

Spring

Riccardo Cattaneo



Ambiente di Sviluppo

Per scrivere un programma java abbiamo bisogno di :

- Un programma che ci permetta di scrivere il codice, per iniziare può andar bene un semplice editor di testo come il blocco note (**NotePad**), esistono diversi ambienti di sviluppo più complessi come ad es. **Eclipse STS** (**S**pring **T**ool **S**uite) che sono open source e gratuiti;
- Il **Java Development Kit**, versione **Standard Edition (JDK)** che è scaricabile gratuitamente dal sito proprietario del linguaggio (Oracle), per iniziare abbiamo quindi bisogno di un compilatore e di una JVM. Il JDK ci offre tutto l'occorrente per lavorare in modo completo.

Installare JDK

Oracle, che dal 2010 (anno in cui ha acquistato Sun) è proprietaria del marchio Java, supporta il Java Development Kit su molteplici architetture e sistemi operativi: tutte le versioni di Windows, le versioni di Windows Server; Mac OS X; Linux, RedHat, Ubuntu ecc.

Se utilizzate uno di questi sistemi operativi vi basterà aprire con il vostro browser preferito l'URL :

<https://www.oracle.com/java/technologies/javase-downloads.html>

Java SE Downloads

Java Platform, Standard Edition

Java SE 14

Java SE 14.0.1 is the latest release for the Java SE Platform

- [Documentation](#)
- [Installation Instructions](#)
- [Release Notes](#)
- [Oracle License](#)
 - [Binary License](#)
 - [Documentation License](#)
- [Java SE Licensing Information User Manual](#)
 - [Includes Third Party Licenses](#)
- [Certified System Configurations](#)
- [Readme](#)

Oracle JDK



JDK Download










Documentation Download



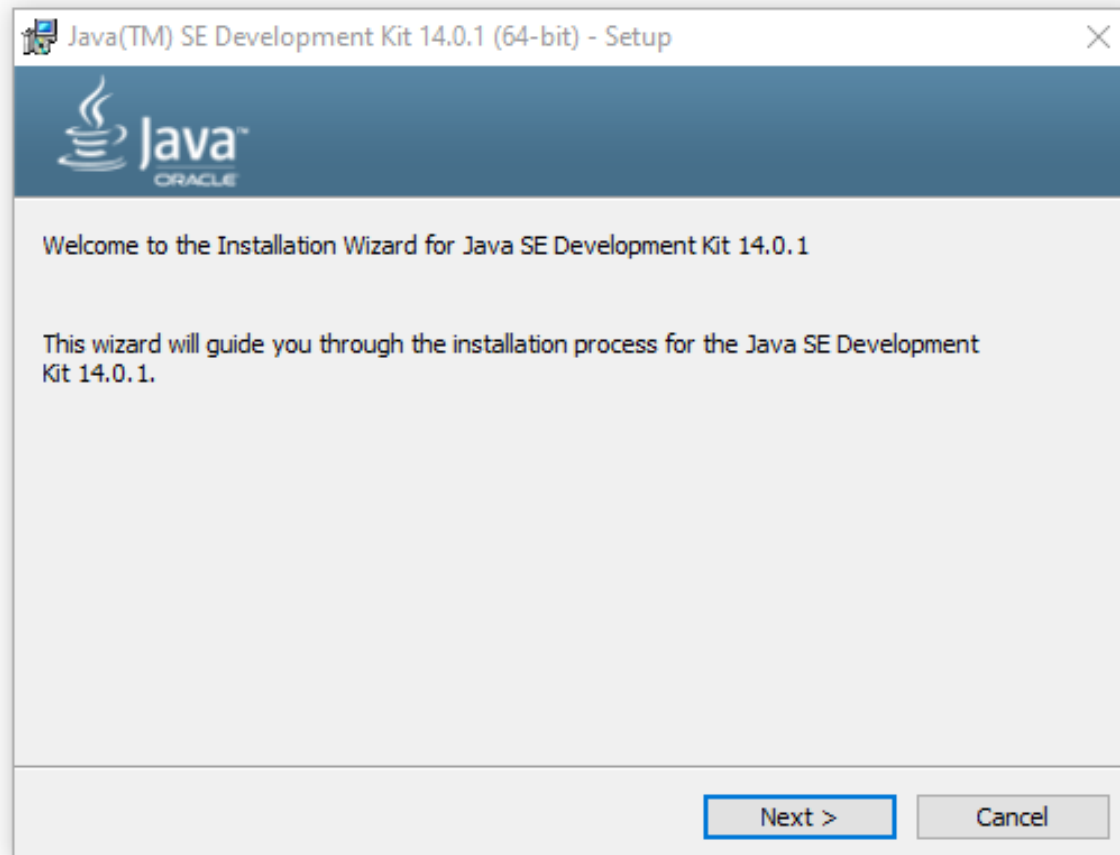
Java SE Development Kit 14

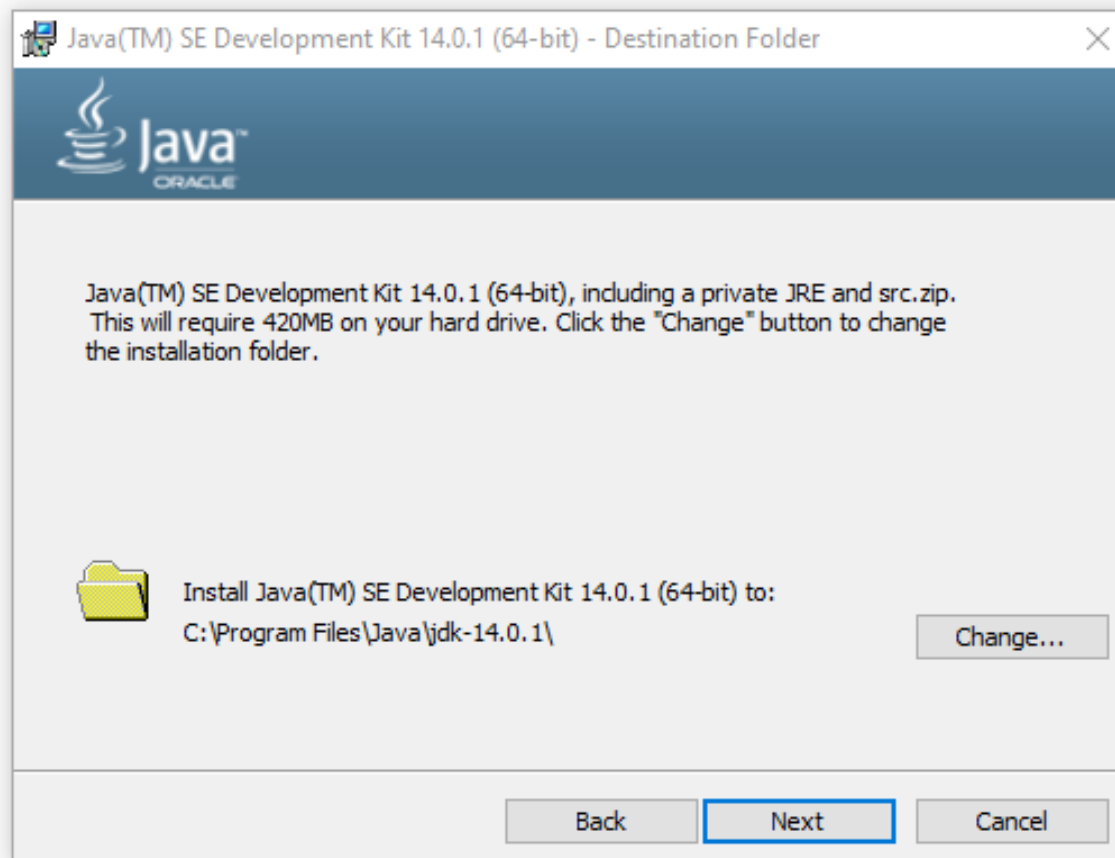
This software is licensed under the [Oracle Technology Network License Agreement for Oracle Java SE](#)

Product / File Description	File Size	Download
Linux Debian Package	157.92 MB	 jdk-14.0.1_linux-x64_bin.deb
Linux RPM Package	165.04 MB	 jdk-14.0.1_linux-x64_bin.rpm
Linux Compressed Archive	182.04 MB	 jdk-14.0.1_linux-x64_bin.tar.gz
macOS Installer	175.77 MB	 jdk-14.0.1_osx-x64_bin.dmg
macOS Compressed Archive	176.19 MB	 jdk-14.0.1_osx-x64_bin.tar.gz
Windows x64 Installer	162.07 MB	 jdk-14.0.1_windows-x64_bin.exe
Windows x64 Compressed Archive	181.53 MB	 jdk-14.0.1_windows-x64_bin.zip

La procedura di installazione è totalmente automatica e sarà sufficiente accettare i default del sistema di installazione.

Ad esempio l'installazione su Windows prevede pochi semplici passi: la selezione delle componenti da installare e la scelta del path.







