Ecosistema Spring

The Popularity of Spring

Spring è una tecnologia di [back-end] che ha diverse finalità e usi, il più comune riguarda lo sviluppo di applicazioni web. Spring è un nome generico che può riferirsi al core del framework, ma più spesso si riferisce a tutto l’insieme degli strumenti Spring.

Le tecnologie di back-end, come ad esempio il framework Spring, sono influenzate dal crescente successo delle tecnologie edge-cloud e dell’architettura a microservizi. Secondo quanto riportato da O’Reilly’s Technology Trends for 2024 (https://www.oreilly.com/radar/technology-trends-for-2024/), le tecnologie Java legate all’ingegneria del software cloud e alle architetture di microservizi risultano attualmente rilevanti.

Perché usare Spring?

L’obiettivo iniziale di Spring, e ancora oggi una linea guida fondamentale del framework, è rimuovere la complessità, il disordine e il codice ripetitivo. In sostanza, Spring mira a rendere più semplice per gli sviluppatori la costruzione di un sistema.

Spring non è una scelta tutto o niente. Puoi effettivamente selezionare e utilizzare solo le parti di Spring che hanno senso per il tuo sistema.

Un aspetto interessante di Spring è quanto sia un investimento a lungo termine impararlo, dato che la sua evoluzione è piuttosto unica. Da un lato, è attivamente sviluppato e sempre in miglioramento ai margini. Dall’altro, al suo nucleo, è altamente stabile.

Questo articolo evidenzia i fattori chiave della rilevanza di Spring: * Ecosistema vasto * Testabilità * Monitoraggio * Osservabilità * Sviluppo rapido * Server embedded * Open source

L’Ecosistema di Spring

Grazie al successo del framework, l’ecosistema di Spring è ora piuttosto vasto, come puoi vedere sul sito ufficiale.

Spring Core

- Tecnologie di base: iniezione delle dipendenze, eventi, risorse, i18n, validazione, data binding, conversione di tipo, SpEL, AOP.
- Testing: oggetti mock, framework TestContext, Spring MVC Test, WebTestClient.
- Accesso ai dati: transazioni, supporto DAO, JDBC, ORM, Marshalling XML.
- Integrazione: remoting, JMS, JCA, JMX, email, attività, pianificazione, cache e osservabilità

Spring Boot

Spring Boot semplifica lo sviluppo di un’applicazione Spring.

Spring Boot è un’estensione del framework Spring che fornisce una configurazione predefinita con un approccio opinato alla costruzione di applicazioni web con Spring. Prima di Boot, un’applicazione Spring necessitava di molte configurazioni solo per iniziare.

Caratteristiche principali di Spring Boot:

- Server web embedded (non è necessario distribuire file WAR)
- Fornisce dipendenze starter per semplificare la configurazione del build
- Configura automaticamente le librerie quando possibile
- Fornisce funzionalità pronte per la produzione come metriche, health-check e configurazione esternalizzata
- Evita la configurazione XML

Spring Data

Spring Data fornisce un modello di programmazione familiare e coerente per l’accesso ai dati, mantenendo comunque le caratteristiche specifiche dello storage sottostante.

- Astrazioni potenti per repository e mapping di oggetti personalizzati
- Derivazione dinamica delle query dai nomi dei metodi del repository
- Classi base di dominio di implementazione che forniscono proprietà di base
- Supporto per auditing trasparente (creazione, ultima modifica)

Spring Cloud

Spring Cloud offre un’esperienza pronta all’uso per casi d’uso tipici e un meccanismo di estensione per coprirne altri.

- Configurazione distribuita
- Registrazione e scoperta dei servizi
- Routing
- Chiamate tra servizi
- Load balancing
- Circuit Breakers
- Messaggistica distribuita

2. Com’è fatto un vero progetto

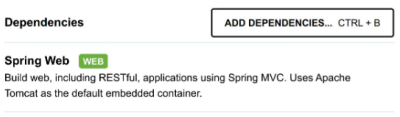
Creazione del Progetto

In questa lezione, costruiremo un semplice microservizio a livelli per gestire un catalogo di prodotti. Spring Initializr è un utile generatore di progetti Spring per un avvio rapido.

Sceglieremo un progetto Maven utilizzando Java e dovremo compilare i dettagli del progetto:

- Group: com.nbicocchi
- Artifact: product-service-no-db

Per la sezione delle dipendenze, selezioniamo la dipendenza Spring Web. Anche se non ci concentreremo sugli aspetti web in questo modulo, la includiamo perché consente di mantenere l’applicazione in esecuzione dopo l’avvio.



Ora possiamo cliccare sul pulsante “Generate” per scaricare il progetto, decomprimerlo e importarlo in un IDE.

Se stai usando IntelliJ, puoi importare il progetto andando nel menu principale, selezionando File > Open e navigando nel percorso in cui si trova il progetto per aggiungerlo.

