



# Spring

## introduzione





# Introduzione

- Spring framework Fornisce un ambiente flessibile e completamente configurabile.
- Diversi progetti Java che sfruttavano il framework Spring avevano una configurazione dell'infrastruttura molto simile.
- Si stavano creando delle vere e proprie convenzioni per quanto riguarda la configurazione di un'applicazione Spring.
- Da qui nasce Spring Boot.



# Spring boot

## introduzione





# Introduzione

- Spring Boot è un progetto open source che fa parte del framework Spring.
- Sviluppato per semplificare la creazione di applicazioni stand-alone, production-grade basate sul framework Spring.
- Stand-alone: Applicazione che non deve essere «distribuita» su un web server. Il web server è incorporato.
- Production-grade: Applicazioni robuste che possono essere mandate in produzione.





# Introduzione

- Spring Boot si basa sull'approccio «Convention over configuration».
  - Se l'applicazione segue le convenzioni stabilite, non ci sarà la necessità di ulteriori file di configurazione.
  - Saranno necessari file di configurazione se il comportamento dell'app si discosta dalla convenzione.
- Tale concetto è usato nei framework per:
  - Ridurre il numero di decisioni che uno sviluppatore deve prendere.
  - Mantenere flessibilità



# Vantaggi di Spring Boot

- Riduce il tempo necessario per lo sviluppo e incrementa la produttività:
- Consente agli sviluppatori di concentrarsi sullo sviluppo della business logic dell'applicazione, invece di occupare troppo tempo per configurare l'infrastruttura dell'applicazione.
- offre un approccio più semplice e rapido per impostare, configurare ed eseguire le app.
- Elimina la complessità della configurazione necessaria per app basate su Spring.





# Vantaggi di Spring Boot

- Fornisce delle funzionalità «non funzionali» molto comuni alle applicazioni:
  - Server incorporati, sicurezza, metriche, controllo salute dell'applicazione e configurazioni esternalizzate .
- Riduce la necessità di scrivere codice «boilerplate» e configurazioni XML:
  - Non dobbiamo generare codice che si ripete.
  - Non è necessario aggiungere delle configurazioni XML.





# Vantaggi di Spring Boot

- Packaging semplificato:
  - Tutto quello di cui l'app ha bisogno (codice e dipendenze) è presente dentro un JAR che può essere mandato in esecuzione direttamente con il comando «java -jar <name.jar>»
  - Non più tanti JAR singoli per mandare in produzione un'applicazione in sicurezza.





# Vantaggi di Spring Boot

- Integrazione delle app Spring Boot con altri progetti Spring framework:
  - Le applicazioni Spring boot si integrano perfettamente con altri progetti dell'ecosistema Spring framework come Spring Data, Spring Security, Spring Cloud ecc..





# Vantaggi di Spring Boot

- Gestione delle dipendenze semplificato:
  - In passato, per sviluppare un'applicazione Spring framework gli sviluppatori occupavano tanto tempo nella gestione delle dipendenze(tanti JAR) e delle versioni di esse.
  - Adesso possiamo usare gli «Starters»





# Configurazione automatica

- Spring Boot è composto da un insieme di macro librerie che consentono di automatizzare la fase di configurazione dell'app.
- Queste macro librerie si chiamano «starters».
- Ogni starter che usiamo nel nostro progetto contiene un insieme di JAR.
- Gli starter sono opinionati, cioè si basano sull'opinione dei creatori di Spring di ciò che pensano possa servire al tuo progetto.





# Configurazione automatica

- Esempio, se inserisco lo starter-data-jpa nel progetto, allora significa che nel classpath verranno configurati N JAR che fanno parte dello starter-data-jpa. Tra cui hibernate, transactions e molti altri JAR.
- Spring boot capirà quali bean creare e come configurarli tramite lo scanning dei JAR presenti nel classpath.
- La configurazione del nostro progetto Spring Boot dipende da quali starters abbiamo scelto come dipendenze.



# Come utilizzare Spring Boot



- Possiamo creare un progetto Spring Boot tramite:
  - Applicativo web chiamato Spring Initializr.
  - IDE.
  - Command line interface di Spring Boot.
  - Tool di build esterni come ad esempio Maven o Gradle



# Spring Initializr



- Sono dei web service REST con un'interfaccia grafica che si occupano di generare il progetto in base alle nostre specifiche.
- Interfaccia grafica raggiungibile all'indirizzo: <https://start.spring.io/>
- Dobbiamo fornire le specifiche del nostro progetto:
  - Tipo di progetto(Maven o Gradle)
  - Linguaggio(Java o altri)
  - Versione di Spring Boot
  - Project Metadata(groupid, artifact, packaging ecc..)
  - Dependencies(Selezionare gli starters di cui abbiamo bisogno)