Installazione Eclipse STS

Per poter sviluppare un'applicazione Spring abbiamo bisogno del Tool di Sviluppo. La scelta fatta è stata quella di utilizzare Lo Spring Tools versione 4 basato su eclipse.

Per scaricarlo è sufficiente recarsi al seguente indirizzo web, scegliere la propria piattaforma e scaricarlo :

<https://spring.io/tools>

Spring Tools 4

Spring Tools 4 is the next generation of Spring tooling for your favorite coding environment. Largely rebuilt from scratch, it provides world-class support for developing Spring-based enterprise applications, whether you prefer Eclipse, Visual Studio Code, or Theia IDE.

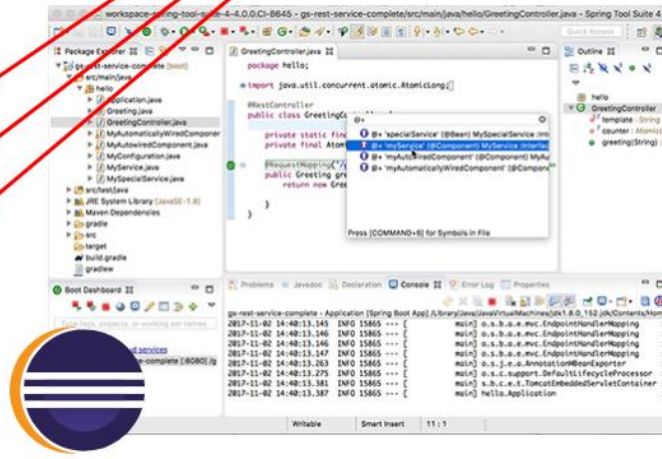
Spring Tools 4 for Eclipse

The all-new Spring Tool Suite 4.
Free. Open source.

LINUX 64-BIT

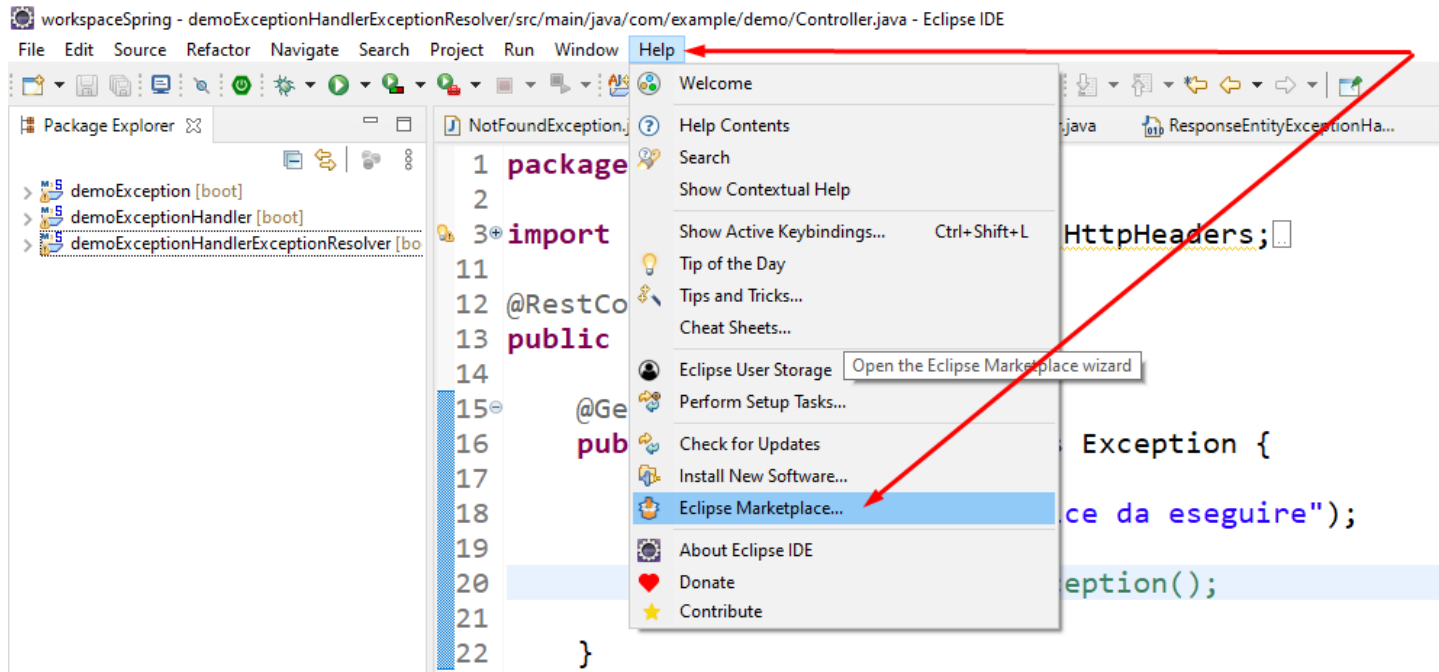
MACOS 64-BIT

WINDOWS 64-BIT



Installazione plugin su eclipse

Un'altra alternativa, se avete già eclipse installato, è quella di aggiungere un plugin al nostro IDE. Per aggiungerlo è sufficiente andare nel Marketplace di eclipse e cercare spring :



Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation. Press the "more info" link to learn more about a solution.

Search Recent Popular Favorites Installed Research at the Eclipse

Find: All Markets All Categories Go

Spring Tools 4 (aka Spring Tool Suite 4) 4.20.1.RELEASE

Spring Tools 4 is the next generation of Spring Boot tooling for your favorite coding environment. Largely rebuilt from scratch, it provides world-class support... [more info](#)

by VMware, EPL
[spring](#) [Spring IDE](#) [Cloud](#) [Spring Tool Suite STS](#)

★ 4055 Installs: 2,73M (24,563 last month) **Install**

JUnit Tools 4 Spring 1.2.4

Tool for helping the developers to write simple and maintainable unit tests with minimal effort on the tedious part and let them focus on the important part of... [more info](#)

by csorbazoli, Apache 2.0
[JUnit 5](#) [code generation](#) [test generator](#)

★ 7 Installs: 5,28K (695 last month) **Install**

Spring

Eclipse Marketplace

Confirm Selected Features

Press Confirm to continue with the installation. Or go back to choose more solutions to install.

Spring Tools 4 (aka Spring Tool Suite 4) 4.20.1.RELEASE <https://download.springsour>

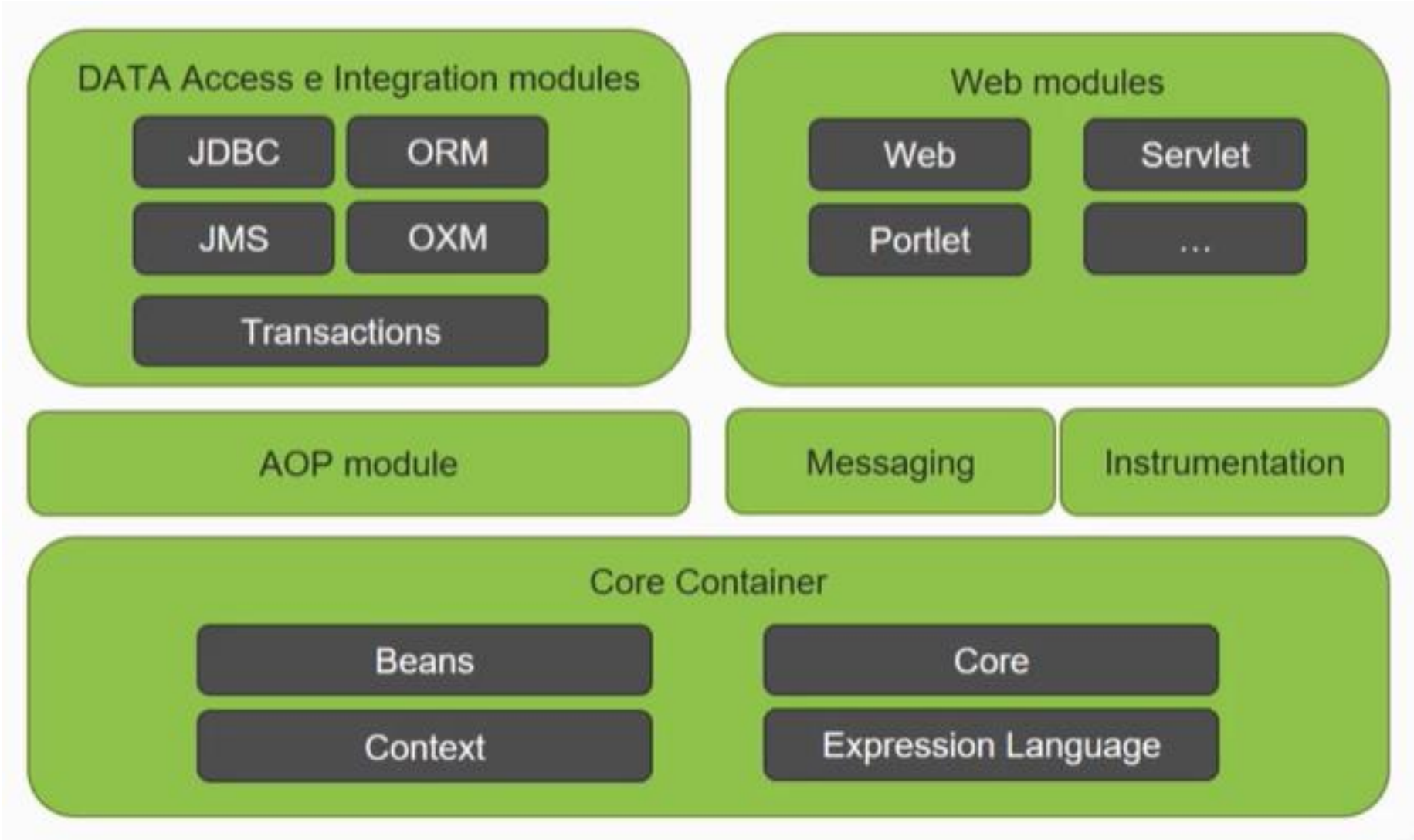
- ☒ Spring Boot Language Server Feature (required)
- ☒ Spring Tool Suite 4 Main Feature (required)
- ☐ Cloud Foundry Manifest Language Server Feature
- ☐ Concourse Pipeline Language Server Feature
- ☒ Spring IDE Boot Microservices Dash