Aggiunta del Livello di Persistenza

Creeremo un semplice livello di persistenza all’interno del package com.nbicocchi.product.persistence.model, aggiungendo una classe Product.

Livello di persistenza: livello che si occupa della memorizzazione dei dati su un DB o uno storage di massa. Solitamente il livello di persistenza contiene il model (definizione dell’entità) e il repository (operazioni eseguibili sulle entità)

```
@AllArgsConstructor
@NoArgsConstructor
@RequiredArgsConstructor
@Data
public class Product {
    private Long id;
    @NonNull @EqualsAndHashCode.Include private String uuid;
    @NonNull private String name;
    @NonNull private Double weight;
}
```

Questa classe ha tre attributi chiave: uuid, name e weight.

Successivamente, aggiungeremo il repository nel package com.nbicocchi.product.persistence.repository:

```
@Repository
public class ProductRepository {
    private final List<Product> products = new ArrayList<>();

    public Optional<Product> findById(Long id) {
        return products.stream().filter(p -> p.getId().equals(id)).findFirst();
    }

    public Optional<Product> findByUuid(String uuid) {
        return products.stream().filter(p -> p.getUuid().equals(uuid)).findFirst();
    }

    public Iterable<Product> findAll() {
        return products;
    }

    public Product save(Product product) {
        Product toSave = new Product(
                product.getId(),
                product.getUuid(),
                product.getName(),
                product.getWeight()
        );
        if (Objects.isNull(toSave.getId())) {
            toSave.setId(new Random().nextLong(_000_000L));
        }
        Optional<Product> existingProject = findById(product.getId());
        existingProject.ifPresent(products::remove);
        products.add(toSave);
        return toSave;
    }

    public void delete(Product product) {
        products.remove(product);
    }
}
```

Aggiunta del Livello di Servizio

Cos'è: Il livello di servizio (o business) si occupa della logica di business dell'applicazione. Funziona come **mediatore tra il livello di presentazione e il livello di persistenza.** Questo livello gestisce il flusso di dati e le operazioni aziendali, come le regole di validazione e il calcolo di dati.

```
@Service
```

```
public class ProductService {
    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public Optional<Product> findByUuid(String uuid) {
        return productRepository.findByUuid(uuid);
    }

    public Iterable<Product> findAll() {
        return productRepository.findAll();
    }

    public Product save(Product product) {
        return productRepository.save(product);
    }

    public void delete(Product product) {
        productRepository.delete(product);
    }
}
```

Aggiunta del Livello di Presentazione

Cos'è: Il ***livello di presentazione*** **è la parte dell'applicazione che interagisce con l'utente.** Include tutto ciò che è visibile all'utente finale e gestisce l'interfaccia utente, le interazioni e la visualizzazione dei dati.

Passando al livello di presentazione, aggiungeremo un’interfaccia simile nel package com.nbicocchi.controller:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping("/{uuid}")
    public Product findByUuid(@PathVariable String uuid) {
        return productService.findByUuid(uuid).orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
    }

    @GetMapping
    public Iterable<Product> findAll() {
        return productService.findAll();
    }

    @PostMapping
    public Product create(@RequestBody Product product) {
        return productService.save(product);
    }

    @PutMapping("/{uuid}")
    public Product update(@PathVariable String uuid, @RequestBody Product product) {
        Optional<Product> optionalProject = productService.findByUuid(uuid);
        optionalProject.orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
        product.setId(optionalProject.get().getId());
        return productService.save(product);
    }

    @DeleteMapping("/{uuid}")
    public void delete(@PathVariable String uuid) {
```

```
        Optional<Product> optionalProject = productService.findByUuid(uuid);
        optionalProject.orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND));
        productService.delete(optionalProject.get());
```

Interfacce ApplicationRunner e CommandLineRunner

In Spring Boot, ***CommandLineRunner e ApplicationRunner*** sono due interfacce che consentono di eseguire codice all’avvio di un’applicazione Spring Boot. Sono tipicamente utilizzate per eseguire operazioni di inizializzazione prima che l’applicazione inizi a elaborare le richieste.

```
@Log
@SpringBootApplication
public class App implements ApplicationRunner {
    ProductRepository productRepository;

    public App(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public static void main(final String... args) {
        SpringApplication.run(App.class, args);
    }

    @Override
    public void run(ApplicationArguments args) {
        productRepository.save(new Product("171f5df0-b213-4a40-8ae6-fe82239ab660", "Laptop", 2.2));
        productRepository.save(new Product("f89b6577-3705-414f-8b01-41c091abb5e0", "Bike", 5.5));
        productRepository.save(new Product("b1f4748a-f3cd-4fc3-be58-38316afe1574", "Shirt", 0.2));