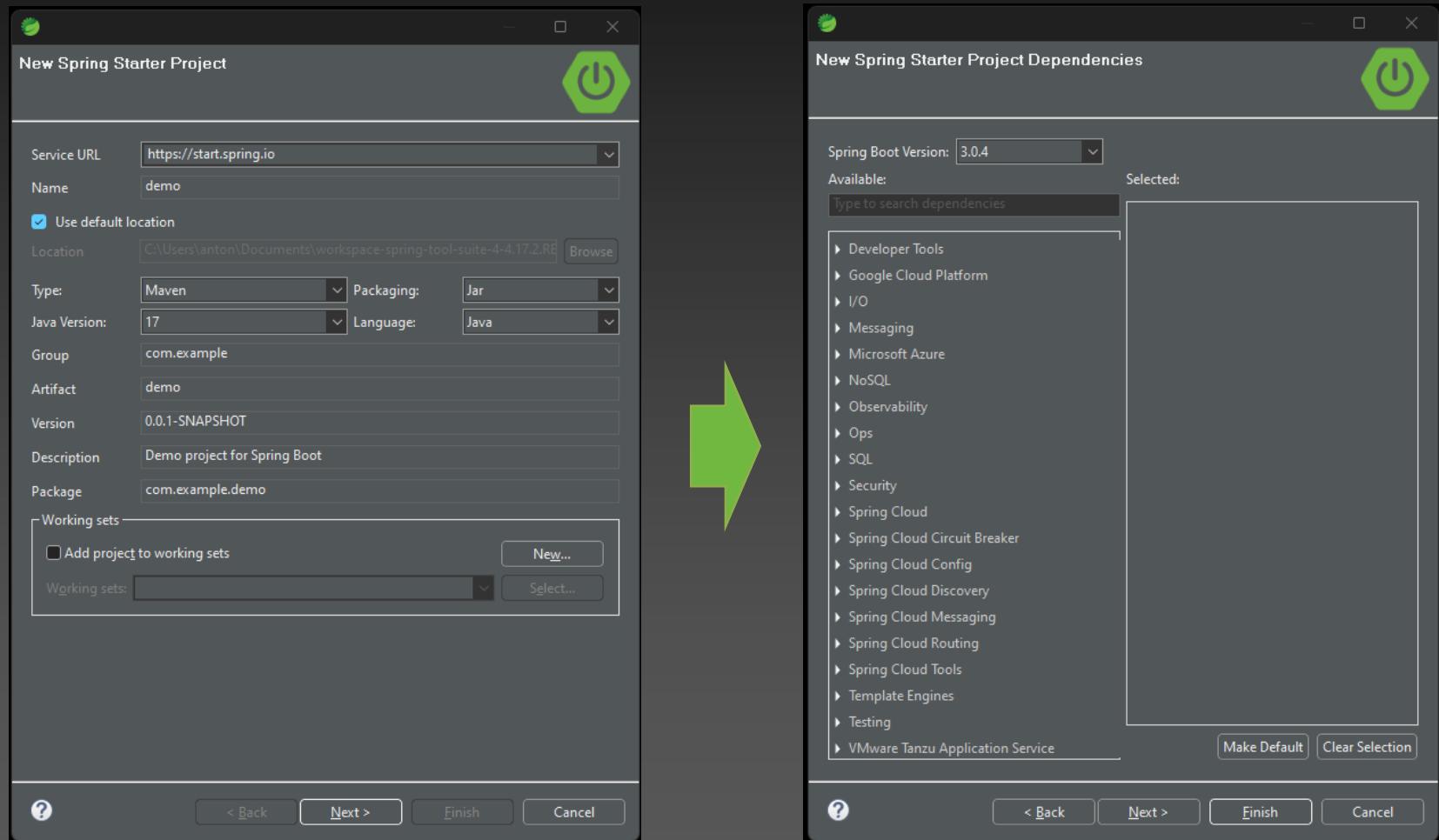


# Spring Initializr

The screenshot shows the Spring Initializr web application. It has a dark theme with green highlights. On the left, there are sections for Project (Gradle - Groovy selected), Language (Java selected), and Spring Boot (3.0.4 selected). In the center, there's a 'Project Metadata' section with fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), Package name (com.example.demo), Packaging (Jar selected), Java version (17 selected), and Java minor versions (19, 11, 8). On the right, there's a 'Dependencies' section with a button to 'ADD DEPENDENCIES... CTRL + B'. At the bottom, there are three buttons: 'GENERATE CTRL + D', 'EXPLORE CTRL + SPACE', and 'SHARE...'.



# IDE (Spring tool suite)



# Starters



## ➤ spring-boot-starter-parent:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>4.0.0</version>
  <relativePath/>
</parent>
```

- Questo starter serve per specificare la versione di Spring Boot.
- Specificando la versione nel tag <parent> stiamo comunicando al pom «padre» che tutti gli starter che verrano inseriti nel pom «figlio», ovvero nel nostro pom, saranno della stessa versione specificata per il parent.



# Starters



## ➤ spring-boot-starter:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
```

- Questo starter contiene tutti i JAR necessari per creare un progetto Spring boot.
- Contiene i JAR spring-core, spring-context ecc.. Tutti JAR che in passato avremmo dovuto inserire singolarmente con applicazioni basate su Spring framework.
- Altri starters come «starter-web» o «starter-data-jpa» includono lo starter «spring-boot-starter» al loro interno. Per questo motivo spesso viene omesso.