Confirm > Finish Cancel

Introduzione

Cos'è Spring? Nato nel 2004, è un framework open source di java ed è il framework più popolare per applicazioni enterprise, ed è stato creato per ridurre la complessità di sviluppo. Ha una struttura modulare che permette di utilizzarlo interamente o in parte, senza stravolgere l'architettura del progetto.

Quali sono i moduli di Spring ?



Il Core è il motore del framework ed è obbligatorio. Gli altri moduli possono essere utilizzati facoltativamente a seconda di quello che dobbiamo fare.

In questo corso verrà utilizzato, tra i vari moduli che vedremo, il modulo Spring Web che è utilizzato per lo sviluppo di applicazioni web e si basa sul design pattern MVC. Un design pattern è uno schema architetturale volto a smaltire la complessità di progettazione e sviluppo di applicazioni enterprise.

Spring è stato sempre definito un framework «**leggero**» da utilizzare per lo sviluppo di applicazioni Java. Leggero non si riferisce al numero di classi che lo compongono o alle dimensioni del jar.

Leggero significa che è necessario apportare poche o nessuna modifica al codice della nostra applicazione per usufruire dei vantaggi delle componenti core di Spring.

Configurazione

La Configurazione di Spring può avvenire in 3 modi :

- File .xml
- Classi di configurazione java
- Annotation (metodo più usato ed è quello che utilizzeremo maggiormente)

IOC e DI

Per il funzionamento di Spring ci sono 2 concetti fondamentali che dobbiamo conoscere : **IOC** (Inversion Of Control) e **DI** (Dependency Injection).

IOC : è un principio architetturale basato sull'inversione del flusso del sistema, non è più il programmatore a doversi occupare di creare, inizializzare gli oggetti e chiamare metodi, ma lo farà il framework attraverso una specifica configurazione.

Ad esempio supponiamo di guidare l'automobile per andare a lavoro. Questo vuol dire che noi controlliamo l'auto e mentre guidiamo l'auto non possiamo fare altro.



Invertiamo lo scenario, invece di guidare l'auto noleggiamo un taxi. In questo caso non siamo noi a guidare la macchina, lasciamo che sia l'autista a farlo. In questo modo noi potremo concentrarci sul nostro lavoro principale.



DI : è una specifica implementazione dell'IOC, utilizzando questa tecnica, le dipendenze di una classe non saranno più inizializzate dalla classe stessa ma iniettate dall'esterno.

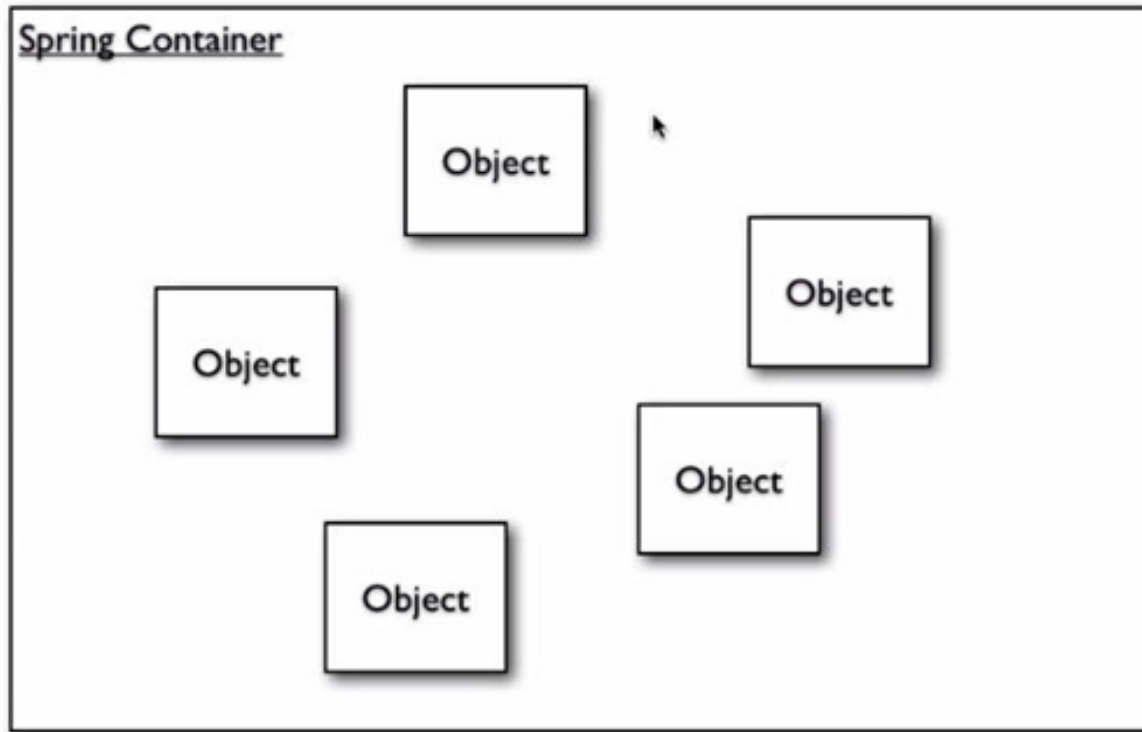
Possiamo quindi dire che **Spring implementa l'IOC attraverso la DI**. Per capirlo facciamo subito un esempio.... Passiamo alla pratica.

Il Container di Spring

La prima cosa da sapere è che Spring ha un «contenitore» che lavora dietro le quinte (di solito è una classe che implementa **ApplicationContext**) ed il compito di questo «contenitore» è quello di istanziare e configurare le classi (chiamate anche **Beans**).

Questo container per funzionare deve essere configurato, quindi in riferimento a quanto detto all'inizio della lezione, questa configurazione può essere fatta tramite file **xml**, **classi java** oppure **annotation**. I moduli Spring che consentono di implementare l'IoC sono **beans**, **core** e **context**.