        Iterable<Product> products = productRepository.findAll();
        for (Product product : products) {
            log.info(product.toString());
        }
    }
}
```

Test del Microservizio Product

Il resto della sezione rimane invariato poiché i comandi e le risposte JSON non necessitano di traduzione.

3. Maven

Struttura della directory del progetto

Poiché il nostro progetto si basa su Maven, si segue il layout standard delle directory di Maven che è il seguente.

- **src/main/java:** Questa directory contiene il *codice sorgente Java dell’applicazione che stiamo sviluppando*; tutti i nostri package e classi vengono inseriti in questa directory.
- **src/main/resources:** Questa directory contiene tutti gli artefatti non Java utilizzati dalla nostra applicazione; qui possiamo inserire file di configurazione e proprietà.
- **src/test/java e src/test/resources:** Queste directory contengono il codice sorgente dei test e le relative risorse, in modo simile alle directory src/main/*.

Configurazione del progetto (pom.xml)

Per prima cosa, abbiamo una parte fondamentale del pom.xml, ovvero il parent:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.4.3</version>
    <relativePath/> <!-- cerca il parent nel repository -->
</parent>
```

“Un "parent" in un `pom.xml` è un modo per definire che un progetto è un sotto-progetto (o modulo) di un altro progetto. Questo parent può definire proprietà condivise, dipendenze, plugin e altre configurazioni che il progetto figlio può ereditare.

Stiamo utilizzando il Boot parent, che definisce dipendenze, plugin e proprietà che il nostro progetto erediterà. Questo semplifica notevolmente la configurazione, evitando di doverle definire tutte manualmente.

Successivamente, definiamo le informazioni identificative di base del nostro progetto: groupId, artifactId, version e packaging:

```
<groupId>com.nbicocchi</groupId>
<artifactId>product-service-no-db</artifactId>
<version>0.0.1-SNAPSHOT</version>
<description>Demo product per Spring Boot</description>
```

Poi abbiamo alcune proprietà che sovrascrivono quelle definite nel parent:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>21</java.version>
</properties>
```

Ora arriviamo alla sezione dependencies ricordando *The Twelve Factors #2: la gestione delle dipendenze deve avvenire in modo centralizzato dividendo la configurazione di esse dal codice sorgente* → **Dipendenze**: Spring Boot offre il supporto per il dependency injection, che consente di gestire le dipendenze delle componenti dell’applicazione in modo centralizzato.

Questa sezione del pom.xml configura le librerie necessarie per sviluppare un’applicazione web Spring Boot.

```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
      </dependency>
</dependencies>
```

- **Lombok**: semplifica il codice generando automaticamente metodi comuni in Java tramite annotazioni, riducendo la ridondanza e migliorando la manutenibilità.
- **Spring Boot Starter Web**: consente di sviluppare applicazioni web utilizzando Spring MVC. Include librerie essenziali per creare API REST, gestire richieste HTTP e rendere contenuti web. Inoltre, fornisce un server incorporato (es. Tomcat) per eseguire l’applicazione senza configurazioni esterne.

- **Spring Boot Starter Actuator:** aggiunge funzionalità di monitoraggio, metriche e diagnostica per l’applicazione. Espone vari endpoint per monitorare lo stato dell’applicazione (es. /health per i controlli di stato e /metrics per le metriche delle prestazioni).
- **Spring Boot Starter Test:** include librerie essenziali per testare applicazioni Spring Boot, come JUnit per i test unitari, Mockito per il mocking degli oggetti e il framework di test di Spring per i test di integrazione.

Esecuzione del progetto

Esecuzione con il plugin Spring Boot Maven

Il **plugin Spring Boot Maven fornisce funzionalità per costruire ed eseguire l’applicazione.** Deve essere incluso esplicitamente nella **sezione build > plugins del file pom.xml:**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Ora possiamo eseguire l’applicazione con il comando:

```
mvn spring-boot:run
```

Nota: eseguire l’applicazione in questo modo non è consigliato in produzione, perché:

- Il codice sorgente deve essere presente sul server.
- L’avvio a freddo è lento poiché deve scaricare dipendenze, compilare l’app e poi eseguire l’applicazione.

In produzione, è meglio eseguire un artefatto preconfezionato come un file JAR o un container Docker.

Esecuzione come JAR

Un normale file JAR non può essere eseguito direttamente, poiché non include tutte le dipendenze necessarie.

Il plugin Spring Boot Maven crea un **fat jar** contenente tutti i file della classe dell’applicazione e le dipendenze necessarie, rendendolo un’applicazione autonoma.

