

Introduzione

spring



ISISLab

Tiziano Citro

Outline



Spring



Spring Boot



Spring Stereotype



Spring Web MVC



Spring Data

Outline



Spring Session



Spring Security



Spring Cloud



Thymeleaf



Tips e ambiente di sviluppo

Spring

Spring è considerato il più popolare tra i framework per lo sviluppo di applicazioni Java Enterprise.

Ha l'obiettivo di gestire la complessità relativa allo sviluppo di applicazioni enterprise in Java.

Fornisce tutto il necessario per utilizzare il linguaggio Java in un ambiente enterprise offrendo soluzioni ai problemi comuni allo sviluppo di questa tipologia di applicazioni.

Essendo il framework Java più popolare al mondo vanta una grande e attiva community che fornisce feedback sulla base di casi d'uso reali.

Architettura

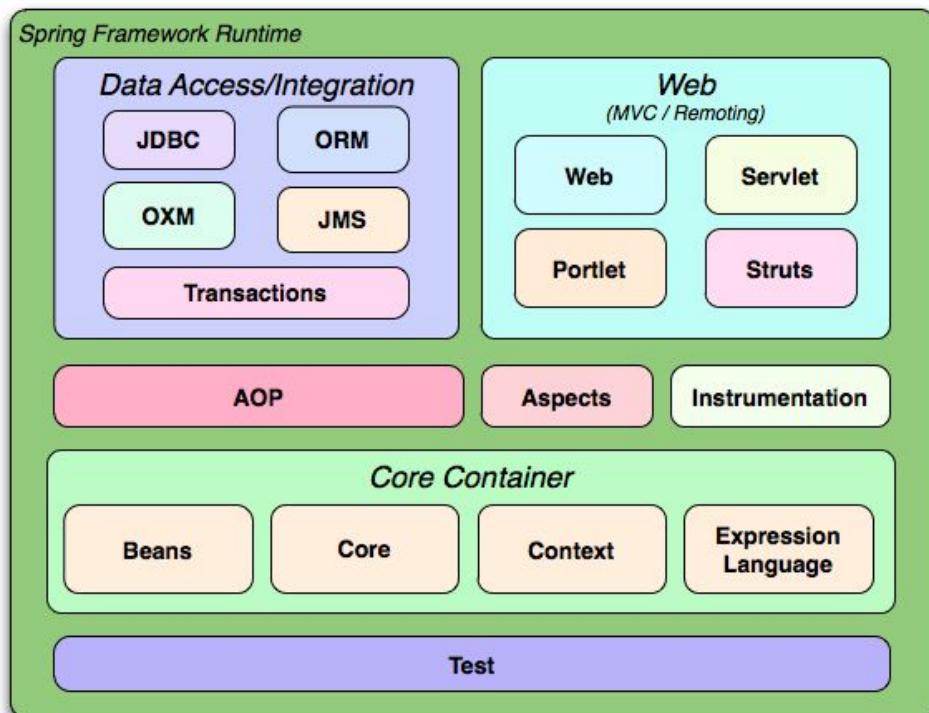
Il modulo **Core** fornisce le funzionalità fondamentali, come: Inversion of Control, Dependency injection, application context (come un registro JNDI), ecc.

Il modulo di **Data Access** fornisce integrazione con sorgenti dati (database, cache, ecc.), astraendo tecnologie come JDBC, JMS, ecc.

Il modulo **Web** fornisce le funzionalità necessarie alle applicazioni Web, come gestione richieste e risposte.

I moduli **AOP** e **Instrumentation** offrono funzionalità trasversali, come interceptor, logging, ecc.

Il modulo **Test** fornisce funzionalità di test integrate con gli altri moduli.



Struttura modulare

Il framework è progettato con una struttura modulare.

Il modulo core è sempre necessario mentre gli altri moduli possono essere introdotti se necessari all'applicazione.

Ci sono moduli (dipendenze) per diverse necessità:

- Spring Data JPA per l'interazione con database relazionali;
- Spring Data Mongo per l'interazione con MongoDB;
- Spring Security per la gestione della sicurezza;
- ecc.



Apache Maven è uno strumento di gestione dei progetti software.

Basato sul concetto di un Project Object Model (POM), Maven può gestire la build e documentazione di un progetto in maniera centralizzata.

Un POM è l'unità fondamentale di lavoro in Maven. È un file XML chiamato pom.xml che contiene informazioni sul progetto e dettagli di configurazione utilizzati da Maven per il processo di build.

Contiene ad esempio le dipendenze del progetto.

Gestione dipendenze in Spring

Tutte le dipendenze necessarie ad un'app (come moduli, librerie, ecc.) vengono gestite tramite software di build automation e dependency management, come Maven e Gradle.

Nel caso di Maven, le dipendenze di un'app sono indicate in un file **pom.xml**.