A Spring Container



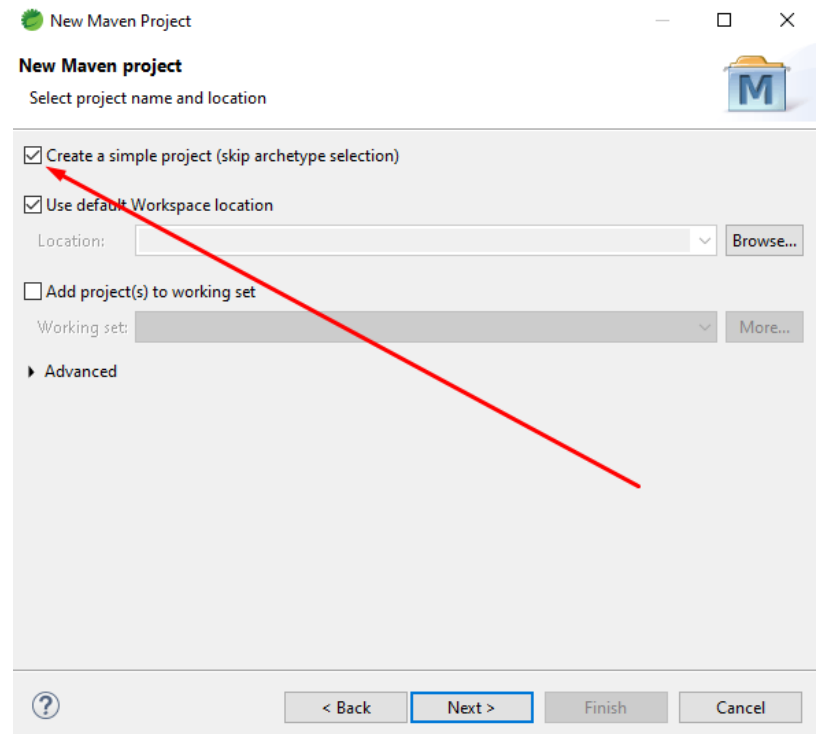
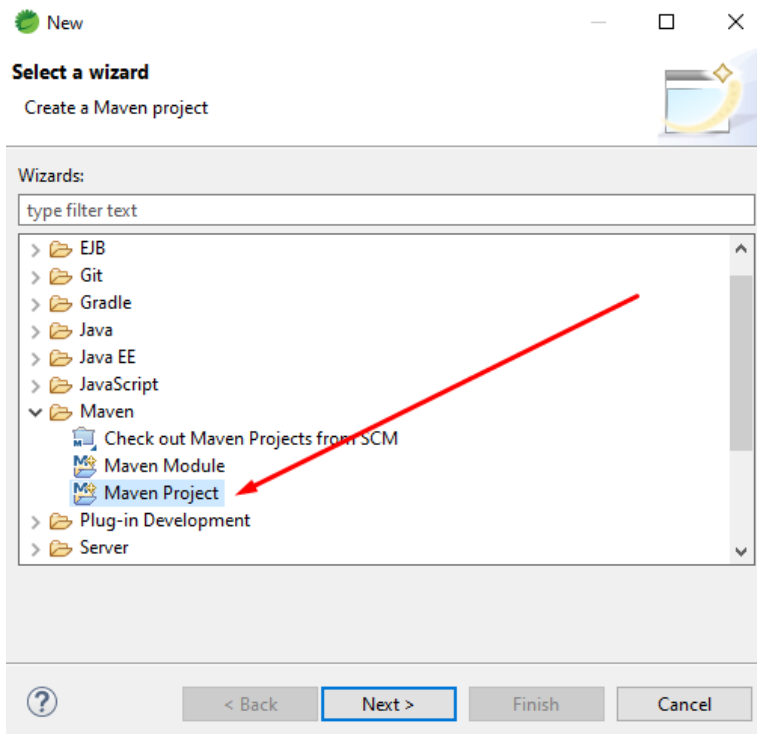
In Spring sono presenti diverse implementazioni dell'interfaccia `ApplicationContext`. Nelle applicazioni Spring è possibile creare un'istanza delle seguenti classi:

- **`ClassPathXmlApplicationContext`**
- **`FileSystemXmlApplicationContext`**
- **`XmlWebApplicationContext`**
- **`AnnotationConfigApplicationContext`**

Le prime tre hanno bisogno di file XML che dovranno contenere i metadati di configurazione dei bean. La classe `AnnotationConfigApplicationContext` invece utilizza codice Java per i metadati.

IoC con XML

Passiamo ora alla pratica con un primo esempio (al momento non utilizziamo Spring Boot). Creiamo un semplice progetto maven





New Maven project

Configure project



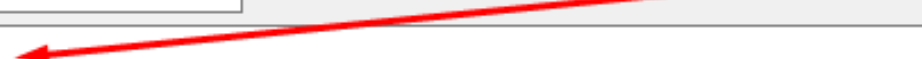
Artifact

Group Id: 

Artifact Id: 

Version:

Packaging:

Name: 

Description:

Parent Project

Group Id:

Artifact Id:

Version:

Advanced



< Back

Next >

Finish

Cancel

La prima cosa da fare è risolvere le dipendenze. Dobbiamo dire al nostro progetto di importare i moduli base di spring che sono : **context**, **beans** e **core**. Per farlo ci aiutiamo con il sito **mvnrepository.com** che è uno dei repository ufficiali di maven.

Andiamo su google, cerchiamo il sito ed andiamo nella home page e sul campo ricerca digitiamo spring :

spring

Search

Repository

- Central 16.1k
- Sonatype 3.8k
- Spring Plugins 2.6k
- Spring Lib M 2.3k
- JCenter 1.0k
- Alfresco 788
- Spring Releases 532
- Spring Milestones 517

Group


- org.springframework 2.9k
- com.github 2.7k
- org.apache 1.4k
- io.github 491
- com.springsource 145
- io.fabric8 110
- org.jboss 102
- org.zalando 102


Category


- Maven Archetype 266
- Web App 220
- Maven Plugins 79
- Java Spec 39
- Android Package 15
- Web Assets 8
- Social Network Client 6
- Message Queue Client 5


Found 19437 results


Sort: **relevance** | popular | newest


1. Spring Context
[org.springframework » spring-context](#)
 Spring Context
 Last Release on May 12, 2021


2. Spring Core
[org.springframework » spring-core](#)
 Spring Core
 Last Release on May 12, 2021


3. Spring Web
[org.springframework » spring-web](#)
 Spring Web
 Last Release on May 12, 2021


4. Spring Beans
[org.springframework » spring-beans](#)
 Spring Beans
 Last Release on May 12, 2021


5. Spring Web MVC
[org.springframework » spring-webmvc](#)
 Spring Web MVC
 Last Release on May 12, 2021

Spring

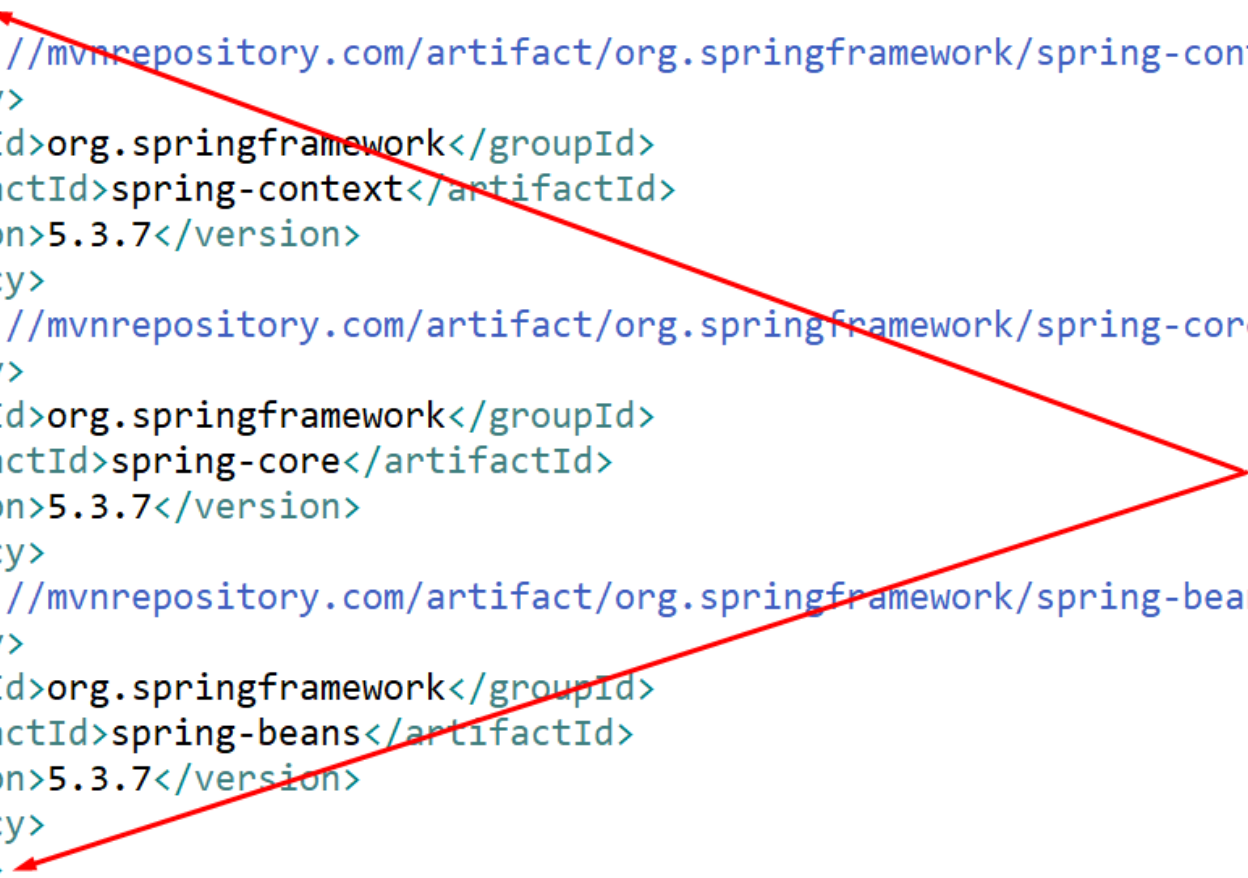
Nello specifico abbiamo detto che a noi ci servono i moduli beans, core e context.