```
mvn clean package -Dmaven.skip.test=true
```

Eseguiamo quindi il JAR generato:

```
java -jar target/product-service-no-db-0.0.1-SNAPSHOT.jar
```

Possiamo fornire configurazioni di runtime con argomenti a riga di comando:

```
java -jar -Dserver.port=8082 target/product-service-no-db-0.0.1-SNAPSHOT.jar
```

O con variabili d’ambiente:

```
SERVER_PORT=8182 java -jar target/product-service-no-db-0.0.1-SNAPSHOT.jar
```

O entrambi:

```
SERVER_PORT=8082 java -jar -Dserver.port=8083
target/product-service-no-db-0.0.1-SNAPSHOT.jar
```

Riduzione del Codice Boilerplate con Lombok

In questa guida, esploreremo come ridurre significativamente la quantità di codice boilerplate in Java. Sebbene Java sia un linguaggio potente, spesso diventa verboso quando si gestiscono operazioni di routine o si aderisce alle pratiche del framework. Questa verbosità non aggiunge molto valore commerciale, ed è qui che entra in gioco Project Lombok per migliorare la produttività.

Dipendenze del Progetto

Per includere Lombok nel tuo progetto, puoi scegliere tra due approcci:

- Se stai creando un nuovo progetto, aggiungi semplicemente la dipendenza tramite il Spring Boot Initializr
- Se desideri aggiungere Lombok a un progetto esistente (e stai usando Maven), includi la seguente dipendenza nel tuo pom.xml

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

Esempio Pratico

Supponiamo di dover definire un evento in una classe. Specificamente, il nostro evento sarà caratterizzato da:

- Il tipo di evento (ad es., crea, elimina o aggiorna)
- Una chiave che identifica i dati (ad es., un ID messaggio)
- Un elemento dati (i dati reali nell’evento)
- Un timestamp che indica quando si è verificato l’evento

Per implementare completamente costruttori, getter, setter e altri metodi, finiremmo con qualcosa del genere:

```
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    private Type eventType;
    private K key;
    private T data;
    private ZonedDateTime eventCreatedAt;

    public Event() {}

    public Event(Type eventType, K key, T data) {
        this.eventType = eventType;
        this.key = key;
        this.data = data;
        this.eventCreatedAt = ZonedDateTime.now();
    }

    public Type getEventType() {
```

```
        return eventType;
    }

    public K getKey() {
        return key;
    }

    public T getData() {
        return data;
    }

    public ZonedDateTime getEventCreatedAt() {
        return eventCreatedAt;
    }

    @Override
    public String toString() {
        return "Event{" + "eventType=" + eventType + ", key=" + key + ", data=" + data + ",
eventCreatedAt=" + eventCreatedAt + '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Event<?, ?> event = (Event<?, ?>) o;
        return eventType == event.eventType &&
                Objects.equals(key, event.key) &&
                Objects.equals(data, event.data) &&
                Objects.equals(eventCreatedAt, event.eventCreatedAt);
    }

    @Override
    public int hashCode() {
        return Objects.hash(eventType, key, data, eventCreatedAt);
    }
}
```

Creare un oggetto random di tipo `Event<String, Integer>` appare così:

```
int index = RANDOM.nextInt(Event.Type.class.getEnumConstants().length);
Event<String, Integer> event = new Event(
    Event.Type.class.getEnumConstants()[index],
    UUID.randomUUID().toString(),
    RANDOM.nextInt(100)
);
```

Questo processo è noioso e ingombrante, influenzando la chiarezza. Gli IDE possono generare automaticamente codice per te, ma ciò non risolve ancora il problema sottostante del codice verboso. Entra in gioco Lombok!

`@NoArgsConstructor, @AllArgsConstructor e @RequiredArgsConstructor`

Lombok fornisce annotazioni per evitare di scrivere manualmente i costruttori:

- **`@NoArgsConstructor`** genera un costruttore senza argomenti. Se non è possibile a causa di campi finali, il compilatore genererà un errore a meno che non usi @NoArgsConstructor(force = true), che inizializza i campi finali con valori predefiniti (0, false o null). I campi con vincoli come @NonNull non verranno controllati, quindi assicurati che quei campi siano inizializzati successivamente.
- **`@AllArgsConstructor`** genera un costruttore con un parametro per ogni campo. I campi contrassegnati con @NonNull includono controlli per null.
- **`@RequiredArgsConstructor`** genera un costruttore per tutti i campi finali non inizializzati e i campi contrassegnati con @NonNull. Un controllo per null viene aggiunto per i campi @NonNull.

Ecco come appare la nostra classe dopo aver utilizzato i costruttori di Lombok:

```
@NoArgsConstructor
@AllArgsConstructor
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    private Type eventType;
    private K key;
    private T data;
    private ZonedDateTime eventCreatedAt;

    public Type getEventType() {
        return eventType;
    }

    public K getKey() {
        return key;
    }

    public T getData() {
        return data;
    }

    public ZonedDateTime getEventCreatedAt() {
        return eventCreatedAt;
    }
```

```
@Override
    public String toString() {
        return "Event{" + "eventType=" + eventType + ", key=" +
key + ", data=" + data + ", eventCreatedAt=" + eventCreatedAt +
'}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Event<?, ?> event = (Event<?, ?>) o;
        return eventType == event.eventType &&
                Objects.equals(key, event.key) &&
                Objects.equals(data, event.data) &&
                Objects.equals(eventCreatedAt,
event.eventCreatedAt);
    }