Nell'esempio stiamo aggiungendo il modulo web e il modulo test.

```
<project xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <version>0.0.1</version>
    <name>project-name</name>
    <description>project-description</description>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

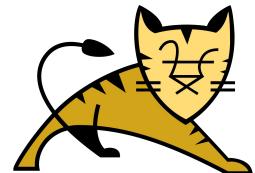
Spring Boot

Configurare un'applicazione Spring non è banale, oltre alle dipendenze è necessario configurare manualmente molte delle funzionalità, ad esempio tramite XML.

Spring Boot semplifica la creazione di applicazioni Spring-based, richiedendo una configurazione minima.

Alcuni vantaggi nel configurare un'applicazione Spring tramite Spring Boot:

- Riduzione del tempo necessario per la configurazione, ad esempio eliminando le configurazioni XML;
- Una soluzione opinionata ma che possa essere facilmente estesa e personalizzata;
- Funzionalità comuni a tutte le app, come sicurezza, health-check, configurazione esternalizzata, ecc.



Embedded web server

Spring Boot fornisce anche un web server insieme al necessario per l'applicazione.

In questo modo non è necessario dover configurare un web server in cui deployare l'app ma basterà avviare l'app e il web server sarà avviato automaticamente con l'applicazione deployata al suo interno.

Di default, il web server è Apache Tomcat (o semplicemente Tomcat).

Tomcat è open source e sviluppato dalla Apache Software Foundation, che implementa diverse specifiche come JavaServer Pages e servlet, fornendo quindi una piattaforma software per l'esecuzione di applicazioni web sviluppate in Java.

Tomcat include anche funzionalità di web server tradizionali, ereditate da Apache HTTP Server.

Creazione progetto

Creare un'applicazione con Spring Boot è estremamente semplice.

Tramite Spring Initializer è possibile generare un progetto Spring Boot con pochi click. Integrato nella maggior parte degli IDE, è anche disponibile online (<https://start.spring.io>).

Le dipendenze si gestiscono come per Spring, ad esempio con un file pom.xml nel caso di Maven.

È possibile selezionare un pool di dipendenze di partenza al momento della creazione del progetto.

Esempio creazione progetto

**Project**

- Gradle - Groovy Gradle - Kotlin
 Maven

Language

- Java Kotlin Groovy

Spring Boot

- 3.2.0 (SNAPSHOT) 3.2.0 (RC2) 3.1.6 (SNAPSHOT) 3.1.5
 3.0.13 (SNAPSHOT) 3.0.12 2.7.18 (SNAPSHOT) 2.7.17

Project MetadataGroup Artifact Name Description Package name Packaging Jar WarJava 21 17 11 8**Dependencies****ADD DEPENDENCIES... ⌘ + B****Spring Web** WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

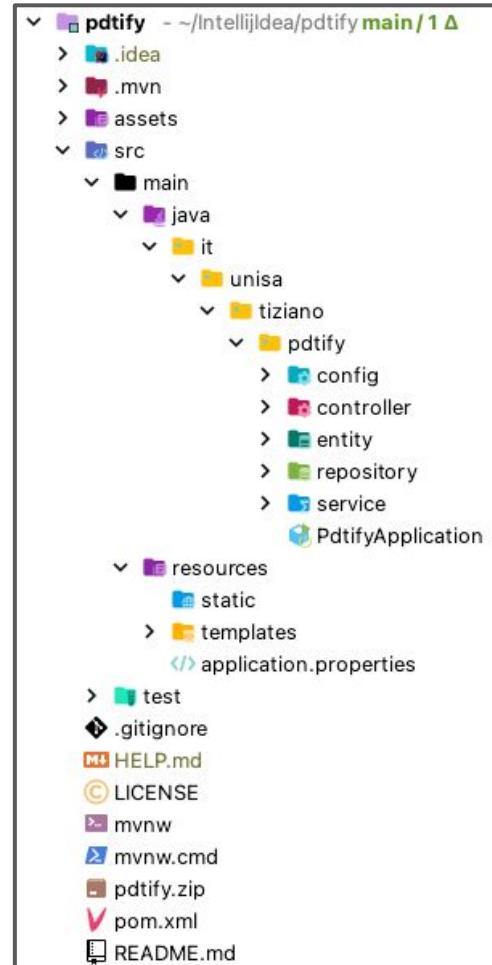
Thymeleaf TEMPLATE ENGINES

A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Struttura progetto

Un progetto Spring Boot ha sempre:

- Un file pom.xml;
- Una directory src/main/java in cui scrivere il codice dell'applicazione;
- Una directory src/main/resources in cui aggiungere risorse statiche o configurazioni;
- Una directory src/test in cui scrivere i test per il codice dell'applicazione;
- Una classe <Nome>Application.java contenente lo starting point dell'applicazione.



Starting Point dell'applicazione

Una classe <Nome>Application.java contiene lo starting point dell'applicazione.

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

Configurare proprietà dell'applicazione

Un'applicazione necessita spesso di proprietà il cui valore non può essere configurato automaticamente da Spring Boot, come ad esempio la stringa di connessione per un database.

Queste proprietà possono essere specificate all'interno del file **application.properties** o del file **application.yml**.

Esempio di proprietà di base e comuni a tutte le applicazioni Spring-based:

```
server.port = 8080
```

```
logging.level.root = INFO
```

Spring IoC Container

Lo Spring IoC Container è una componente cruciale del modulo core. Responsabile per la creazione, la configurazione e la gestione dell'intero ciclo di vita degli oggetti.

Gli oggetti gestiti dal container sono detti Spring Beans e sono gestiti tramite la dependency injection.

Il container riceve istruzioni su quali oggetti istanziare, configurare e gestire tramite i metadati di configurazione forniti.

I metadati di configurazione possono essere rappresentati tramite XML, **annotazioni** o codice.

Spring Stereotype

Spring fornisce annotazioni speciali usate per creare automaticamente i bean nel contesto dell'applicazione.

L'annotazione **@Component** rappresenta lo stereotype principale e generico per definire uno Spring Bean.

Altre annotazioni derivano da **@Component**, come **@Service**, **@Repository**, **@Controller** e **@RestController**.

Principali Spring Stereotype

@Service: una classe è annotata con @Service se contiene la logica di business.

@Repository: una classe è annotata con @Repository per indicare che si occupa di operazioni CRUD, di solito viene utilizzata con implementazioni DAO (Data Access Object) o repository che interagiscono con database, cache, ecc.

@Controller: una classe annotata con @Controller indica una componente responsabile di gestire le richieste degli utenti e restituire la risposta appropriata, che può anche essere HTML.

@RestController: una classe annotata con @RestController è simile ad una classe annotata con @Controller ma che si occupa solo di API backend e quindi invierà solo risposte in formati come JSON.

Esempio @Component

```
import org.springframework.stereotype.Component;

@Component
public class GreetingLogger {
    public void log() {
        System.out.println("Hello there!");
    }
}
```

Vedremo più avanti esempi di @Repository, @Service, @Controller e @RestController.

@Configuration e @Bean

Una delle annotazioni più importanti in Spring è l'annotazione **@Configuration**.

@Configuration indica che la classe ha metodi annotati con **@Bean** e che quindi definiscono come creare particolari **Spring Bean** necessari all'applicazione.

Spring al momento di creare il bean della classe **@Configuration**, genererà i Spring Bean definiti dai metodi **@Bean**, che potranno essere usati tramite dependency injection.

Vedremo un esempio di **@Configuration** e **@Bean** più avanti.

Utile per i POJO non definiti dall'applicazione.

Scope dei Spring Bean

Di default i bean sono gestiti come singleton, ma possono essere configurati per far sì che ne venga generata una nuova istanza per ogni richiesta oppure per ogni sessione.

La maniera in cui il container gestisce il ciclo di vita del bean, definisce lo **scope del bean**.

Esempi di scopo:

- Singleton;
- Request;
- Session;
- Prototype.

Spring Web MVC

Anche noto come Spring Web o Spring MVC.

Spring Web MVC è il framework web costruito sulla Servlet API di Java ed è stato incluso in Spring fin dal principio.

Fornisce tutto il necessario per sviluppare applicazioni web, REST API, backend, ecc.

Importato nei progetti aggiungendo il modulo [spring-boot-starter-web](#).

```
<project xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <version>0.0.1</version>
    <name>project-name</name>
    <description>project-description</description>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

@Controller e @RestController

Spring Web Consente di creare bean controller per gestire le richieste HTTP in ingresso, tramite le annotazioni @Controller o @RestController.

Il mapping dei controller alle richieste HTTP avviene utilizzando l'annotazione **@RequestMapping**, che indica il path base della richiesta che il controller deve gestire (ad es. /users).

I metodi dei controller vengono etichettati con annotazioni che permettono di specificare il path della richiesta HTTP che il singolo metodo deve gestire: @GetMapping, @PostMapping, @DeleteMapping, ecc.

- Permettono anche di specificare parti del path da usare come variabili (vedremo dopo un esempio).