Per importarli dobbiamo cliccare su ognuno di essi, selezioniamo l'ultima versione disponibile e copiamo il codice xml incollandolo all'interno del nostro POM.

```
corsospring/pom.xml  ✖
1<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema
2  <modelVersion>4.0.0</modelVersion>
3  <groupId>com.test.corso</groupId>
4  <artifactId>corsospring</artifactId>
5  <version>0.0.1-SNAPSHOT</version>
6  <name>corsospring</name>
7
8  <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
9<dependency>
10  <groupId>org.springframework</groupId>
11  <artifactId>spring-context</artifactId>
12  <version>5.3.7</version>
13</dependency>
14  <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
15<dependency>
16  <groupId>org.springframework</groupId>
17  <artifactId>spring-core</artifactId>
18  <version>5.3.7</version>
19</dependency>
20  <!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
21<dependency>
22  <groupId>org.springframework</groupId>
23  <artifactId>spring-beans</artifactId>
24  <version>5.3.7</version>
25</dependency>
26
27</project>
```

Tutte le dipendenze vanno messe nel POM xml all'interno del tag **<dependencies>** che andremo a scrivere manualmente :

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.7</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.3.7</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>5.3.7</version>
  </dependency>
</dependencies>
```



Dopo aver salvato il file POM.xml in automatico il nostro IDE scarica le dipendenze. Lo possiamo notare in basso a destra di eclipse con una barra progressiva fino al 100%.

Una volta terminato il download possiamo verificare che all'interno del nostro progetto nella directory «Maven Dependencies» troviamo ora i jar beans, context, core.

Andiamo ora a creare una semplice classe Java contenente il main

```
package corsospring;  
  
public class Inizio {  
    public static void main(String[] args) {  
  
    }  
}
```

Adesso dobbiamo creare il nostro IoC container utilizzando una istanza della classe `ClassPathXmlApplicationContext` in questo modo (il costruttore prende in ingresso il nome del file che conterrà l'elenco della definizione dei nostri bean che creeremo tra poco):

```
package corsospring;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

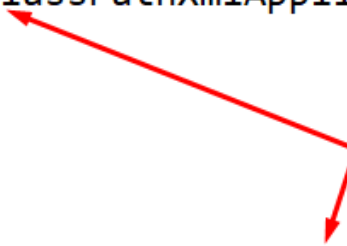
public class Inizio {

    public static void main(String[] args) {

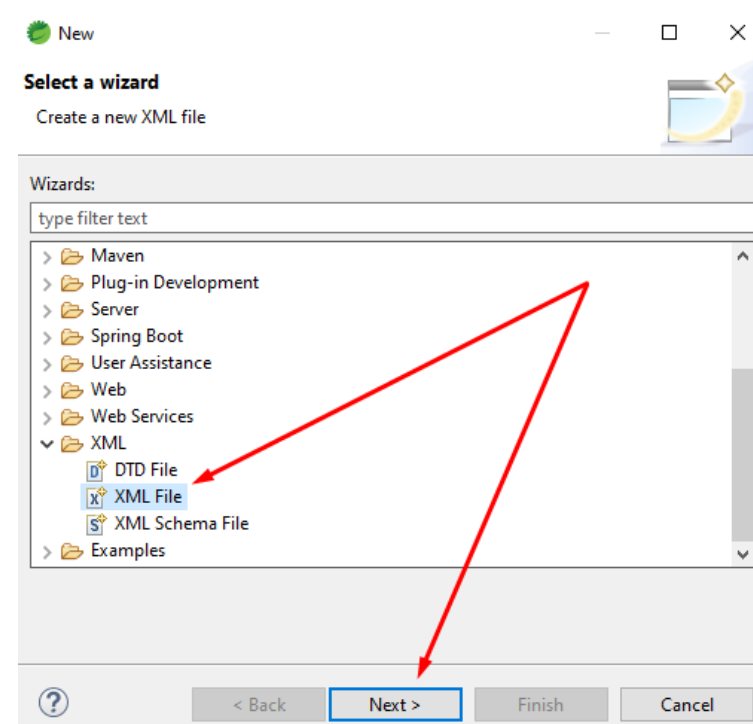
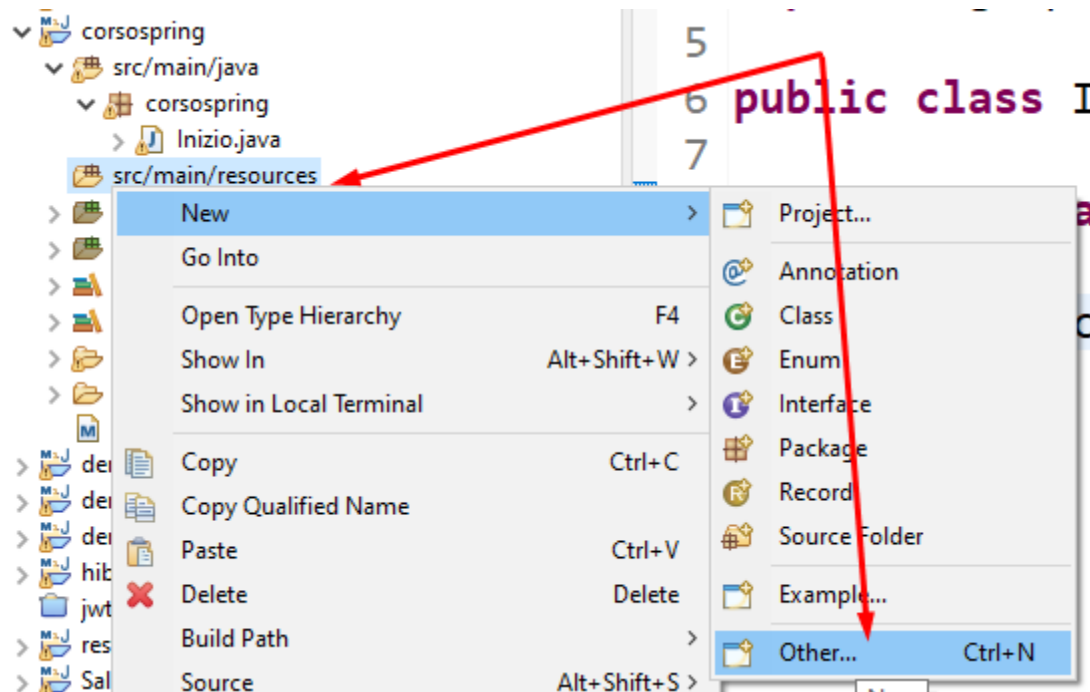
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

    }

}
```



Il file bean.xml lo andremo a creare all'interno della directory src/main/resources in questo modo :



Andiamo ora a creare una classe all'interno del nostro progetto e chiamiamola SalutoService con all'interno un metodo che torna una stringa «Benvenuto!».

```
1 package corsospring;
2
3 public class SalutoService {
4
5     public String saluto() {
6         return "Benvenuto!";
7     }
8 }
9
```

Dobbiamo ora inserire all'interno del file beans.xml l'elenco di tutti i bean (di tutte le classi) che vogliamo far gestire a Spring. Per farlo dobbiamo inserire il tag **<beans>** con all'interno un tag **<bean>** per ogni classe.

Per farlo andiamo sulla documentazione ufficiale del sito di Spring e vediamo come e cosa scrivere :



Why Spring ▾ Learn ▾ Projects ▾ Training Support Co

Spring make product

WHY SPRING

QU

- Overview
- Spring Boot
- Spring Framework
- Spring Cloud
- Spring Cloud Data Flow
- Spring Data
- Spring Integration
- Spring Batch
- Spring Security
- [View all projects](#)

DEVELOPMENT TOOLS
Spring Tools 4

Spring Framework 6.1.0



OVERVIEW

LEARN

SUPPORT

Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** and what you can achieve with them.

6.1.0	CURRENT	GA	Reference Doc.	Api Doc.
6.1.1-SNAPSHOT		SNAPSHOT	Reference Doc.	Api Doc.
6.0.15-SNAPSHOT		SNAPSHOT	Reference Doc.	Api Doc.
6.0.14		GA	Reference Doc.	Api Doc.

Spring Framework Documentation



Overview	History, Design Philosophy, Feedback, Getting Started.
Core	IoC Container, Events, Resources, i18n, Validation, Data Binding, Type Conversion, SpEL, AOP, AOT.
Testing	Mock Objects, TestContext Framework, Spring MVC Test, WebTestClient.
Data Access	Transactions, DAO Support, JDBC, R2DBC, O/R Mapping, XML Marshalling.
Web Servlet	Spring MVC, WebSocket, SockJS, STOMP Messaging.
Web Reactive	Spring WebFlux, WebClient, WebSocket, RSocket.
Integration	REST Clients, JMS, JCA, JMX, Email, Tasks, Scheduling, Caching, Observability, JVM Checkpoint Restore.
Languages	Kotlin, Groovy, Dynamic Languages.
Appendix	Spring properties.
Wiki	What's New, Upgrade Notes, Supported Versions, additional cross-version information.

Configuration Metadata

As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

Note

XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose [Java-based configuration](#) for their Spring applications.

For information about using other forms of metadata with the Spring container, see:

- [Annotation-based configuration](#): define beans using annotation-based configuration metadata.
- [Java-based configuration](#): define beans external to your application classes by using Java rather than XML files. To use these features, see the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata configures these beans as `<bean/>` elements inside a top-level `<beans/>` element. Java configuration typically uses `@Bean`-annotated methods within a `@Configuration` class.

These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, persistence layer objects such as repositories or data access objects (DAOs), presentation objects such as Web controllers, infrastructure objects such as a JPA `EntityManagerFactory`, JMS queues, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of repositories and business logic to create and load domain objects.

The following example shows the basic structure of XML-based configuration metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
```