# Starters

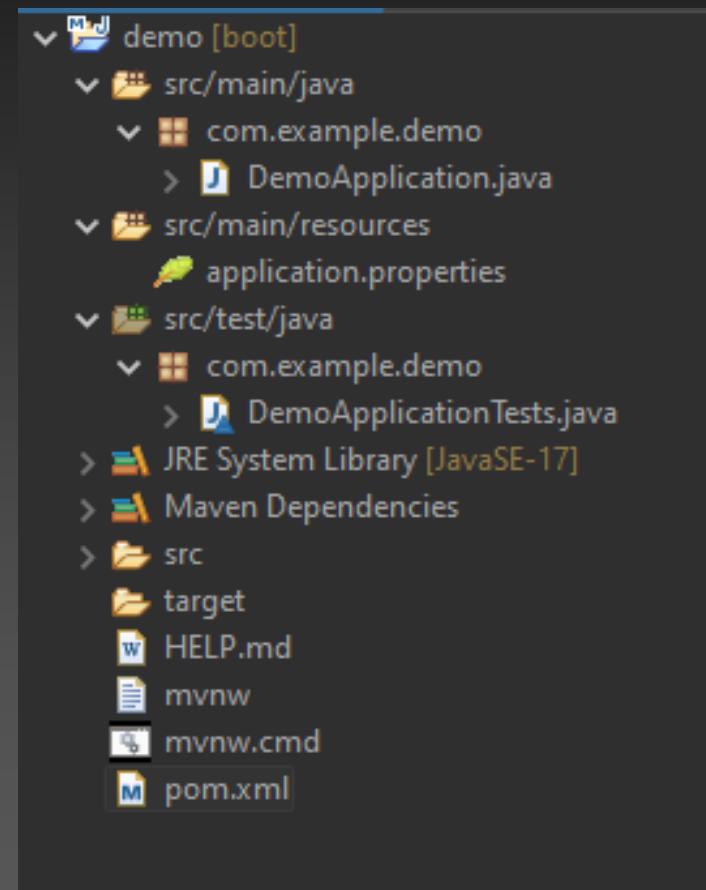


- Esistono 50+ starters che possiamo utilizzare per creare un progetto Spring Boot.
- Scelgo gli starters in base alle tecnologie e funzionalità che la mia applicazione deve avere.
- Altri:
  - spring-boot-starter-actuator
  - spring-boot-starter-web
  - spring-boot-starter-data-jpa
  - spring-boot-starter-security
  - ecc...





# Struttura progetto





# Struttura progetto

```
└── src/main/java
    └── com.example.demo
        └── DemoApplication.java
```

- All'interno della cartella «src/main/java» ci saranno tutti i package e sottopackage dell'applicazione.
- All'interno del package che viene creato in automatico vanno creati manualmente ulteriori package che conterranno i componenti dell'applicazione: classi di configurazione, entities, services, controllers ecc...





# Struttura progetto

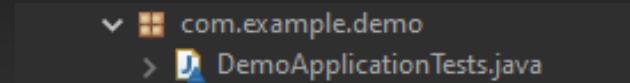
```
@SpringBootApplication  
public class DemoApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
  
}
```

- L'annotazione @SpringBootApplication incapsula le seguenti annotazioni:
- @Configuration -> Definire @Bean all'interno della classe annotata.
- @ComponentScan -> Creazione dei bean annotati con le annotazione stereotype presenti nel package e sotto package.
- @EnableAutoConfiguration -> Abilita l'auto configurazione di Spring Boot. Crea e configura ulteriori beans in base ai JAR presenti nel classpath.





# Struttura progetto



- All'interno della cartella «src/test/java» ci saranno tutti i package e sottopackage che conterranno le classi di test.
- Tali classi servono a testare i componenti dell'applicazione.





# Struttura progetto

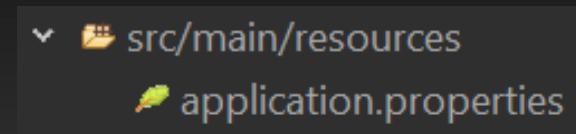
```
@SpringBootTest  
public class DemoApplicationTests {  
  
    void contextLoads() {  
  
    }  
  
}
```

- L'annotazione `@SpringBootTest` permette la creazione di un `ApplicationContext` che contiene i bean della nostra applicazione.
- Tramite l'annotazione `@Autowired` possiamo iniettare tali bean e testarli all'interno dell'ambiente di testing.
- La fase di testing è molto importante. Ci sono diverse metodologie, best practice e tecnologie che possiamo utilizzare.





# Struttura progetto



- La cartella «src/main/resources» contiene le risorse dell'app.
- Di default viene generato un file chiamato «application.properties».
- Esiste anche l'estensione «application.yml»
- In questi file possiamo configurare Spring Boot.
- Spring Boot fornisce varie proprietà che possono essere configurate:
  - Server properties
  - Data properties
  - Mail properties
  - Security properties
  - Logging properties
  - ecc..





