Launchpad GET localhost:8080/listapersona No Environment

Untitled Request BUILD

GET localhost:8080/listapersona Send Save

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies Code

Headers Hide auto-generated headers

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Postman-Token ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/> Host ⓘ	<calculated when request is sent>	
<input checked="" type="checkbox"/> User-Agent ⓘ	PostmanRuntime/7.26.8	
<input checked="" type="checkbox"/> Accept ⓘ	*/*	
<input checked="" type="checkbox"/> Accept-Encoding ⓘ	gzip, deflate, br	
<input checked="" type="checkbox"/> Connection ⓘ	keep-alive	
<input checked="" type="checkbox"/> Content-Type	application/json	
Key	Value	Description

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "nome": "Mario",
4     "cognome": "Rossi"
5   },
6   {
7     "nome": "Luigi",
8     "cognome": "Bianchi"
9   }
10 ]
```

Status: 200 OK Time: 4 ms Size: 237 B Save Response

Spring

Launchpad

POST localhost:8080/inseriscipersona

Untitled Request

POST localhost:8080/inseriscipersona

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "nome": "Lucia",  
3   "cognome": "Verdi"  
4 }
```

The screenshot shows a REST client interface. At the top, there's a 'Launchpad' button and a request configuration bar showing 'POST localhost:8080/inseriscipersona'. Below this is the 'Untitled Request' section. The request method is 'POST' and the URL is 'localhost:8080/inseriscipersona'. The 'Body' tab is selected, and the 'raw' radio button is chosen for the body format. The body content is a JSON object: { "nome": "Lucia", "cognome": "Verdi" }. Red arrows highlight the flow from the top bar to the specific configuration elements: the method, the URL, the 'Body' tab, and the 'raw' body format.

PUT

Per quanto riguarda il verbo PUT, valgono le stesse regole viste per il verbo POST, quello che cambia è solo l'annotazione sul metodo nel controller, invece di usare l'annotazione `@PostMapping` su userà **`@PutMapping`**.

Ricordiamo che il metodo PUT a differenza del POST, aggiorna i dati già esistenti, quindi quello che cambia è la parte del Repository che vedremo più avanti.

DELETE

Per quanto riguarda il verbo DELETE, valgono le stesse regole viste per il verbo POST, quello che cambia è solo l'annotazione sul metodo nel controller, invece di usare l'annotazione `@PostMapping` su userà **`@DeleteMapping`**.

Ricordiamo che il metodo DELETE a differenza del POST, elimina i dati, quindi quello che cambia è la parte del Repository che vedremo più avanti.

RequestMapping

In alternativa all'utilizzo delle singole annotazioni, possiamo utilizzare l'annotazione RequestMapping in questo modo :

@RequestMapping(value = "/inseriscipersona", method = RequestMethod.POST)

È la stessa cosa!

```
//@PostMapping("/inseriscipersona")
@RequestMapping(value = "/inseriscipersona", method = RequestMethod.POST)
public Persona inserisciPersona(@RequestBody Persona p) {

    return ser.inserisciPersonaService(p);
}
```


Esercizio

1. Scaricare ed Installare Spring Tool Suite
2. Configurare progetto Spring «demo»
3. Configurare progetto Spring «demo-mvc»
4. Modificare «demo-mvc» per esporre servizi
5. Scaricare ed installare Postman
6. Testare l'applicazione «demo-mvc» con Postman
7. Ripassare gli argomenti trattati