XML


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5                           https://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="..." class="...">
8
9     </bean>
10
11    <bean id="..." class="...">
12
13    </bean>
14
15 </beans>
```

L'id identifica univocamente un bean in tutto il container, mentre class identifica il tipo di classe che il container dovrà istanziare per noi.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5         https://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="salutami" class="corsospring.SalutoService"></bean>
8
9 </beans>
```

A questo punto nel main posso richiamare il bean attraverso il metodo **getBean** dell'oggetto context :

```
1 package corsospring;
2
3 import org.springframework.context.ApplicationContext;
4 import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6 public class Inizio {
7
8     public static void main(String[] args) {
9
10         ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
11
12         SalutoService s = context.getBean("salutami", SalutoService.class);
13
14         System.out.println(s.saluto());
15     }
16
17 }
18
```

IoC con Annotation

Vediamo ora come creare un IoC container con una classe java invece del file xml, sfruttando quindi la classe `AnnotationConfigApplicationContext`.

Nel main, invece di istanziare una `ClassPathXmlApplicationContext` andiamo ad istanziare una `AnnotationConfigApplicationContext` in questo modo :

```
7 public class Inizio {  
8  
9     public static void main(String[] args) {  
10  
11         //ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
12         ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);  
13  
14         SalutoService s = context.getBean("salutami", SalutoService.class);  
15  
16         System.out.println(s.saluto());  
17     }  
18  
19 }
```

Questo costruttore prende in ingresso non un file xml ma una classe di configurazione che è una normale classe java con una annotation particolare : **@Configuration**

```
1 package corsospring;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class Config {
8
9     @Bean(name = "salutami")
10    public SalutoService getSalutoService() {
11        return new SalutoService();
12    }
13 }
14
```

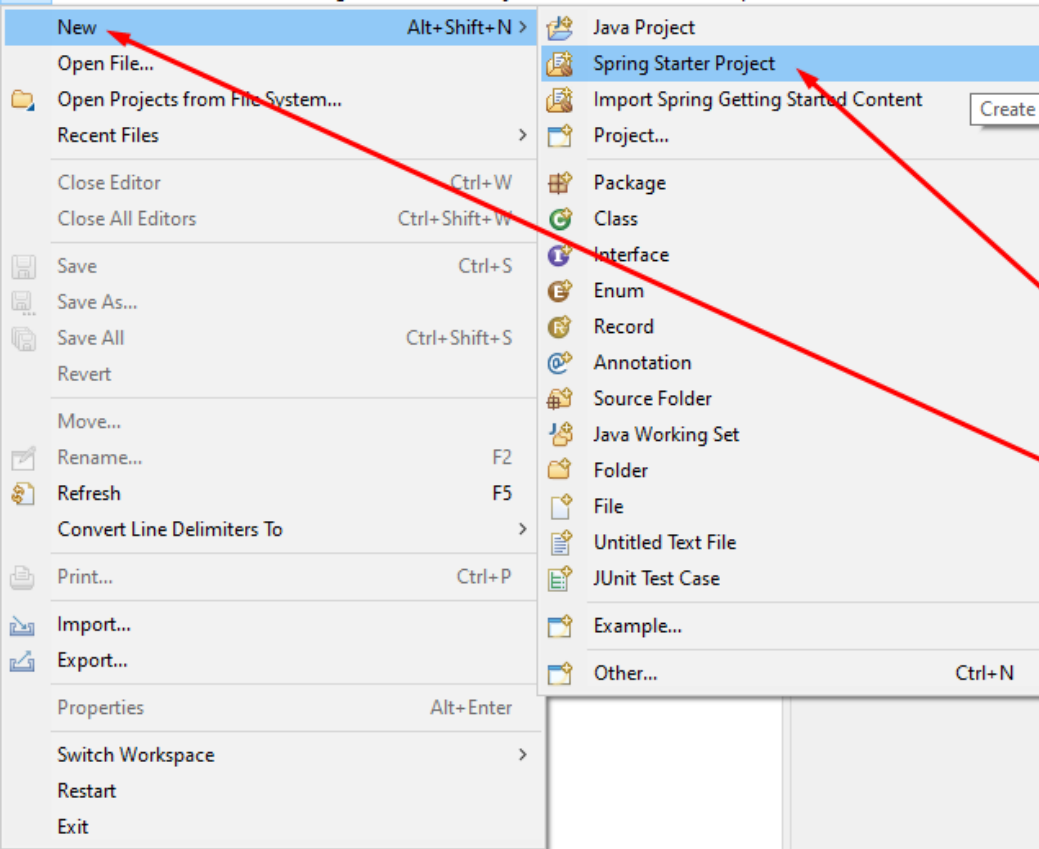
Qui abbiamo fatto la stessa cosa che in precedenza abbiamo fatto nel file beans.xml, il resto rimane tutto invariato :

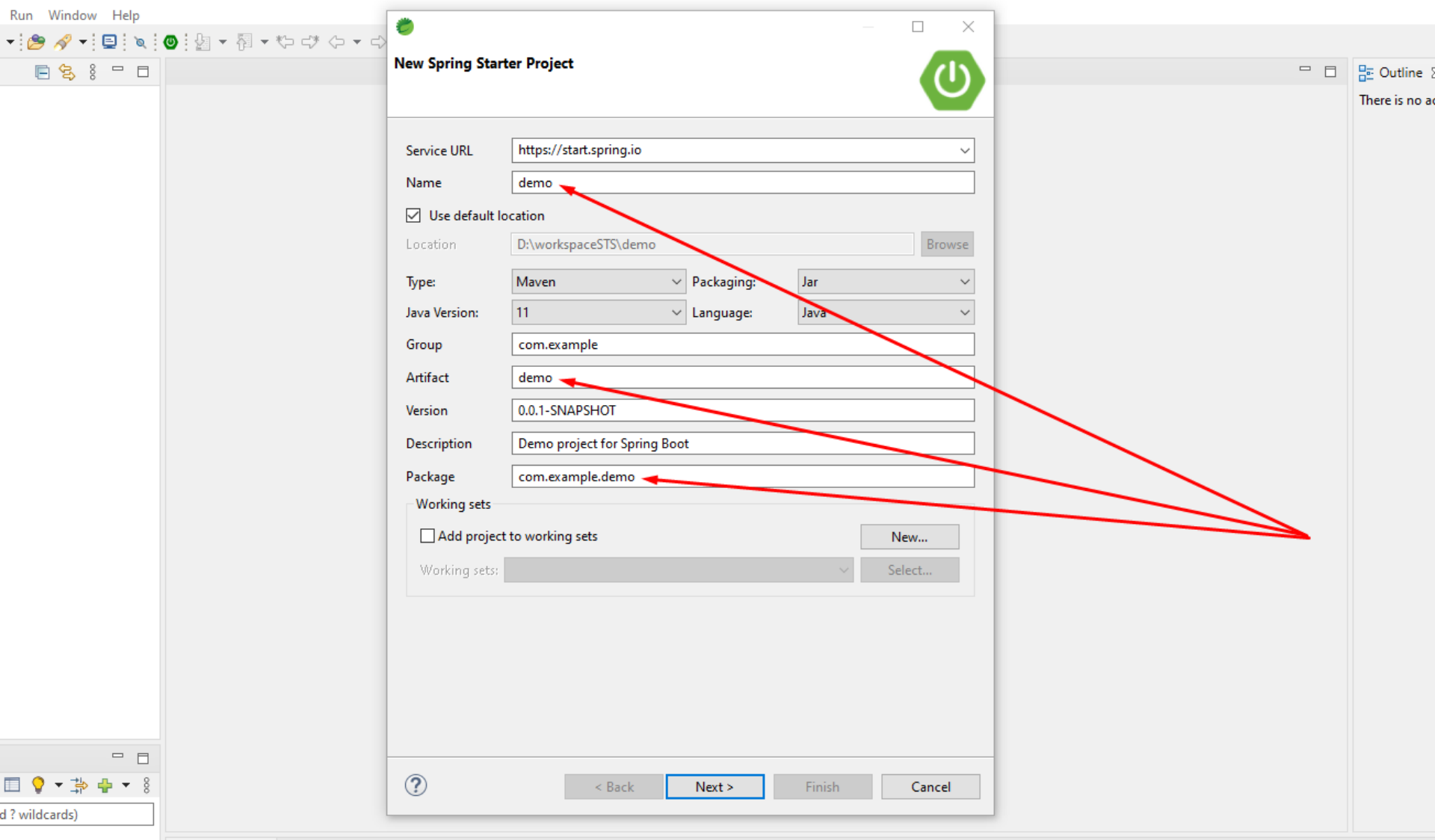
Spring Boot

Spring Boot è un progetto Spring che ha lo scopo di rendere più semplice lo sviluppo e l'esecuzione di applicazioni Spring:

- un'applicazione Spring può richiedere una gran quantità di metadati di configurazione.
- Spring Boot semplifica lo sviluppo delle applicazioni, poiché ne effettua una configurazione automatica (ove possibile), sulla base di valori di default "intelligenti".