# Spring Boot

## Spring Web RESTful





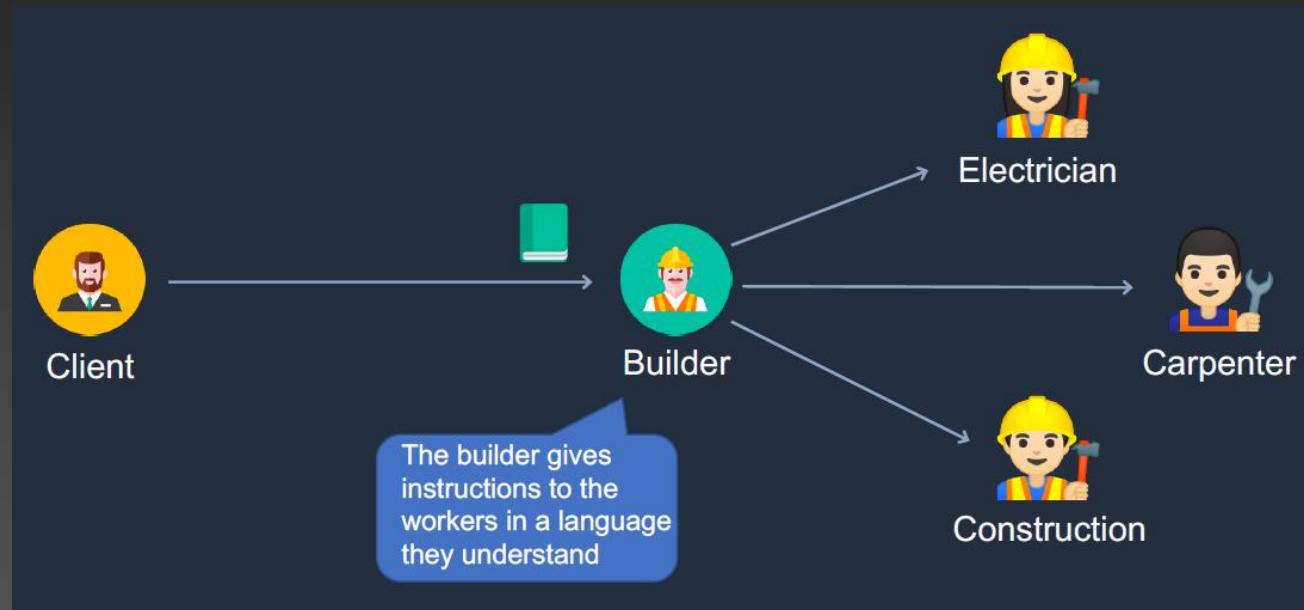
# Cos'è un'API

- API è l'acronimo Application Programming Interface, ovvero un insieme di definizioni e protocolli per la creazione e l'integrazione di software applicativi.
- Meccanismi che consentono a due componenti software di comunicare tra loro:
  - Client - Server
  - Server – Server
- Diversi tipologie di API:
  - RESTful
  - SOAP
  - RPC
  - Websocket





# Cos'è un'API





# Cos'è un'API RESTful

- Un'API REST è un'API conforme ai principi di progettazione dello stile architettonale REST.
- REST «REpresentational State Transfer» è uno stile architettonale per sistemi distribuiti.
- I servizi web REST rappresentano un approccio differente ai servizi web rispetto a quello offerto dai servizi basati su SOAP, RPC.
- Usa il protocollo HTTP per eseguire chiamate tra i componenti SW.
- Al giorno d'oggi le architetture a microservizi web REST sono molto comuni.
- Roy Fielding è il creatore:
- [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)





# Proprietà architettura REST

- Client - Server:
- Client e server sono due applicazioni separate.
- In determinati contesti un server ha il ruolo di «client», e un altro server ha il ruolo «server».
- Possono evolvere indipendentemente.
- Al server non interessa con quali tecnologie e come è stato implementato il client. E viceversa.





# Proprietà architettura REST

- Stateless:
  - Comunicazione tra client e server è stateless.
  - Il server non memorizza nessuna informazione di sessione che riguarda il client.
  - Tutte le informazioni di cui il server ha bisogno per elaborare una risposta devono essere contenute nella richiesta.
  - Migliora la scalabilità: non esistono comunicazioni interne al server per la gestione della sessione. I componenti sono indipendenti, quindi possono scalare facilmente.
  - Performance di rete peggiori: richieste di grandi dimensioni





# Proprietà architettura REST

- Caching lato client:
  - Allo scopo di non fare sempre la stessa richiesta, il client può memorizzare la risposta dal server nella sua cache.
  - Questo è consentito solo se il server applica una configurazione all'header «cache-control» della risposta che vuole far memorizzare nel client.
  - Migliora le performance.
  - Lo svantaggio è che il client potrebbe utilizzare dei dati «obsoleti».





# Proprietà architettura REST

- Uniform interface:
  - E' ciò che differenzia le API REST dalle altre.
  - L'interazione con un'API REST è identica per diverse tipologie di applicazioni client(smartphone, browsers, servizi ecc.)
  - Per raggiungere questo obiettivo ci sono quattro vincoli:
    - Identificazione delle risorse: es: «/user/23» è l'URI di una risorsa
    - Manipolazione delle risorse tramite rappresentazioni: JSON ecc..
    - Richieste autodescrittive: ogni richiesta deve contenere il necessario
    - HATEOS(Hypermedia As The Engine Of application State):
    - URI di altre risorse nella risposta da inviare al client. Il client scopre risorse utili(spring-boot-starter-HATEOS)





# Proprietà architettura REST

- Architettura a livelli:
  - Architettura composta da diversi livelli. Ogni livello ha la propria responsabilità(es: MVC).
  - Ogni livello non deve conoscere come sono implementati gli altri.
  - Migliora sicurezza: ogni layer è disaccoppiato dagli altri.
  - Peggiora latenza: la richiesta deve passare per tutti i livelli.





# Proprietà architettura REST

- Codice on demand(opzionale):
  - Il server può inviare al client delle riposte che contengono codice eseguibile (es: Javascript ecc..)
  - Potenziali problemi di sicurezza.





# Risorsa

- Una risorsa è una vera e propria informazione che il server può fornire (es: un prodotto, una collezione di prodotti ecc..)
- Le risorse sono identificate univocamente tramite un URL
  - <http://localhost:8080/myapp/resources/users> --> Collezione di risorse
  - <http://localhost:8080/myapp/resources/users/23> --> Singola risorsa
- I metodi HTTP sono usati per eseguire determinate operazioni sulle risorse: es. leggere, aggiungere, modificare, eliminare una risorsa.