    @Override
    public int hashCode() {
        return Objects.hash(eventType, key, data, eventCreatedAt);
    }
 }
```

Poiché non abbiamo campi `@NonNull` o finali, `@RequiredArgsConstructor` non è necessario.

`@Getter e @Setter`

Le annotazioni `@Getter e @Setter` generano automaticamente getter e setter per tutti i campi, semplificando ulteriormente il codice:

```
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    private Type eventType;
    private K key;
    private T data;
    private ZonedDateTime eventCreatedAt;

    @Override
    public String toString() {
        return "Event{" + "eventType=" + eventType + ", key=" + key + ", data=" + data + ", eventCreatedAt=" + eventCreatedAt + '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return false;
        Event<?, ?> event = (Event<?, ?>) o;
        return eventType == event.eventType &&
                Objects.equals(key, event.key) &&
                Objects.equals(data, event.data) &&
                Objects.equals(eventCreatedAt, event.eventCreatedAt);
    }

    @Override
    public int hashCode() {
        return Objects.hash(eventType, key, data, eventCreatedAt);
    }
}
```

@ToString e @EqualsAndHashCode

Lombok può anche generare automaticamente i metodi toString(), equals() e hashCode():

```
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
@ToString
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    private Type eventType;
    private K key;
    private T data;
    private ZonedDateTime eventCreatedAt;
}
```

Ora la nostra classe consiste solo di campi e annotazioni Lombok!

@Data

Per una maggiore semplicità, l’annotazione @Data raggruppa @ToString, @EqualsAndHashCode, @Getter, `@Setter` e `@RequiredArgsConstructor`. La nostra classe può ora essere riscritta come:

```
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    @NonNull private Type eventType;
    private K key;
    private T data;
    private ZonedDateTime eventCreatedAt;
}
```

Se introduciamo campi finali o @NonNull, i costruttori appropriati vengono generati automaticamente da Lombok.

@Builder e @Builder.Default

L’annotazione @Builder consente di creare oggetti in modo più leggibile, utile soprattutto in classi complesse:

```
Event event = Event.builder()
  .eventType(Event.Type.CREATE)
  .key(UUID.randomUUID().toString())
  .data(RANDOM.nextInt(100))
  .eventCreatedAt(ZonedDateTime.now())
  .build();
```

Puoi persino impostare valori predefiniti con @Builder.Default:

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class Event<K, T> {
    public enum Type {CREATE, DELETE, UPDATE}
    @Builder.Default private Type eventType = Type.CREATE;
    @Builder.Default private K key = UUID.randomUUID().toString();
    private T data;
    @Builder.Default private ZonedDateTime eventCreatedAt = ZonedDateTime.now();
}
```

In questo modo, la creazione di oggetti può essere semplificata e i valori predefiniti saranno utilizzati a meno che non vengano sovrascritti:

```
Event event = Event.builder()
  .data(RANDOM.nextInt(100))
  .build();
```

Dependency Injection

Cos’è l’Inversion of Control?

L**’Inversion of Contro**l è un principio nella programmazione software che trasferisce il controllo di porzioni di un programma a un contenitore o un framework esterno. Lo utilizziamo più spesso nel contesto della programmazione orientata agli oggetti.

A differenza dell’OOP tradizionale, in cui il nostro codice personalizzato chiama una libreria, **l’IoC consente a un framework di controllare il flusso di un programma e di chiamare il nostro codice personalizzato**. Per consentire ciò, i framework utilizzano astrazioni con comportamenti aggiuntivi integrati. Se vogliamo aggiungere il nostro comportamento, dobbiamo estendere le classi del framework o collegare le nostre classi.

Possiamo ottenere l’Inversione di Controllo attraverso vari meccanismi:

- Pattern Strategy
- Pattern Service Locator
- Pattern Factory
- Pattern di Iniezione delle Dipendenze

Il Contenitore IoC di Spring

L’interfaccia **org.springframework.context.ApplicationContext** rappresenta il contenitore IoC di Spring che è **responsabile dell’istanza, della configurazione e dell’assemblaggio dei bean.** Ti consente di esprimere gli oggetti che compongono la tua applicazione e le ricche interdipendenze tra tali oggetti. Diverse implementazioni dell’interfaccia ApplicationContext sono fornite di serie con Spring.

Il seguente diagramma offre una panoramica ad alto livello di come funziona Spring. Le classi della tua applicazione sono combinate con i metadati di configurazione in modo tale che, dopo che l’ ApplicationContext è stato creato e inizializzato, avrai un sistema o un’applicazione completamente configurata ed eseguibile.

Iniezione delle Dipendenze

Classe A ha una dipendenza dalla Classe B quando interagisce con essa in qualche modo.