- Spring Boot fornisce inoltre delle opzioni per la costruzione (build) e il rilascio (deploy) delle applicazioni in produzione
- Il risultato del build della tua applicazione Spring Boot sarà un standalone JAR file con al suo interno un embedded server. Questo vuol dire che **non avrai bisogno di configurare un servlet container come Tomcat** per eseguire la tua applicazione; basterà eseguire il jar risultato della build e Spring boot tirerà su un embedded server che fungerà da container per la tua web application.






Spring

Scegliere un **nome** per il progetto (per il momento possiamo lasciare anche «demo»), e lasciamo di default anche **Artifact** e **Package** (Artifact lo vediamo più avanti).


A questo punto cliccare su Next e ci appare una finestra dove Spring ci chiede le dipendenze che vogliamo scaricare (in altre parole dobbiamo scegliere i moduli di Spring che vogliamo utilizzare)

Help





— □ ×



New Spring Starter Project Dependencies

Spring Boot Version: 2.4.0 ▾

Available:

Selected:

web ×

▼ Messaging

☐ WebSocket

▼ Template Engines

☐ Thymeleaf

☐ Apache Freemarker

▼ Testing

☐ Testcontainers

▼ Web

☒ Spring Web

☐ Spring Reactive Web

☐ Spring Web Services


☐ Jersey

☐ Vaadin

X Spring Web

Make Default

Clear Selection



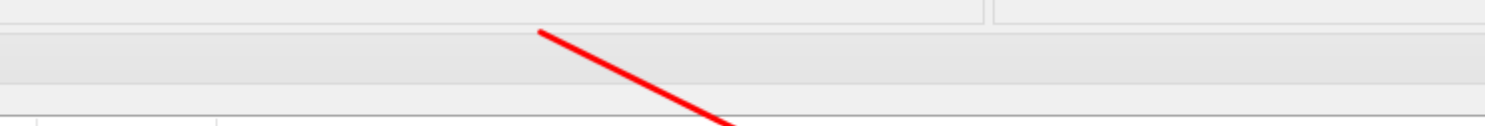
< Back

Next >

Finish

Cancel

Attendiamo ora qualche secondo che Spring provvederà a scaricare i moduli da noi selezionati



The screenshot shows the 'Import Getting Started Content' progress bar in the Canvas LMS interface. The progress bar is at 26% and is highlighted with a red arrow. The interface shows a table with columns 'Location' and 'Type'.

workspaceSTS - demo/src/main/java/com/example/demo/DemoApplication.java - Spring Tool Suite 4

File Edit Source Refactor Navigate Search Project Run Window Help



Package Explorer

- demo [boot]
 - src/main/java
 - com.example.demo
 - DemoApplication.java
 - src/main/resources
 - src/test/java
 - JRE System Library [JavaSE-11]
 - Maven Dependencies
 - src
 - target
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml

DemoApplication.java

```
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class DemoApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(DemoApplication.class, args);
11     }
12
13 }
14
```

Per capire bene il «meccanismo» della Dependency Injection seguiamo passo passo il seguente esempio :

1 – Per prima cosa andiamo a creare nel package principale (nel nostro caso `com.example.demo`) una nuova classe e la chiamiamo «DatabaseProduzione» e al suo interno dichiariamo due variabili di tipo `String` : `url` e `nomedb` ed il suo costruttore.

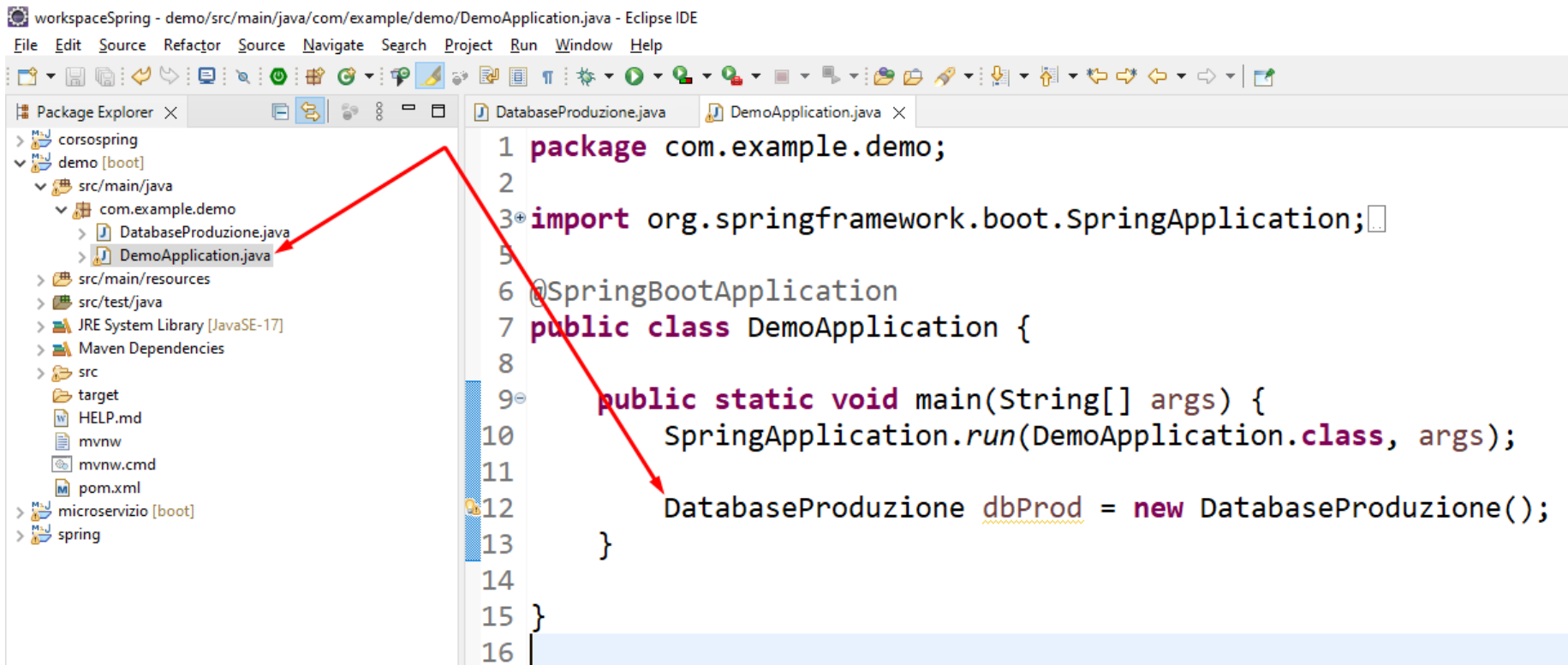
workspaceSpring - demo/src/main/java/com/example/demo/DatabaseProduzione.java - Eclipse IDE

File Edit Source Refactor Source Navigate Search Project Run Window Help

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer displays the project structure: `corsospring` (a project), `demo [boot]` (a module), `src/main/java` (a package), `com.example.demo` (a package), `DatabaseProduzione.java` (a class), and `DemoApplication.java` (a class). A red arrow points from the `DatabaseProduzione.java` file in the Package Explorer to the corresponding file in the editor. The editor shows the following Java code:

```
1 package com.example.demo;
2
3 public class DatabaseProduzione {
4
5     String url;
6     String nomeDb;
7
8     public DatabaseProduzione() {
9
10    }
11 }
12
```


2 – Andiamo ora nella nostra classe principale (dove si trova il main) e proviamo ad inizializzare la classe DatabaseProduzione. La inizializziamo nell'unico modo che conosciamo :



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer displays the project structure: 'workspaceSpring' - 'demo/src/main/java/com/example/demo'. The 'DemoApplication.java' file is selected. On the right, the code editor shows the following Java code:

```
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class DemoApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(DemoApplication.class, args);
11
12         DatabaseProduzione dbProd = new DatabaseProduzione();
13     }
14
15 }
16
```

Two red arrows originate from the Package Explorer. One arrow points to the 'DemoApplication.java' file, and the other points to the line of code in the editor: `DatabaseProduzione dbProd = new DatabaseProduzione();`.