# Risorsa

Method	Purpose
GET	Read, possibly cached
POST	Update or create without a know ID
PUT	Update or create with a know ID
DELETE	Remove
HEAD	Read headers, has version changed?
OPTION	List the "Allow"ed methods





# Headers HTTP

GET /home.html HTTP/1.1  
Host: developer.mozilla.org  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0)  
Gecko/20100101 Firefox/50.0  
Accept:  
text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate, br  
Referer: https://developer.mozilla.org/testpage.html  
Connection: keep-alive  
Upgrade-Insecure-Requests: 1  
If-Modified-Since: Mon, 18 Jul 2016 02:36:04 GMT  
If-None-Match: "c561c68d0ba92bbeb8b0fff2a9199f722e3a621a"  
Cache-Control: max-age=0

## Esempio di response headers

200 OK  
Access-Control-Allow-Origin: \*  
Connection: Keep-Alive  
Content-Encoding: gzip  
Content-Type: text/html; charset=utf-8  
Date: Mon, 18 Jul 2016 16:06:00 GMT  
Etag: "c561c68d0ba92bbeb8b0f612a9199f722e3a621a"  
Keep-Alive: timeout=5, max=997  
Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT  
Server: Apache  
Set-Cookie: mykey=myvalue; expires=Mon, 17-Jul-2017 16:06:00 GMT;  
Max-Age=31449600; Path=/; secure  
Transfer-Encoding: chunked  
Vary: Cookie, Accept-Encoding  
X-Backend-Server: developer2.webapp.scl3.mozilla.com  
X-Cache-Info: not cacheable; meta data too large  
X-kuma-revision: 1085259  
x-frame-options: DENY



# Rappresentazioni delle Risorse (HTTP Body)



- I componenti REST eseguono azioni su una risorsa usando una rappresentazione per acquisire lo stato corrente o previsto di tale risorsa e trasferendo tale rappresentazione tra i componenti.
- Esempio: per modificare un prodotto(la risorsa) il client deve fornire al server una rappresentazione prevista della risorsa(prodotto modificato).
- Le rappresentazioni supportate sono:
  - JSON
  - XML
  - HTML
  - altri
- HTTP Headers «accept» e «content-type» usati per negoziare il tipo di rappresentazione.





# Rappresentazione JSON

- La più comune al giorno d'oggi.
- JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati tra componenti SW.
- È un insieme di coppie chiave-valore
- Vantaggi:
  - Per le persone facile da leggere e scrivere
  - Per le macchine facile da generare e analizzare
  - Indipendente dal linguaggio di programmazione





# Rappresentazione JSON

```
JSON Object → {  
    "company": "mycompany",  
    "String Value  
    "companycontacts": { ← Object Inside Object  
        "phone": "123-123-1234",  
        "email": "myemail@domain.com"  
    },  
    "employees": [ ← JSON Array  
        {  
            "id": 101,  
            "name": "John",  
            "contacts": [  
                "email1@employee1.com",  
                "email2@employee1.com"  
            ]  
        },  
        {  
            "id": 102, ← Number Value  
            "name": "William",  
            "contacts": null ← Null Value  
        }  
    ]  
}
```





# Status code

- Lo status code di una risposta HTTP indica al client se una specifica richiesta HTTP è stata completata correttamente.
  - 1xx - risposte Informative:
    - Abbiamo ricevuto la tua richiesta e la stiamo processando
  - 2xx - risposte di successo:
    - Abbiamo ricevuto con successo la richiesta, ecco la risposta.
  - 3xx - risposte di reindirizzamento:
    - Serve un'ulteriore azione per soddisfare la richiesta.
  - 4xx - risposte errore lato client:
    - La richiesta contiene errore di sintassi o non può essere soddisfatta.
  - 5xx - risposte errore lato server:
    - Il server non è riuscito a soddisfare una richiesta apparentemente valida.



# Spring Boot e Spring Cloud



## Spring Boot

### BUILD ANYTHING

Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production-ready applications.

## Spring Cloud

### COORDINATE ANYTHING

Built directly on Spring Boot's innovative approach to enterprise Java, Spring Cloud simplifies distributed, microservice-style architecture by implementing proven patterns to bring resilience, reliability, and coordination to your microservices.

- Spring Boot: Sviluppare i microservizi.
- Spring Cloud: Coordinare i microservizi.



# Spring Boot REST



- Starter necessario per sviluppare un microservizio REST in Spring Boot

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
```



# Spring Boot REST



- Per creare un'API REST dobbiamo creare un classe annotata con `@RestController`.
- L'URI della risorsa è specificato con l'annotazione `@RequestMapping`
- Tale annotazione si può utilizzare sia sulla definizione della classe che sulla definizione del metodo.
- I metodi della classe `@RestController` si chiamano endpoint.

```
@RestController  
@RequestMapping(value = "/api/v1")  
public class HelloController{  
  
    @GetMapping(value = "/hello")  
    public String hello() {  
        return "hello";  
    }  
}
```

→ **URI: /api/v1/hello**  
**URL: http://host:porta/api/v1/hello**



# Spring Boot REST



- Per testare le API possiamo usare:
  - Classi di testing di Spring Boot
  - Browser
  - Postman
  - altri

The screenshot shows a Postman interface. At the top, it says "GET" and "http://localhost:8080/api/v1/hello". Below that, under "Params", there is a table with one row:

KEY	VALUE	DESCRIPTION
Key	Value	Description

At the bottom, the response section shows a status of "200 OK" and the body content "1 hello".





# Spring Boot

## Input/Output





# Parametri in input

- Per eseguire delle operazioni, un web service necessita dei dati input.
- Oltre alle rappresentazioni di risorse è possibile anche passare dei parametri nella richiesta ad una risorsa.
- Tipicamente nelle operazioni di lettura.
- Due tipi:
  - URI path parameters
  - Query parameters
- Entrambi estraggono dati dall'URI





# URI path parameters

- @PathVariable

```
@RestController  
@RequestMapping(value = "/api/v1")  
public class HelloController{  
  