Esempio di @RestController

```
@RestController
@RequestMapping( "/users" )
public class UserController {

    @GetMapping( "/" )
    public User getUsers() {
        // ...
    }

    @PostMapping( "/" )
    public User saveUser(
        // ...
    ) {
        // ...
    }

    @DeleteMapping( "/delete-users" )
    public void deleteUsers() {
        // ...
    }
}
```

Il controller *UserController* gestirà le richieste all'URL <https://my-domain/users>.

Nel dettaglio:

- Il metodo *getUsers()* gestirà le richieste GET all'URL <https://my-domain/users>.
- Il metodo *saveUser()* gestirà le richieste POST all'URL <https://my-domain/users>.
- Il metodo *deleteUsers()* gestirà le richieste DELETE all'URL <https://my-domain/users/delete-users>.

Elementi delle richieste HTTP

Spring Web fornisce nel contesto dell'applicazione dei bean *HttpMessageConverters* che si occupano di convertire in tipi primitivi e oggetti istanze di classi Java i seguenti elementi:

- **Parametri nella *query string* della richiesta:** @RequestParam
- **Parametri nel path della richiesta:** @PathVariable
- **Header della richiesta:** @RequestHeader
- **Body della richiesta:** @RequestBody

Esempio mapping delle richieste

```
@RestController
@RequestMapping( "/users" )
public class UserController {

    @GetMapping( "/{userId}" )
    public User getUser(@PathVariable UUID userId) {
        // ...
    }

    @PostMapping( "/" )
    public void saveUser(
        @RequestHeader( "retries" ) Integer retries,
        @RequestBody User user
    ) {
        // ...
    }

    @DeleteMapping( "/{userId}" )
    public void deleteUser(@PathVariable UUID userId) {
        // ...
    }
}
```

- Il metodo `getUser()` gestirà le richieste GET all'URL <https://my-domain/users/:userId>, dove **userId** è una variabile con valore diverso in ogni richiesta.
- Il metodo `saveUser()` gestirà le richieste POST all'URL <https://my-domain/users> e riceverà un body da mappare ad un'istanza della classe **User** e un parametro **retries** nell'header HTTP indicante il numero massimo di tentativi per effettuare il salvataggio.
- Il metodo `deleteUser()` gestirà le richieste DELETE all'URL <https://my-domain/users/:userId>, dove **userId** è una variabile con valore diverso in ogni richiesta.

@Service e @Autowired

Buona pratica di sviluppo prevede che i controller si occupino solo della gestione delle richieste e risposte e che la logica di business dell'applicazione sia gestita da dei servizi.

I servizi vengono forniti ai controller tramite **dependency injection**. La dependency injection in Spring è dichiarata tramite l'annotazione **@Autowired**.

Implementare un servizio prevede la definizione di un'interfaccia e una classe per implementare i metodi definiti da annotare con l'annotazione **@Service**.

Esempio @Service

Interfaccia

```
public interface UserService {  
  
    User getUser(UUID userId);  
    void saveUser(User user);  
    void deleteUser(UUID userId);  
}
```

Implementazione

```
@Service  
public class UserServiceImpl implements UserService {  
  
    @Override  
    public User getUser(UUID userId) {  
        // ...  
    }  
  
    @Override  
    public void saveUser(User user) {  
        // ...  
    }  
  
    @Override  
    public void deleteUser(UUID userId) {  
        // ...  
    }  
}
```

Esempio @Autowired

Facciamo dependency injection del servizio nel controller.

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{userId}")
    public User getUser(@PathVariable UUID userId) {
        // ...
    }

    @PostMapping("/")
    public void saveUser(
        @RequestHeader("retries") Integer retries,
        @RequestBody User user
    ) {
        // ...
    }

    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable UUID userId) {
        // ...
    }
}
```



```
@RestController
@RequestMapping("/users")
public class UserController {

    private UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/{userId}")
    public User getUser(@PathVariable UUID userId) {
        userService.getUser(userId);
    }

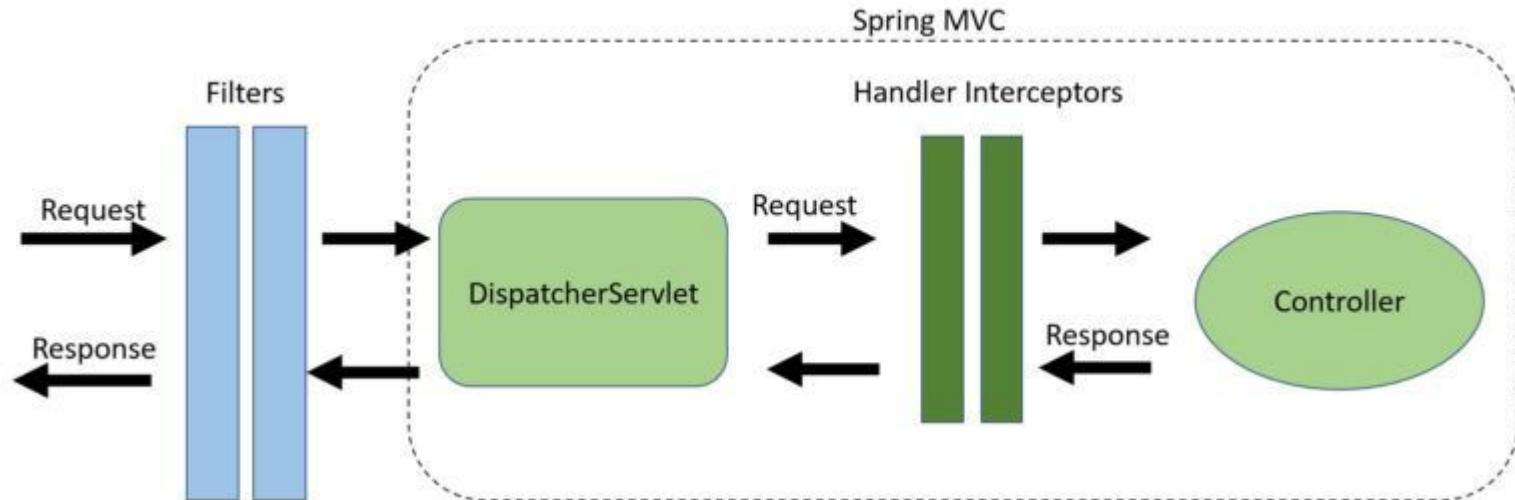
    @PostMapping("/")
    public void saveUser(
        @RequestHeader("retries") Integer retries,
        @RequestBody User user
    ) {
        userService.saveUser(user);
    }