Un’opzione è che la Classe A si assuma la responsabilità di istanziare B da sola:

```
public class A {
    private B bDependency;
```

```
    public A() {
        bDependency = new B();
    }
}
```

In alternativa, questa responsabilità può essere esterna, il che significa che la dipendenza proviene dall’esterno:

```
public class A {
    private B bDependency;

    public A(B bDependency) {
        this.bDependency = bDependency;
    }
}
```

L’iniezione è semplicemente il processo di iniezione della dipendenza B nell’oggetto di tipo A. Poiché l’istanza della dipendenza B non viene più effettuata in A, quella responsabilità appartiene ora al framework. Separare la responsabilità di istanziare una classe dalla logica in quella classe è un concetto molto utile:

- **Porta a un sistema più debolmente accoppiato e aggiunge flessibilità** (le dipendenze possono essere scambiate a runtime).
- Utile sia nell’architettura dell’applicazione, sia nei test, perché l’iniezione delle dipendenze facilita la commutazione tra diverse implementazioni della dipendenza. Ad esempio, possiamo passare un mock di una dipendenza anziché un oggetto dipendenza completo.

Tipi di Iniezione delle Dipendenze

Ci sono tre modi principali per iniettare dipendenze:

- ***Iniezione via costruttore***
- ***Iniezione via setter***
- ***Iniezione via campo***

Iniezione via Costruttore

In questo caso, iniettiamo le dipendenze in una classe tramite gli argomenti del suo costruttore (ogni argomento rappresenta una dipendenza che Spring inietterà automaticamente).

Diamo un’occhiata al nostro *ProductService* in cui un oggetto della classe *ProductRepository* viene iniettato tramite un argomento del costruttore:

```
@Service
public class ProductService {
   private final ProductRepository productRepository;

   @Autowired
   public ProductService(ProductRepository productRepository) {
      this.productRepository = productRepository;
   }

   public Optional<Product> findByUuid(String uuid) {
      return productRepository.findByUuid(uuid);
   }

   public Iterable<Product> findAll() {
      return productRepository.findAll();
```

```
    }

    public Product save(Product product) {
        return productRepository.save(product);
    }

    public void delete(Product product) {
        productRepository.delete(product);
    }
}
```

Nota che poiché abbiamo un solo costruttore, l’annotazione @Autowired_ è facoltativa. Se definiamo più di un costruttore e vogliamo che uno di essi inietti dipendenze durante la creazione del bean, allora dobbiamo aggiungere @Autowired_ al costruttore richiesto.

Iniezione via Setter

Nell’iniezione basata su setter, iniettiamo le dipendenze utilizzando i metodi setter delle dipendenze richieste dichiarate come campi: * dipendenze opzionali * cambiamenti delle dipendenze a runtime

```
@Service
public class ProductService {
    private ProductRepository productRepository;

    @Autowired
    public void setProductRepository(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public Optional<Product> findById(Long id) {
        return productRepository.findById(id);
    }

    public Iterable<Product> findAll() {
        return productRepository.findAll();
    }

    public Product save(Product product) {
        return productRepository.save(product);
    }

    public void delete(Product product) {
        productRepository.delete(product);
    }
}
```

Iniezione via Campo

Nell’iniezione delle dipendenze tramite campo **iniettiamo le dipendenze utilizzando l’annotazione _@Autowired_ direttamente sui campi**. Questo approccio non è più raccomandato.

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;

    public Optional<Product> findById(Long id) {
        return productRepository.findById(id);
    }

    public Iterable<Product> findAll() {
        return productRepository.findAll();
    }
```

```
    public Product save(Product product) {
        return productRepository.save(product);
    }

    public void delete(Product product) {
        productRepository.delete(product);
    }
}
```

Le Annotazioni *@Qualifier_ e* @Primary_

`@Qualifier` e `@Primary` sono annotazioni utilizzate per gestire l’iniezione delle dipendenze quando ci sono più bean dello stesso tipo, consentendo un maggiore controllo su quale specifico bean viene scelto dal framework.

`@Qualifier` viene utilizzato per disambiguare tra più bean dello stesso tipo. Quando Spring trova più bean che corrispondono a una dipendenza, genera un errore a meno che non specifichi quale utilizzare. Utilizzando `@Qualifier`, puoi indicare il bean specifico che desideri iniettare.

Supponiamo di avere due implementazioni di un’interfaccia `Vehicle`:

```
public interface Vehicle {
    void start();
}

@Component
public class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("L'auto sta
partendo");
    }
}
```

```
@Component
public class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("La moto sta
partendo");
    }
}
```

Se desideri iniettare il bean `Car` in una classe, puoi utilizzare `@Qualifier`:

```
@Component
public class Driver {
    private final Vehicle vehicle;

    @Autowired
    public Driver(@Qualifier("car") Vehicle vehicle)
    {
        this.vehicle = vehicle;
    }
    public void drive() {
        vehicle.start();
    }
}
```

`@Primary` viene utilizzato per specificare quale bean deve essere preferito quando non è specificato alcun `@Qualifier`. Se hai più bean dello stesso tipo e uno di essi è contrassegnato come `@Primary`, Spring utilizzerà quel bean per impostazione predefinita, a meno che non venga indicato diversamente.