    @GetMapping("/hello/{nome}")  
    public String hello(@PathVariable("nome") String nome){  
        return "hello "+nome;  
    }  
}
```

A screenshot of the Postman application interface. At the top, it shows a GET request to the URL `http://localhost:8080/api/v1/hello/utente`. Below the URL, there are tabs for Params, Authorization, Headers (8), Body, and Pre-request Script. The Body tab is selected and contains sub-tabs for Body, Cookies, Headers (5), and Test Results. Under the Body tab, there are buttons for Pretty, Raw, Preview, Visualize, and Text. The Text button is currently selected, showing the response body: `1 hello utente`.



# Query parameters

- @RequestParam

```
@RestController  
@RequestMapping(value = "/api/v1")  
public class HelloController{  
  
    @GetMapping("/hello")  
    public String hello(@RequestParam("nome") String nome){  
        return "hello "+nome;  
    }  
}
```

The screenshot shows the Postman application interface. At the top, it says "GET" and the URL "http://localhost:8080/api/v1/hello?nome=utente". Below this, there are tabs for "Params", "Authorization", "Headers (6)", "Body", "Pre-request Script", "Tests", and "Settings". The "Params" tab is currently active. It displays a table titled "Query Params" with one row:

KEY	VALUE
nome	utente

Below the table, there are tabs for "Body", "Cookies", "Headers (5)", and "Test Results". The "Body" tab is active and contains a "Pretty" button, "Raw" button, "Preview" button, "Visualize" button, and a "Text" dropdown set to "Text". Under the "Text" dropdown, the response body is shown as "1 hello utente".





# Annotazioni metodi HTTP

HTTP Method	@RequestMapping	Spring 4.3 Annotation
GET	@RequestMapping(method=RequestMethod.GET)	@GetMapping
POST	@RequestMapping(method=RequestMethod.POST)	@PostMapping
DELETE	@RequestMapping(method=RequestMethod.DELETE)	@DeleteMapping
PUT	@RequestMapping(method=RequestMethod.PUT)	@PutMapping

- GET: lettura risorsa
- POST: creazione risorsa
- PUT: modifica risorsa
- DELETE: eliminazione risorsa





# Annotazioni metodi HTTP

- Tutte le annotazioni dei metodi HTTP sono internamente annotate con @RequestMapping.
- Tale annotazione contiene degli attributi che si possono usare anche sulle nuove annotazioni.
  - value, path
  - name
  - headers
  - produces
  - consumes

```
@PostMapping(value = "/automobile/aggiungi",
    consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
```





# Conversione Rappresentazioni

- La conversione da oggetto JAVA a JSON e viceversa, viene svolta in automatico.
- Il bean che ha il compito di svolgere le conversioni implementa l'interfaccia `HttpMessageConverter<T>`.
- Esistono diverse implementazioni concrete di tale interfaccia. Ognuna pensata per un tipo di conversione diversa.
- Di default questo bean converte gli oggetti JAVA in JSON e viceversa. Ma è possibile riconfigurarla per aggiungere altre tipologie di conversione.(es: XML ecc..)





# Conversione in JSON

```
@GetMapping("/automobile")
public Automobile getAutomobile(){
    return new Automobile(1,"fiat","panda",400000,25, 2,1999);
}
```

```
{
  "id": 1,
  "marca": "fiat",
  "modello": "panda",
  "chilometri": 400000,
  "immatricolazione": "1999-02-25"
}
```





# Conversione Rappresentazioni

- `@RequestBody` comunica a spring di effettuare la conversione. Il tipo di conversione da applicare dipende dall'header «content-type» della richiesta HTTP.

The screenshot shows a Postman request to `http://localhost:8080/api/v1/automobile/aggiungi`. The request method is `POST`. The `Body` tab is selected, showing a JSON payload:

```
1
2   "marca": "citroen",
3   "modello": "c3",
4   "chilometri": 20000,
5   "immatricolazione": "2021-12-04"
```

The response status is `200 OK` with a response time of `10 ms` and a size of `123 B`. The response body is:

```
Automobile
[id=0,
 marca=citroen,
 modello=c3,
 chilometri=20000,
 immatricolazione=2021-12-04]
```

```
@PostMapping("/automobile/aggiungi")
public void addAutomobile(@RequestBody Automobile auto){
    System.out.println(auto);
}
```





# Spring Boot

## Spring Data JPA





# Spring Boot Data-JPA

- Spring Boot data effettuerà la configurazione automatica del javax.sql.DataSource se nel classpath riesce a trovare un driver supportato. (es: Connector MySQL, H2 ecc..).
- Lo starter data-jpa prende il DataSource configurato in base al driver trovato nel classpath e lo usa per la configurazione JPA.
- Dobbiamo fornire la configurazione del DataSource attraverso delle proprietà nell'application.properties (o .yml).

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```





# Entities

- Tutte le classi che mappano le tabelle del database vanno annotate con `@Entity`
- Con questa annotazione andiamo a segnalare che la classe mappa una specifica tabella del database
- Ogni entities deve essere una classe POJO (plain old java object) ovvero una classe:
  - non final
  - non abstract
  - con un costruttore vuoto
  - variabili private
  - metodi getter e setter



# Entities



```
@Entity  
public class Ordine {  
    <<parametri>>  
    <<Costruttore vuoto>>  
    <<Getter e Setter>>  
}
```





# Entities

- Tutte le Entities devono avere una primary key mappata con @id
- Possiamo aggiungere la tipologia di generazione della primary key tramite l'annotazione @GeneratedValue(strategy = GenerationType.{type})
- I tipi sono:
  - AUTO
    - Indica che il provider di persistenza deve scegliere una strategia appropriata per il database specifico
  - IDENTITY
    - Indica che il provider di persistenza deve assegnare chiavi primarie per l'entità utilizzando una colonna di identità del database
  - SEQUENCE
    - Indica che il provider di persistenza deve assegnare chiavi primarie per l'entità utilizzando una sequenza di database
  - TABLE
    - Indica che il provider di persistenza deve assegnare chiavi primarie per l'entità utilizzando una tabella del database sottostante per garantire l'univocità



# Entities