    @DeleteMapping("/{userId}")
    public void deleteUser(@PathVariable UUID userId) {
        userService.deleteUser(userId);
    }
}
```

Intercettare richieste HTTP

In Spring ci sono due componenti in grado di intercettare richieste HTTP:

- **Filtri**: eseguiti prima di individuare il controller che gestirà la richiesta;
- **HandlerInterceptor**: eseguiti prima, durante e dopo la gestione della richiesta.



DispatcherServlet è la componente che si occupa di fare il forwarding delle richieste ai controller.

Filtri

I filtri vengono eseguiti prima della DispatcherServlet e quindi prima che le richieste siano gestite dall'applicazione.

Possiamo utilizzare i filtri per manipolare e bloccare le richieste in ingresso prima che raggiungano i controller.

Allo stesso modo, possiamo anche bloccare le risposte prima che raggiungano il client.

Possiamo usare i filtri per innumerevoli funzionalità, tra cui:

- Autenticazione;
- Auditing;
- Compressione di immagini e dati;
- Qualsiasi funzionalità che vogliamo disaccoppiare da Spring MVC.

Esempio Filtro

```

@Component
public class UserLoggingFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain,
    ) throws ServletException, IOException {
        try {
            String method = request.getMethod();
            if ("POST".equalsIgnoreCase(method)) {
                Integer retries =
                    Integer.parseInt(request.getHeader("retries"));
                System.out.printf("Numer of retries: %d", retries);
            }
        } catch (NumberFormatException e) {/*...*/}
        filterChain.doFilter(request, response);
    }

    @Override
    protected boolean shouldNotFilter(HttpServletRequest request) {
        String path = request.getServletPath();
        return !path.startsWith("/users");
    }
}

```

Il filtro intercetta solo le richieste per il path `/users` come definito dal metodo **shouldNotFilter()**.

Per ogni richiesta:

- Ottiene il metodo HTTP utilizzato;
- Se POST allora inserisce un log indicante **il numero di tentativi per il salvataggio dell'utente**;
- Prima di terminare, indica che la richiesta può proseguire con il metodo **doFilter()**.

La richiesta sarà poi gestita dal metodo **saveUser()** del controller nell'esempio precedente.

Handler Interceptor

Eseguiti dopo la DispatcherServlet e forniscono metodi per eseguire logica:

- Prima che le richieste siano gestite dai controller;
- Dopo che sono state gestite dai controller ma prima che vengano renderizzate eventuali pagine;
- Dopo il completamento della richiesta.

Possono essere usati per funzionalità come:

- Modificare una pagina prima che sia renderizzata;
- Controlli specifici legati all'autorizzazione;
- Manipolazione del contesto dell'applicazione.

Esempio Handler Interceptor

```
@Component
public class LogInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(
        HttpServletRequest request, HttpServletResponse response,
        Object handler
    ) throws Exception {
        System.out.println("preHandle invoked");
        return true;
    }

    @Override
    public void postHandle(
        HttpServletRequest request, HttpServletResponse response,
        Object handler, ModelAndView modelAndView
    ) throws Exception {
        System.out.println("postHandle invoked");
    }

    @Override
    public void afterCompletion(
        HttpServletRequest request, HttpServletResponse response,
        Object handler, Exception ex
    ) throws Exception {
        System.out.println("afterCompletion invoked");
    }
}
```

preHandle() è eseguito prima che le richieste siano gestite dai controller.

postHandle() è eseguito dopo che sono state gestite dai controller ma prima che vengano renderizzate le pagine.

- Il parametro di tipo **ModelAndView** fornisce accesso alle pagine e al loro contenuto.

afterCompletion() è eseguito dopo il completamento della richiesta.

- Il parametro di tipo **Exception** permette di eseguire logiche condizionate dalla presenza di un errore.

Spring Data

La missione di Spring Data è fornire un modello di programmazione Spring-based per l'accesso ai dati.

Semplifica l'uso di tecnologie di storage, come database relazionali e non relazionali, framework di map-reduce, servizi di dati basati su cloud, ecc.

Spring Data è una famiglia di moduli che contiene numerosi sotto-moduli specifici per tecnologia.

Un modulo di base **Spring Data Commons** funge da building block per tutti gli altri moduli.

Esempi di moduli di Spring Data

- **Spring Data JPA:** modulo di Spring Data per JPA.
- **Spring Data MongoDB:** modulo di Spring Data per lavorare con MongoDB.
- **Spring Data Redis:** modulo per il caching in Redis.
- **Spring Data Azure Cosmos DB:** modulo di Spring Data per Microsoft Azure Cosmos DB.
- **Spring Data Cloud Datastore:** modulo di Spring Data per Google Datastore.
- **Spring Data DynamoDB:** modulo di Spring Data per DynamoDB.
- ...

Ci focalizzeremo su Spring Data JPA.

Spring Data JPA

Semplifica l'implementazione di repository basate su JPA (Java Persistence API) per l'accesso a database relazionali.

- Una repository è una componente che si occupa di operazioni CRUD verso una soluzione di storage.

Gestire l'accesso ai dati per un'applicazione può essere oneroso. Tanto codice boilerplate può dover essere scritto per eseguire anche le query più semplici e se si aggiungono funzionalità come la paginazione, l'auditing e altre opzioni spesso necessarie, il boilerplate e la complessità aumentano.

Spring Data JPA mira a migliorare l'implementazione di funzionalità tipiche per l'accesso ai dati riducendo lo sforzo di sviluppo.

- Tipicamente la definizione di interfacce per repository a cui Spring fornirà una implementazione.

 **HIBERNATE**

Hibernate è un Object Relational Mapping (ORM) che permette di semplificare la gestione dello strato di persistenza delle applicazioni.

Hibernate è anche un'implementazione della specifica JPA. Di conseguenza, può essere facilmente utilizzato in qualsiasi ambiente che supporti JPA, incluse le applicazioni Java SE e Java EE.

Spring Data JPA è un'astrazione al di sopra di Hibernate che lo compatibile con il framework Spring.

Importare Spring Data JPA

Bisogna aggiungere il modulo
spring-boot-starter-data-jpa.

Bisogna poi aggiungere il connector
Java dello specifico database a cui
ci si conserverà, nell'esempio MySQL.

```
<project xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <version>0.0.1</version>
    <name>project-name</name>
    <description>project-description</description>
    <dependencies>
        <!-- Other dependencies -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.33</version>
        </dependency>
    </dependencies>
</project>
```

Configurare l'accesso al database

Bisogna indicare il database a cui connettersi.

Si utilizzano delle proprietà definite da Spring Data JPA nel file application.properties.