```java
@Component
@Primary
public class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("L'auto sta partendo");
    }
}
```

```java
@Component
public class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("La moto sta partendo");
    }
}
```

Qui, `Car` è contrassegnata con `@Primary`, quindi se inietti un `Vehicle` senza specificare un `@Qualifier`, Spring sceglierà automaticamente il bean `Car`:

```java
@Component
public class Driver {
    private final Vehicle vehicle;

    @Autowired
    public Driver(Vehicle vehicle) { // Non è necessario @Qualifier
        this.vehicle = vehicle;
    }

    public void drive() {
        vehicle.start();
    }
}
```

Migliori Pratiche Generali per l’Iniezione delle Dipendenze

- Preferire l’iniezione via costruttore:
    - L’iniezione via costruttore rende chiare le dipendenze dell’oggetto e forza l’immutabilità. Facilita anche i test unitari poiché le dipendenze possono essere simulate e iniettate attraverso il costruttore.
- Gestire le dipendenze opzionali con l’iniezione via setter:
    - L’iniezione via setter è ideale quando alcune dipendenze sono opzionali. L’iniezione via costruttore dovrebbe essere utilizzata per le dipendenze obbligatorie, mentre l’iniezione via setter può essere utilizzata per quelle che potrebbero non essere sempre fornite.
- Utilizzare @Qualifier e @Primary per più bean che implementano la stessa interfaccia:
    - Quando hai più bean dello stesso tipo, utilizza @Qualifier per specificare quale iniettare, oppure contrassegna uno di essi con @Primary per renderlo il predefinito.
- Evitare le dipendenze circolari:
    - Le dipendenze circolari possono causare problemi con l’iniezione, specialmente con l’iniezione per costruttore. Spring genererà un errore se rileva dipendenze circolari a runtime, quindi progetta i tuoi bean per evitarlo.

Dipendenze Circolari in Spring

Una dipendenza circolare si verifica quando due o più bean in un’applicazione Spring dipendono l’uno dall’altro, creando un ciclo che Spring non può risolvere automaticamente. Questa situazione si presenta tipicamente quando *si utilizza l’iniezione delle dipendenze basata su costruttore*, portando a un `BeanCurrentlyInCreationException`.

Considera i seguenti due bean Spring:

```java
@Component
public class A {
    private final B b;

    @Autowired
    public A(B b) {
        this.b = b;
    }
}
@Component
public class B {
    private final A a;

    @Autowired
    public B(A a) {
        this.a = a;
    }
}
```

Qui, `A` dipende da `B`, e `B` dipende da `A`, creando una dipendenza circolare.

Come Spring Gestisce le Dipendenze Circolari

Spring può risolvere le dipendenze circolari **solo se almeno una delle dipendenze è iniettata tramite setter o iniezione di campo**, consentendo al contenitore di creare prima un bean e iniettare l’altro in seguito.

Iniezione via Setter (Raccomandata)

```java
@Component
public class A {
    private B b;

    @Autowired
    public void setB(B b) {
        this.b = b;
    }
}
```

```java
@Component
public class B {
    private A a;

    @Autowired
    public void setA(A a) {
        this.a = a;
    }
}
```

Annotazione @Lazy

Un altro approccio consiste nel contrassegnare una delle dipendenze come @Lazy, ritardando la sua inizializzazione fino a quando non è effettivamente necessaria.

```java
@Component
public class A {
    private final B b;

    @Autowired
    public A(@Lazy B b) {
        this.b = b;
    }
}
```

Questo indica a Spring di iniettare B in A solo quando è effettivamente necessaria, rompendo il ciclo.

5. Definizione dei Bean, Scansione dei Componenti e Annotazioni dei Bean

Quando l’applicazione Spring viene inizializzata, l’**Application Context** inizia ad istanziare degli oggetti chiamati *beans* per metterli a disposizione dell’ecosistema.

Contribuire con Bean al Contesto

I bean possono essere generati tramite diversi metodi. L’annotazione stereotipata più semplice che possiamo utilizzare è @Component_. Fondamentalmente, durante il processo di bootstrapping, Spring scansionerà per eventuali classi annotate con @Component_ e le istanzierà come bean.

```
@Component
public class ProductRepository {
    // ...
}
```

Un’altra opzione comune è creare una classe di configurazione in cui definiamo manualmente un bean. CLASSE CHE CONTRIBUISCE CON DEI BEAN ALL’ECOSISTEMA.