```
@Entity  
public class Ordine {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private long id;  
    <<Costruttore vuoto>>  
    <<Getter e Setter>>  
}
```





# Entities

- Tutte le altre colonne possono essere annotate con @Column questo ci permette di definire dei parametri come:
  - updatable
  - unique
  - nullable
  - insertable
  - name



# Entities



```
@Entity
public class Ordine {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column(updatable = true, unique = true,
            nullable = true, insertable = true,
            name = "nome_ordine")
    private String nomeOrdine;
    <<Costruttore vuoto>>
    <<Getter e Setter>>

}
```





# Entities

- Nel caso di relazione tra più tabelle abbiamo tre tipi di relazioni possibili
- @ManyToOne
- @OneToMany
- @ManyToMany





# ManyToOne

- Se la relazione è una @ManyToOne avremo all'interno della classe un attributo del tipo della classe con la quale esiste la relazione
- Inoltre per utilizzare i parametri della @Column l'annotation sarà @JoinColumn
- La classe con l'annotation @ManyToOne si definisce «proprietaria» della relazione ed è quella che conterrà la foreing-key sul database





# ManyToOne

```
@Entity  
public class Ordine {  
    <<altri parametri>>  
  
    @ManyToOne  
    @JoinColumn  
    private Utente cliente;  
    <<Costruttore vuoto>>  
    <<Getter e Setter>>  
}
```





# OneToMany

- Se la relazione è una @OneToMany avremo all'interno della classe un attributo del tipo Collection<T> dove T è la classe con la quale esiste la relazione
- La classe con la quale esiste la relazione DEVE avere un attributo del tipo della nostra classe mappato con @ManyToOne
- All'interno dell'annotation come parametro inseriremo mappedBy="nomeDellaVariabileDellAltraClasse"



# OneToMany



```
@Entity  
public class Utente {  
    <<altri parametri>>  
  
    @OneToMany (mappedBy = "cliente")  
    private List<Ordine> ordini;  
  
    <<Costruttore vuoto>>  
    <<Getter e Setter>>  
  
}
```





# ManyToMany

- Se la relazione è una @ManyToMany avremo all'interno della classe un attributo del tipo Collection<T> dove T è la classe con la quale esiste la relazione
- Una delle due classi deve diventare "proprietaria" della relazione mappando la tabella di relazione che verrà creata sul db con l'annotation @JoinTable nella quale relazioneremo entrambe le foreing-key
- Se vogliamo rendere la relazione bidirezionale l'altra tabella utilizzerà il mappedBy





# ManyToMany

```
@Entity
public class Ordine {
    <<altri parametri>>

    @ManyToMany
    @JoinTable(name = "prodotti_acquistati",
               joinColumns = @JoinColumn(name="id_ordine"),
               inverseJoinColumns = @JoinColumn(name="id_prodotto"))
    private List<Prodotto> prodotti;

    <<Costruttore vuoto>>
    <<Getter e Setter>>

}
```



# FetchType

- Una cosa che possiamo definire su tutte le tipologie di relazioni tra entità è il `fetchType`,
- Questo va a definire quando scaricare gli attributi dell'oggetto relazionato
- esistono solo due tipi di fetch:
  - `LAZY`
    - Scarica gli attributi solo se viene chiamato il getter dell'attributo
  - `EAGER`
    - Scarica gli attributi appena viene scaricato l'oggetto dal database





# CascadeType

- Un'ulteriore cosa che possiamo definire su tutte le tipologie di relazioni tra entità è il cascadeType,
- Questo va a definire come si comporterà con gli attributi relazionati quando effettueremo operazioni sull'oggetto selezionato
- esistono vari tipi di cascate che sono:
  - ALL
    - Propaga tutte le operazioni
  - PERSIST
    - Propaga solo le operazioni di persistenza
  - MERGE
    - Propaga le operazioni di «merging»
  - REMOVE
    - Propaga le operazioni di rimozione
  - REFRESH
    - propaga il ricaricamento dell'entità dal db ogni volta che verrà effettuato per l'entità "padre"
  - DETACH
    - Propaga anche agli attributi la rimozione dal persistence context





# ManyToMany

```
@Entity
public class Ordine {
    <<altri parametri>

    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinTable(name = "prodotti_acquistati",
               joinColumns = @JoinColumn(name="id_ordine"),
               inverseJoinColumns = @JoinColumn(name="id_prodotto"))
    private Set<Prodotto> prodotti;

    <<Costruttore vuoto>>
    <<Getter e Setter>>

}
```



# Spring Boot Data-JPA

- Spring Boot data effettuerà la configurazione automatica del jakarta.sql.DataSource se nel classpath riesce a trovare un driver supportato. (es: Connector MySQL, H2 ecc..).
- Lo starter data-jpa prende il DataSource configurato in base al driver trovato nel classpath e lo usa per la configurazione JPA.
- Dobbiamo fornire la configurazione del DataSource attraverso delle proprietà nell'application.properties (o .yml).





# Application.properties

- Una volta configurate tutte le nostre classi dovremo aggiungere i parametri all'interno dell'application properties
- I parametri da aggiungere sono
  - L'url del database
  - Lo username di connessione al db
  - La password di connessione al db
  - Il driver class name utilizzato per connettere il database
- Altri parametri che possiamo aggiungere sono
  - Se vogliamo visualizzare in console le query eseguite
  - La tipologia di ddl da eseguire sul db





# Application.properties