```
spring.datasource.url = jdbc:mysql://localhost:3306/mydb?createDatabaseIfNotExist=true
spring.datasource.username = root
spring.datasource.password =
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver
```

Entità

Oggetti POJO che rappresentano dati che possono essere memorizzati nel database.

Esempio entità per un utente

```
@Entity  
public class User {  
    @Id  
    @GeneratedValue(strategy = GenerationType.UUID)  
    private UUID ID;  
    private String name;  
    private String username;  
    private String username;  
}
```

@Entity permette di definire un POJO come un'entità.

@Id permette di definire la chiave primaria, che sarà generata automaticamente perché annotata con **@GeneratedValue**.

Esempio di @Repository con Spring Data JPA

Definiamo l'interfaccia per la repository a cui Spring fornirà l'implementazione:

```
@Repository  
public interface UserRepository extends JpaRepository<User, UUID>{}
```

La sola definizione di questa interfaccia ci fornirà metodi comuni come:

- `findById();`
- `findAll();`
- `save();`
- `saveAll();`
- `count();`
- `...`

Metodi custom nelle repository

All'interfaccia per la repository:

```
@Repository  
public interface UserRepository extends JpaRepository<User, UUID>{}
```

Aggiungiamo un metodo custom per recuperare un utente tramite **username**:

```
@Repository  
public interface UserRepository extends JpaRepository<User, UUID> {  
  
    User findByUsername(String username);  
}
```

Il solo dichiare il metodo ci permette di indicare a Spring Data di fornire un'implementazione.

@Autowired repository nel servizio

Facciamo dependency injection della repository nel servizio.

```
@Service
public class UserServiceImpl implements UserService {

    @Override
    public User getUser(UUID userId) {
        // ...
    }

    @Override
    public void saveUser(User user) {
        // ...
    }

    @Override
    public void deleteUser(UUID userId) {
        // ...
    }
}
```



```
@Service
public class UserServiceImpl implements UserService {

    private UserRepository userRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUser(UUID userId) {
        userRepository.findById(userId);
    }

    public void saveUser(User user) {
        userRepository.save(user);
    }

    public void deleteUser(UUID userId) {
        userRepository.deleteById(userId);
    }
}
```

Spring Session

Spring Session ha l'obiettivo di superare i limiti attuali della gestione delle sessioni HTTP memorizzate direttamente sui web server.

Facilita la condivisione dei dati di sessione tra servizi locali o anche nel cloud senza essere vincolati a un singolo contenitore per la sessione (ad esempio Tomcat).

- Può aiutare ad alleggerire il carico di memoria del server, ad esempio.

Vedremo un esempio molto semplice di migrazione della sessione in Redis.



Redis

Redis è un progetto **open source** che fornisce uno store in-memory di coppie chiave-valore.

Redis offre una serie di strutture dati in memoria molto versatili, che permettono di supportare un'ampia gamma di applicazioni con diverse necessità.

Tra i principali casi d'uso per Redis ci sono il caching e la gestione di sessioni.

Può essere usato anche come database.

Configurare Spring Session con Redis

Aggiungere due semplici dipendenze:

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

E utilizzare la sessione
come nell'esempio:

```
@RestController
public class SessionMessageController {

    @GetMapping("/")
    public List<String> getMessages(HttpSession session) {
        List<String> messages =
            (List<String>) session.getAttribute("messages");
        return messages;
    }
}
```

Spring Security

Spring Security è un framework potente e altamente personalizzabile per l'autenticazione e il controllo degli accessi.

È lo standard de facto per la sicurezza delle applicazioni basate su Spring.

Come tutti i moduli Spring, la vera potenza di Spring Security si trova nella facilità con cui può essere esteso per soddisfare requisiti personalizzati.

Sicuramente tra i più complessi da utilizzare nel modo corretto.

Configurare Spring Security

Aggiungere Spring Security ad un progetto:

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

Abilitare Spring Security creando una classe con le annotazione **@Configuration** e **@EnableWebSecurity**.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
    // ...
}
```

Esempio regole di autenticazione

```
@Configuration  
@EnableWebSecurity  
public class WebSecurityConfig {  
  
    @Bean  
    public SecurityFilterChain securityFilterChain(  
        HttpSecurity http  
    ) throws Exception {  
        http  
            .authorizeHttpRequests((requests) -> requests  
                .requestMatchers("/", "/home").permitAll()  
                .anyRequest().authenticated()  
            )  
            .formLogin((form) -> form  
                .loginPage("/login").permitAll()  
            )  
            .logout((logout) -> logout.permitAll());  
        return http.build();  
    }  
}
```

Il bean **SecurityFilterChain** definisce quali path URL sono da proteggere e quali non lo sono:

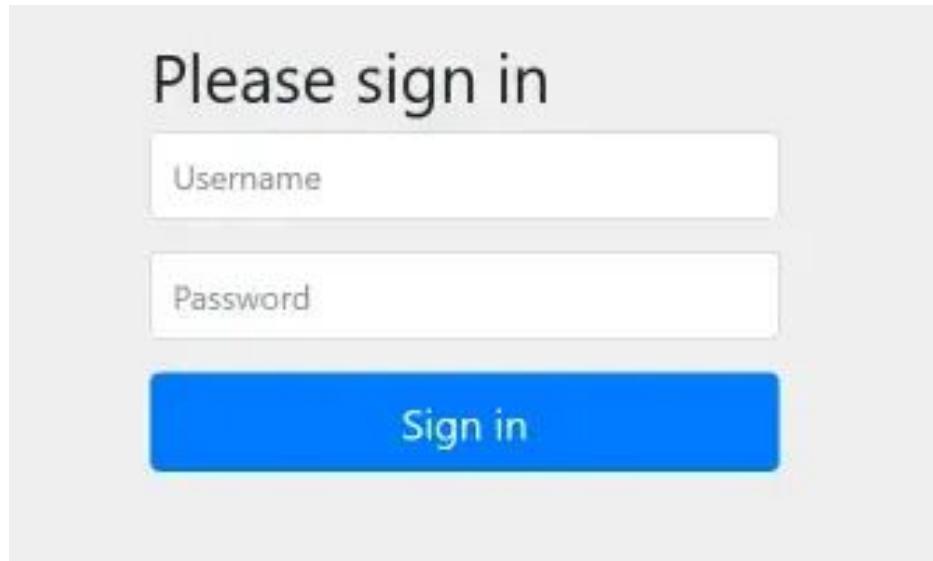
- / e /home non richiedono autenticazione.
- Tutti gli altri path richiedono autenticazione.

Logica custom per la pagina /login specificata dal metodo loginPage(): tutti hanno il permesso di accedervi.