```
@Configuration
public class PersistenceConfig {

    @Bean
    public ProductRepository productRepository() {
        return new ProductRepository();
    }
}
```

L’annotazione @Configuration_ indica a Spring che questa classe deve essere elaborata dal contenitore Spring perché contribuirà con definizioni di bean. Ovviamente, l’annotazione @Bean_ è una di quelle definizioni di bean. In questo caso, si tratta di un bean nominato productRepository, poiché questo è il nome del metodo.

Di default, Spring Boot carica tutte le classi annotate con @Component_ e @Configuration_ che si trovano nello stesso pacchetto della classe principale (e nei sub-pacchetti). Questo è definito dall’annotazione _@ComponentScan_ e può essere modificato manualmente.

NOTA: L’@SpringBootApplication_ include @ComponentScan (In Intellij: Visualizza > Salta alla sorgente). Ecco perché di solito non utilizziamo @ComponentScan_ esplicitamente.

Annotazioni Stereotipate

Oltre a @Component_, ci sono alcune altre annotazioni stereotipate che utilizzano @Component_ sotto il cofano e portano semplicemente un ulteriore livello di semantica.

Ad esempio, nella classe ProductRepository possiamo sostituire @Component_ con l’annotazione @Repository_:

```
@Repository
public class ProductRepository {
    // ...
}
```

Tecnicamente non cambia nulla, ma ciò si adatta meglio alla semantica esatta di questo particolare bean, poiché è effettivamente un repository. Allo stesso modo, possiamo usare l’annotazione @Service_ e @RestController_:

```
@Service
public class ProductService {
```

```
    // ...
}

@RestController
@RequestMapping("/products")
public class ProductController {
    // ...
}
```

Ciclo di Vita del Bean

Fondamentalmente, il ciclo di vita di un bean Spring consiste in 3 fasi:

- fase di inizializzazione
- fase di utilizzo
- fase di distruzione

Ci concentriamo sulle fasi di inizializzazione e distruzione, poiché sono le più interessanti dal punto di vista dell’iniezione delle dipendenze. Il nostro obiettivo qui è comprendere il ciclo di vita, ma anche gli hook nel framework collegati a quel ciclo di vita.

@Component

```
@Component
public class BeanA {
    private static final Logger log = LoggerFactory.getLogger(BeanA.class);

    @PreDestroy
    public void preDestroy() {
        log.info("Il metodo @PreDestroy è stato chiamato.");
    }

    @PostConstruct
    public void postConstruct() {
        log.info("Il metodo @PostConstruct è stato chiamato.");
    }
}
```

@Bean

```
@Configuration
public class AppConfig {
    @Bean(initMethod="initialize", destroyMethod="destroy")
    public BeanB beanB() {
        return new BeanB();
    }
}
public class BeanB {
    private static final Logger log = LoggerFactory.getLogger(BeanB.class);

    public void initialize() {
        log.info("Il metodo custom initialize() è stato chiamato.");
    }

    public void destroy() {
        log.info("Il metodo custom destroy() è stato chiamato.");
    }
}
```

Scope dei Bean

In Spring, **singleton e prototype** sono i due scope di bean comuni che definiscono come i bean vengono creati e gestiti nel contenitore Spring.

```
@Configuration
public class AppConfig {
    @Bean(initMethod="initialize", destroyMethod="destroy")
    @Scope("singleton")
    public BeanB beanB() {
        return new BeanB();
    }

    @Bean(initMethod="initialize", destroyMethod="destroy")
    @Scope("prototype")
    public BeanC beanC() {
        return new BeanC();
    }
}
```

Scope Singleton (scope predefinito)

Nello scope singleton, Spring crea **solo un’istanza** di un bean per contenitore Spring. Questa istanza è condivisa tra tutte le parti dell’applicazione che richiedono il bean. Il bean viene creato quando il contenitore Spring è inizializzato (inizializzazione avido per impostazione predefinita) e resta in memoria per tutta la durata del ciclo di vita del contenitore.

Scope Prototype

Nello scope prototype, viene creata **una nuova istanza** del bean **ogni volta** che viene richiesta dal contenitore Spring. A differenza di singleton, il ciclo di vita di un bean prototype è di breve durata. Una volta che il bean è creato e consegnato dal contenitore, il contenitore non gestisce ulteriormente il ciclo di vita del bean (cioè non si occuperà della distruzione del bean).

Esempio

```
@Component
public class BeanD {
    BeanB beanB1;
    BeanB beanB2;
    BeanC beanC1;
    BeanC beanC2;

    public BeanD(BeanB beanB1, BeanB beanB2, BeanC beanC1, BeanC beanC2) {
        this.beanB1 = beanB1;
        this.beanB2 = beanB2;
        this.beanC1 = beanC1;
        this.beanC2 = beanC2;
        // Imposta un breakpoint qui e fai il debug
        // vedrai che b1 e b2 sono in effetti riferimenti allo stesso oggetto
        // mentre c1 e c2 sono riferimenti a oggetti diversi
    }
}
```