```
spring.datasource.url = jdbc:mysql://localhost:3306/ordini
```

```
spring.datasource.username = root  
spring.datasource.password = toor
```

```
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver
```

```
spring.jpa.show-sql = true  
spring.jpa.hibernate.ddl-auto = update
```



# Application.yml

```
spring:  
  datasource:  
    url: jdbc:mysql://localhost:3306/ordini  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    username: root  
    password: toor  
  
  jpa:  
    show-sql: true  
    hibernate:  
      ddl-auto: update
```

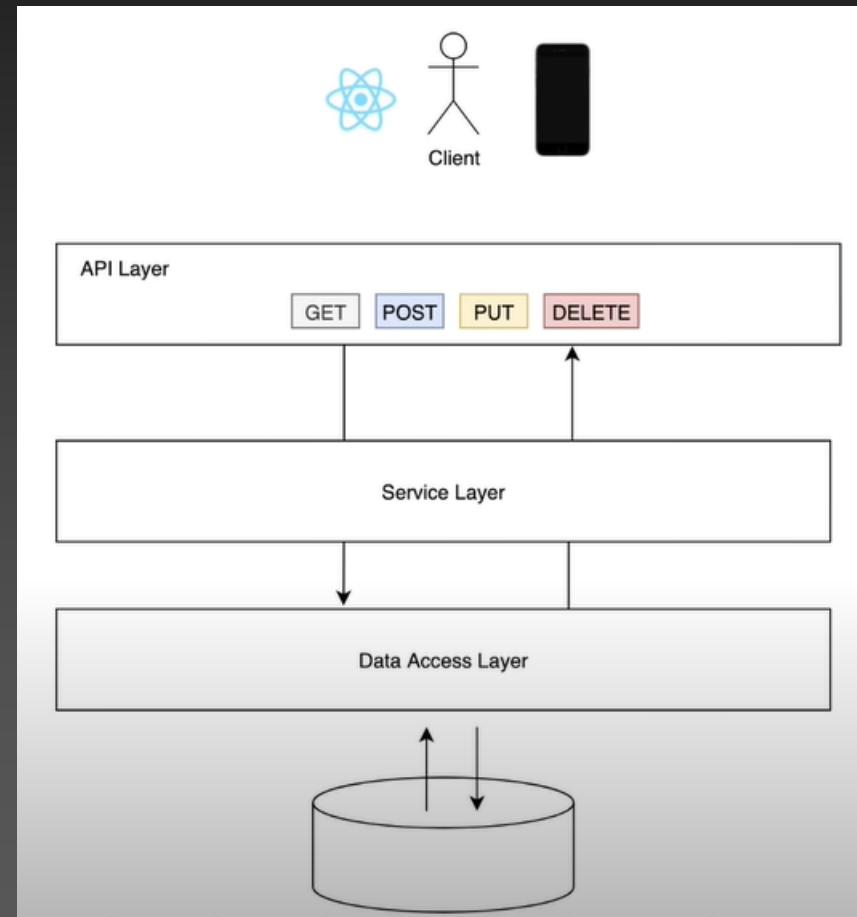


# Spring Boot

## Architettura a Layer



# Architettura a layer





# Repository layer

- Non serve annotare con @Repository la nostra interfaccia.
- L'interfaccia JpaRepository<T, ID> incapsula un'annotazione chiamata @NoRepositoryBean. Questa annotazione comunica a spring di non creare bean del tipo JpaRepository, Ma di creare i bean delle classi che IMPLEMENTANO la nostra interfaccia. Tali bean sono creati in fase di esecuzione.
- Responsabilità:
  - Interagire con la sorgente dati.
  - Un repository per ogni entity.





# Repository layer

```
public interface OrdineRepository extends JpaRepository<Ordine,Long> {  
}
```





# Service layer

- Annotazione @Service
- Responsabilità:
  - Contengono la business logic dell'applicazione.
- Caratteristiche:
  - Invocare metodi del repository.
  - Validazione.
  - Conversione da entity a DTO e viceversa (DTO pattern).





# Service layer

```
public interface OrdineService {  
    Ordine findById(long id);  
    List<Ordine> findAll();  
}
```

```
@Service  
public class OrdineServiceImpl implements OrdineService{  
    @Autowired  
    private OrdineRepository repo;  
  
    @Override  
    public Ordine findById(long id) {  
        return repo.findById(id).orElseThrow(RuntimeException::new);  
    }  
  
    @Override  
    public List<Ordine> findAll() {  
        return repo.findAll();  
    }  
}
```





# Controller layer

- Annotazione @RestController
- Responsabilità:
  - Accettare le richieste HTTP
  - Invocare la logica aziendale.
  - Rispondere in base a ciò che è stato processato.
  - Prima linea contro utenti non autorizzati ad accedere ai service.
  - Gestire eccezioni propagate dagli altri layer.





# Controller layer

```
@RestController
public class OrdineController {

    private final OrdineService service;

    public OrdineController(OrdineService service) {
        this.service=service;
    }

    @GetMapping("/ordine/{id}")
    public ResponseEntity<Ordine> findOrdine(@PathVariable("id") long id){
        Ordine o=service.findById(id);
        if(o==null) return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
        else return ResponseEntity.ok(o);
    }

    @GetMapping("/ordini")
    public ResponseEntity<List<Ordine>> listaOrdini(){
        return ResponseEntity.ok(service.findAll());
    }

}
```

spring  
boot