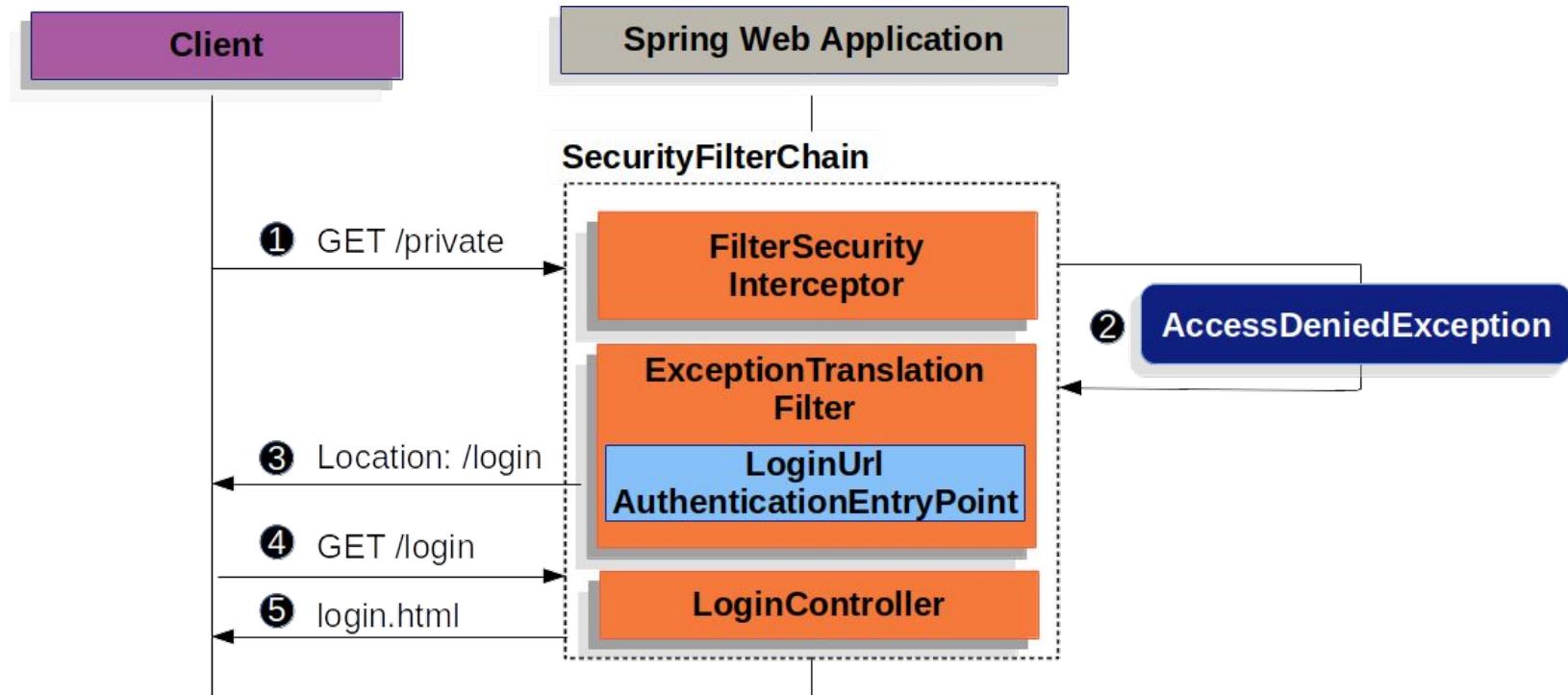
Quando un utente fa il login, viene rimandato alla pagina cui voleva accedere prima di autenticarsi.

Pagina di login di default

Anche se non specificate una pagina di login personalizzata, Spring Security ne fornisce una di default.



Flusso di autenticazione



Disabilitare pagina di login di default

Talvolta le applicazioni non possono richiedere autenticazione manuale.

Si disabilita l'autenticazione manuale configurata di default e si implementano meccanismi "automatici" supportati da Spring Security, ad esempio usando token JWT (JSON Web Token).

```
@Configuration
@EnableWebSecurity
public class DefaultLoginSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity security) throws Exception {
        security.httpBasic().disable();
    }
}
```

Spring Cloud

Spring Cloud fornisce strumenti per consentire agli sviluppatori di sviluppare e gestire agilmente alcuni dei pattern comuni nei sistemi distribuiti.

Esempi: service discovery, API gateway, circuit breaker, lock globali, sessioni distribuite, ecc.

La coordinazione dei sistemi distribuiti può portare a implementazioni ripetitive.

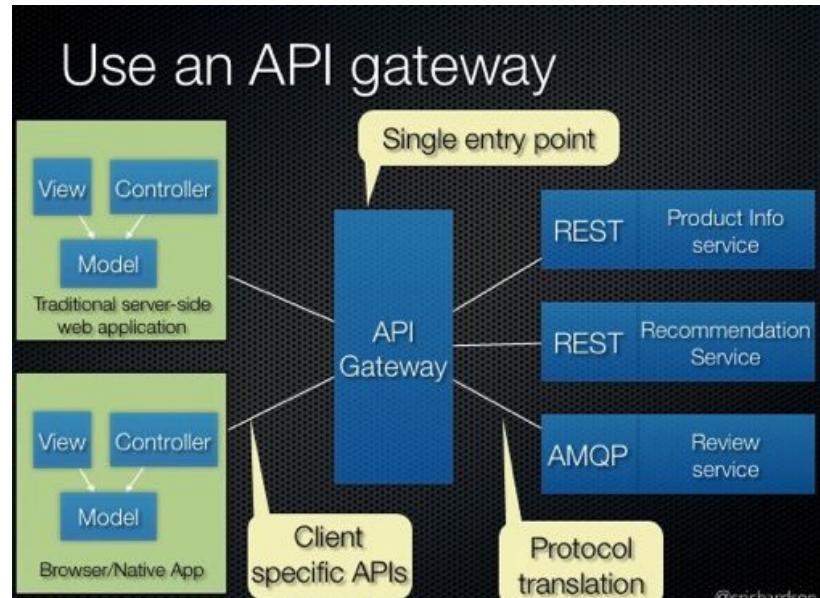
Con Spring Cloud è possibile ottenere soluzioni a problematiche comuni con effort minore.

API Gateway

L'API gateway è il singolo punto d'ingresso per tutti i client ai dati o ai servizi back-end.

Gestisce tutte le attività di accettazione ed elaborazione relative alle chiamate API, incluse:

- Gestione del traffico;
- Supporto CORS;
- Controllo di accessi e autorizzazioni;
- Monitoraggio;
- Terminazione TLS/SSL.
- Gestione delle versioni delle API.



@cnichardson

Service Discovery

Un'applicazione distribuita è eseguita in ambienti dove il numero di istanze dei servizi e la loro locazione cambiano dinamicamente.

Risulta necessario un meccanismo che permetta ai clienti di un servizio di inviare richieste ad un insieme di istanze di servizi che cambia dinamicamente.

La service discovery permette a client, API gateway e altri servizi di scoprire la locazione delle istanze di un servizio.

Distinguiamo:

- client-side service discovery
- server-side service discovery

Service Registry

Il service registry è un database dei servizi, delle loro istanze e delle loro locazioni.

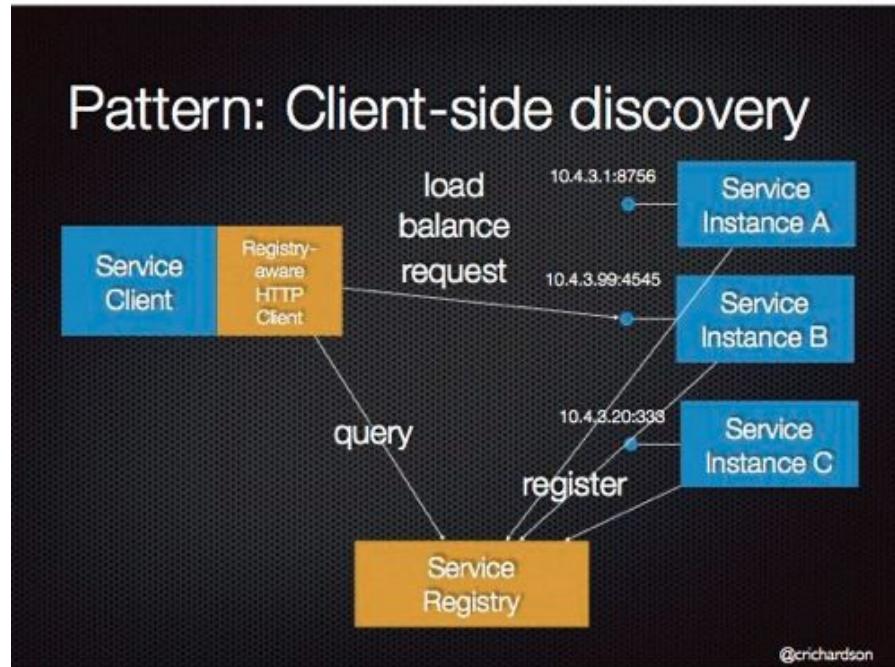
Interrogato per individuare le locazioni delle istanze dei servizi.

I servizi si registrano durante la fase di avvio e rimuovono la propria registrazione durante la fase di spegnimento.

Client-side Service Discovery

Quando il client effettua una chiamata ad un servizio, interroga il service registry per ottenere le locazioni delle istanze del servizio.

Il client esegue load balancing ed inoltra la richiesta ad un'istanza disponibile del servizio.

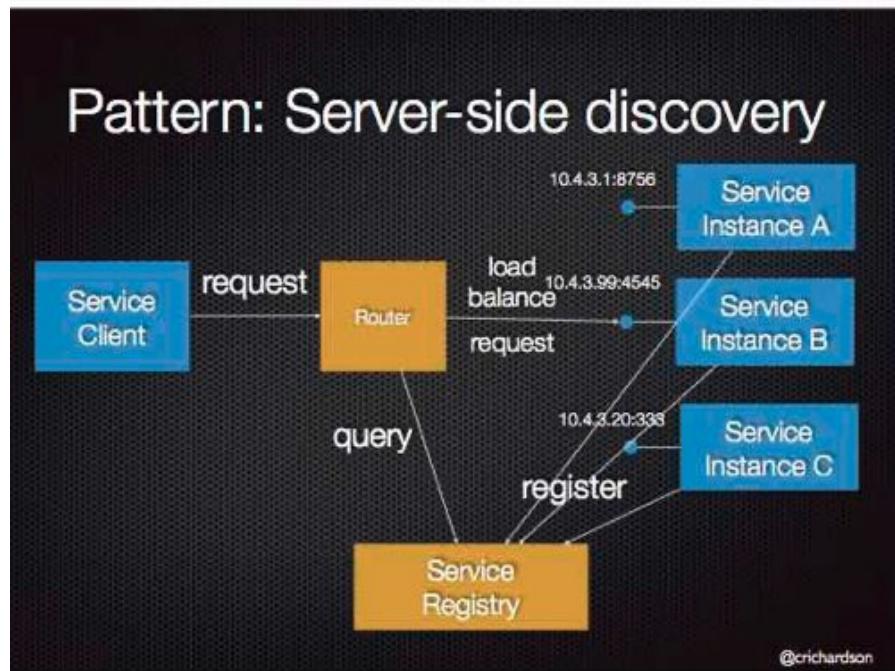


@crichardson

Server-side Service Discovery

Quando il client effettua una chiamata ad un servizio, invia una richiesta ad un load balancer che è in esecuzione ad una locazione ben nota.

Il load balancer interroga il service registry e inoltra la richiesta ad un'istanza disponibile del servizio.





Netflix OSS

Soluzioni software open-source sviluppate da Netflix (<https://github.com/Netflix>).

Alcune di queste soluzioni sono state integrate da Spring all'interno di Spring Cloud.

Esempi:

- **Zuul**: un API gateway a livello 7 che offre funzionalità per routing dinamico, monitoraggio, resilienza, sicurezza, ecc.
- **Eureka**: un servizio utilizzato per service discovery, bilanciamento del carico e il failover, che gioca un ruolo critico anche nell'infrastruttura di Netflix.
- **Hystrix**: è una soluzione di circuit breaking progettata per isolare punti di accesso in caso di fallimento per fermare il fallimento a cascata di servizi dipendenti punto di accesso.

Spring Cloud Netflix

Spring Cloud Netflix fornisce integrazioni di Netflix OSS per le applicazioni Spring-based.

Tramite annotazioni è possibile abilitare e configurare rapidamente pattern comuni all'interno delle applicazioni in ambiente distribuito.

Particolarmente utile per lo sviluppo di applicazioni basate su architetture orientate ai microservizi.

Ad esempio, permette di creare un Server Eureka che gestisca il service registry e poi far si che le applicazioni si registrino a questo server.

Creare un server Eureka (1)

- Aggiungere la dipendenza per il server Eureka:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

- Configurare le proprietà necessarie nel file application.yml (alternativa ad application.properties):

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Creare un server Eureka (2)

- Annotare la classe di partenza del progetto Spring Boot con **@EnableEurekaServer**:

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

Accedere alla dashboard di Eureka

The screenshot shows the Spring Cloud Eureka dashboard interface. At the top, there's a header with the Eureka logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section displays various system metrics. The 'DS Replicas' section lists a single instance named 'localhost'. The 'Instances currently registered with Eureka' section shows no instances available. Finally, the 'General Info' section provides detailed information about the system's memory usage, environment, and CPU count.

Name	Value
total-avail-memory	466mb
environment	test
num-of-cpus	8

Creare un client Eureka (1)

- Aggiungere la dipendenza per il client Eureka:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- Configurare le proprietà necessarie nel file application.yml:

```
spring:
  application:
    name: spring-cloud-eureka-client
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
```

Creare un client Eureka (2)

- Annotare la classe di partenza del progetto Spring Boot con **@EnableEurekaClient**:

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

Visualizzare i client registrati ad Eureka

Accedendo nuovamente alla dashboard è possibile vedere la nuova applicazione registrata ad Eureka.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SPRING-CLOUD-EUREKA-CLIENT	n/a (1)	(1)	UP (1) - heisenbug:spring-cloud-eureka-client:0
General Info			
Name	Value		
total-avail-memory	587mb		
environment	test		
num-of-cpus	8		
current-memory-usage	213mb (36%)		
server-upptime	00:02		

Utilizzare Eureka per la service discovery

```
@Autowired  
private EurekaClient eurekaClient;  
  
public String getSpringCloudEurekaClient() {  
    InstanceInfo service = eurekaClient  
        .getApplication(spring-cloud-eureka-client)  
        .getInstances()  
        .get(0);  
  
    String hostName = service.getHostName();  
    int port = service.getPort();  
  
    // ...  
}
```

Un bean **EurekaClient** permette di accedere al service registry per ottenere le informazioni sulle applicazione connesse.

Si utilizza il metodo **getApplication()** per ottenere il riferimento all'applicazione.

Con il metodo **getInstances()** si accede alle istanze dell'applicazione.

Dalle istanze possiamo ottenere *host* e *porta* per le chiamate HTTP, il tutto conoscendo inizialmente solo il nome dell'applicazione da chiamare, ovvero **spring-cloud-eureka-client**.



Thymeleaf è un template engine per Java per elaborare e gestire HTML, XML, JavaScript e CSS.

Per aggiungerlo ad un progetto Spring Boot usare la dipendenza:

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

E creare una directory `templates` nella directory `src/main/resources` del progetto, dove aggiungere le pagine HTML create.

Esempio pagina HTML con Thymeleaf

Thymeleaf fornisce degli attributi identici a quelli dei tag HTML ma che hanno il prefisso **th:** e che permettono di mostrare valori “dinamici”.

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>User Details</title>
    </head>
    <body>
        <h1 th:text="th:text='Welcome ' + ${user.name}""/>
    </body>
</html>
```

Creiamo un file **userDetails.html** nella directory
src/main/resources/templates.

Nell'esempio **th:text** è usato per parametrizzare il testo da mostrare nell'elemento **<h1>**.

Come fornire i parametri alle pagine HTML

```
@Controller
public class UserDetailsController {

    private final UserDetailsService userDetailsService;

    @Autowired
    public UserDetailsController(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @GetMapping("/user-details")
    public String getUserDetails(
        @RequestParam("id") UUID id,
        Model model
    ) {
        User user = userDetailsService.findById(id);
        model.addAttribute("user", user);
        return "userDetails";
    }
}
```

Creiamo un `@Controller` per fornire dati alla pagina.

- La stringa restituita dal controller deve corrispondere al nome del file HTML.

Usiamo un oggetto **Model** iniettato nel metodo del controller per passare i parametri alla pagina tramite coppie chiave-valore.

La chiama sarà l'identificativo del dato nella pagina, difatti accediamo al nome dell'utente come segue:

```
<h1 th:text="th:text='Welcome ' + ${user.name}""/>
```

Librerie utili

- **Lombok**: libreria per l'autogenerazione di boilerplate tramite annotazioni. Ad esempio `@Getter` per generare metodi getter per tutti i campi di una classe;
- **ModelMapper**: libreria per la conversione tra oggetti istanze di due classi Java per ridurre il boilerplate;
- **Jackson**: libreria per serializzare e deserializzare JSON in Java (esiste anche per XML);
- **Apache Commons**: fornisce una serie di librerie per semplificare la gestione di stringhe, collezioni, ecc.

Esempio Lombok

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

```
@NoArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@ToString
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID ID;
    private String name;
    private String username;
    private String username;
}
```

Aggiungiamo la dipendenza per Lombok.

Con le annotazioni **@NoArgsConstructor** e **@AllArgsConstructor** generiamo due costruttori.

Con l'annotazione **@ToString** generiamo un metodo `toString()` con tutti i campi della classe.

Con le annotazioni **@Getter** e **@Setter** generiamo i metodi getter e setter per tutti i campi della classe.

- Esempio: `getName()` e `setName()`.

Esempio ModelMapper

Aggiungiamo la dipendenza:

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
</dependency>
```

Sviluppiamo una classe **UserMapper** che usa **ModelMapper** per convertire le istanze di una classe **UserEntity** in delle istanze di una classe **UserDTO**:

```
public class UserMapper {
    public UserDTO map(UserEntity user) {
        Model mapper = new ModelMapper();
        return mapper.map(user, UserDTO.class);
    }
}
```

Esempio Jackson

Aggiungiamo la dipendenza:

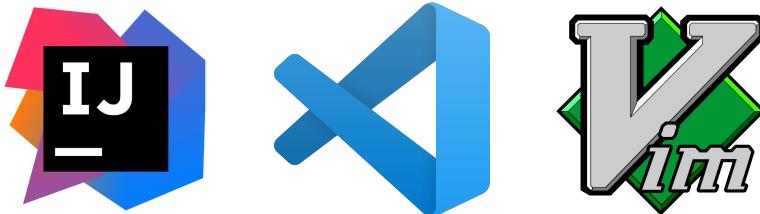
```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Sviluppiamo una classe **UserSerializer** che usa **Jackson** per serializzare in JSON le istanze di una classe **UserDTO**:

```
public class UserSerializer {
    public String serialize(UserDTO user) {
        ObjectMapper objectMapper = new ObjectMapper();
        return objectMapper.writeValueAsString(user);
    }
}
```

Ambiente di sviluppo

IDE & Text Editor

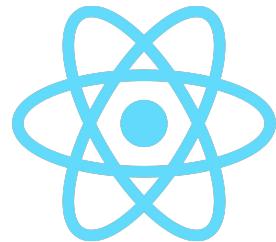


Linguaggi di Programmazione e Scripting



Ambiente di sviluppo

Framework



Version Control



Azure DevOps

Ambiente di sviluppo

Monitoraggio e gestione delle applicazioni



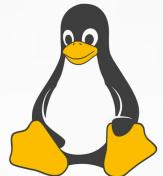
elasticsearch



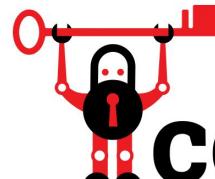
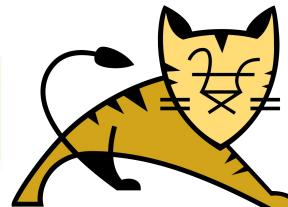
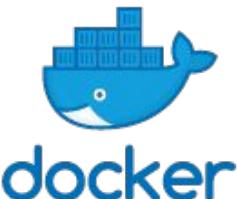
kibana



Distribuzione delle applicazioni

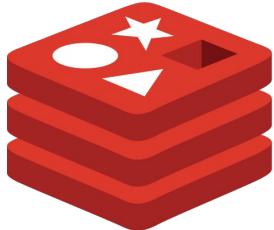


Linux™

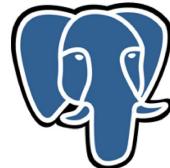


Ambiente di sviluppo

Caching & Comunicazione



Database



Postgre[®]SQL

Autenticazione e autorizzazione





Domande?

Esempio Pratico



spring

Prossimamente su

<https://github.com/tizianocitro/pdtify>



ISISLab