Come è possibile «dire» a Spring che una determinata classe creata dal programmatore venga «inserita» nel container di Spring? Attraverso le seguenti Annotation :

@Component

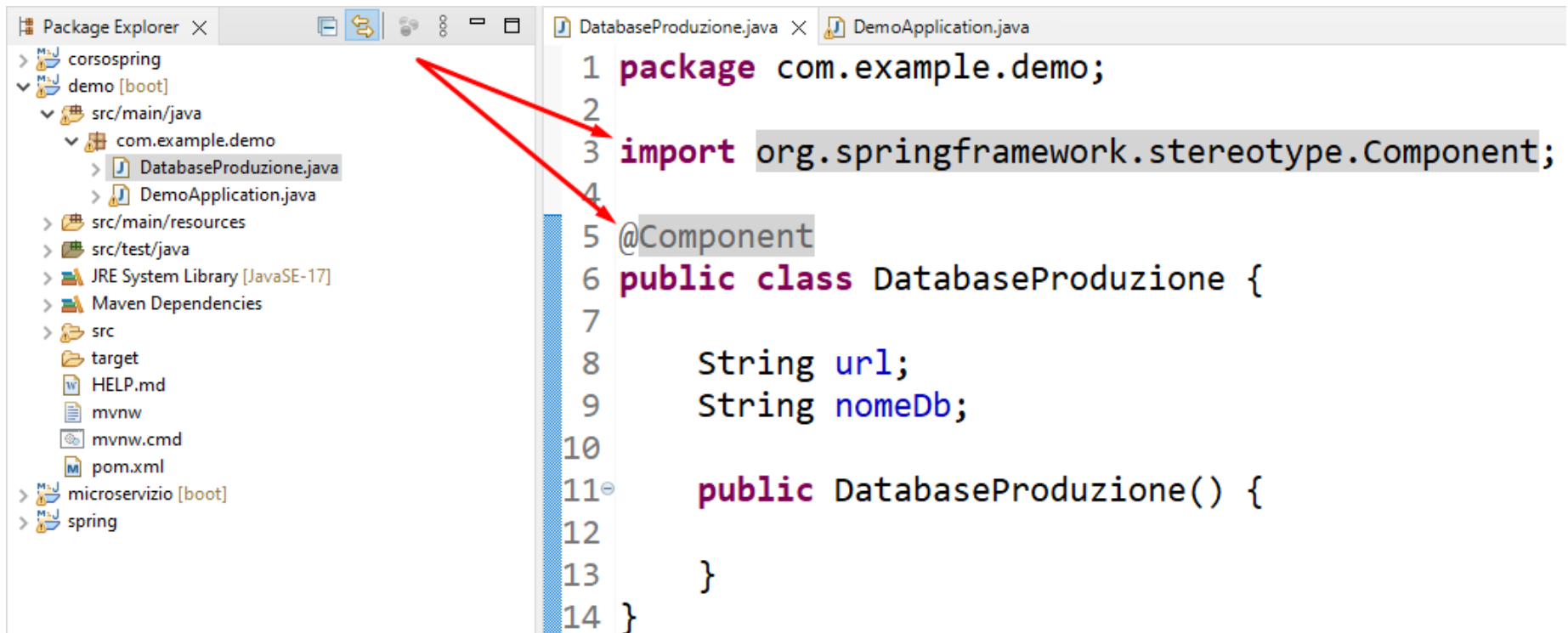
@Controller

@Service

@Repository

Ogni volta che mettiamo un' Annotation su una classe, Spring trasforma la classe in un bean e lo mette all'interno del container. Quando poi ci serve l'istanza di quella classe utilizziamo l'Annotation **@Autowired** per andare a recuperare il bean messo precedentemente nel container.

Torniamo ora al nostro esempio e vediamo come possiamo applicare i concetti appena esposti : prendiamo la classe DatabaseProduzione ed inseriamo l'annotation @Component (che naturalmente va importata) :

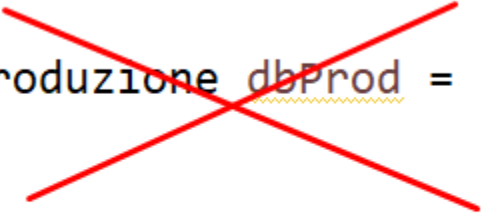


```
1 package com.example.demo;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class DatabaseProduzione {
7
8     String url;
9     String nomeDb;
10
11     public DatabaseProduzione() {
12
13     }
14 }
```

Con questa Annotation, Spring, quando inizializzeremo il progetto, andrà a vedere tutte le classi con l'annotation `@Component` (ed anche tutte la altre annotazioni), creerà un bean e lo metterà nel suo container.

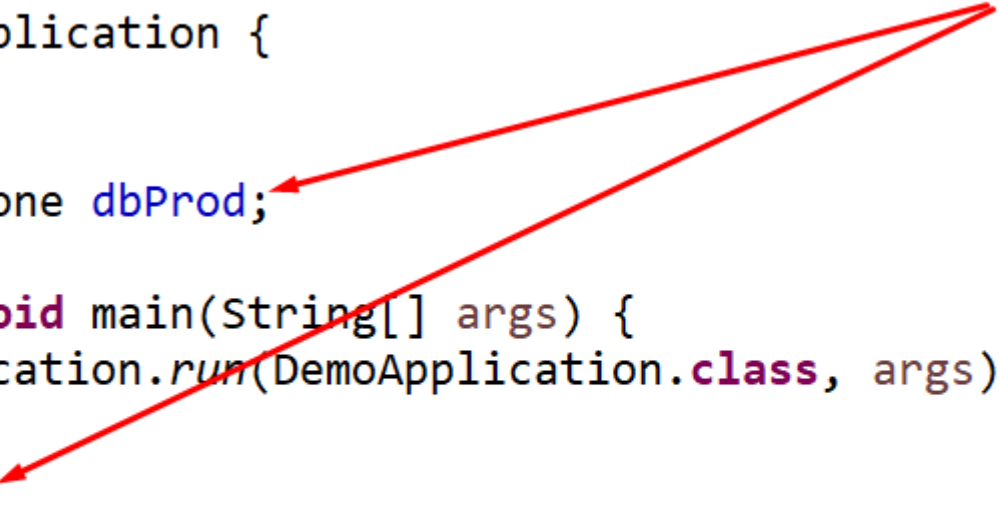
Dove ci servirà l'istanza di `DatabaseProduzione`, non faremo più la new di `DatabaseProduzione`, ma attraverso l'annotation `@Autowired` andremo semplicemente a dichiarare una variabile di quel tipo :

```
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class DemoApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(DemoApplication.class, args);
11
12         DatabaseProduzione dbProd = new DatabaseProduzione();
13     }
14
15 }
16
```



ATTENZIONE : @autowired può essere utilizzato SOLO sulle variabili di istanza (non all'interno dei metodi) quindi siamo obbligati a dichiarare la variabile al di fuori del main :

```
7 @SpringBootApplication
8 public class DemoApplication {
9
10     @Autowired
11     DatabaseProduzione dbProd;
12
13     public static void main(String[] args) {
14         SpringApplication.run(DemoApplication.class, args);
15
16         dbProd.url;
17     }
18
19 }
20
```

Two red arrows originate from the right side of the code block. The first arrow points from the 'DemoApplication' class declaration on line 8 to the 'dbProd' field declaration on line 11. The second arrow points from the 'dbProd' field declaration on line 11 to the 'dbProd.url;' field access on line 16, which is underlined in the original image.

Questo avviene perché il main è un metodo statico, e di conseguenza non può usare o far riferimento a variabili e metodi di istanza in quanto hanno un ciclo di vita differente.

Non potendo utilizzare le variabili annotate con `@autowired` all'interno del main dobbiamo trovare un'altra soluzione, e cioè scrivere un metodo «non statico» e richiamarlo direttamente senza passare per il main... ma come posso farlo ?

@GetMapping

lo scopo principale dell'annotazione @GetMapping è quello di definire un'associazione tra un metodo ed una url; l'elemento in ingresso dell'annotazione serve proprio a specificare quale url saranno associati al metodo.

Per esempio, nella nostra Classe principale posso dichiarare un metodo con l'annotazione @GetMapping che fa da ponte tra una chiamata http ed il metodo (senza dover passare per il main).

@Controller

Oltre ad annotare il metodo con @GetMapping, dobbiamo anche «abilitare» la classe ad intercettare chiamate http tramite l'annotazione @Controller che approfondiremo più avanti.

Per il momento limitiamo ad aggiungere le annotazioni appena descritte sulla classe e sul metodo in questo modo :

```
-
 9 @SpringBootApplication
10 @Controller
11 public class DemoApplication {
12
13     @Autowired
14     DatabaseProduzione dbProd;
15
16     public static void main(String[] args) {
17         SpringApplication.run(DemoApplication.class, args);
18     }
19
20     @GetMapping("/prova")
21     public void test() {
22         System.out.println(dbProd.nomeDb);
23     }
24
25 }
26
```

Avviare ora l'Applicazione : tasto destro sul progetto → run as → Spring boot App . Per richiamare quel metodo appena scritto (test()) è sufficiente aprire il browser e richiamare il nostro application server alla root principale :

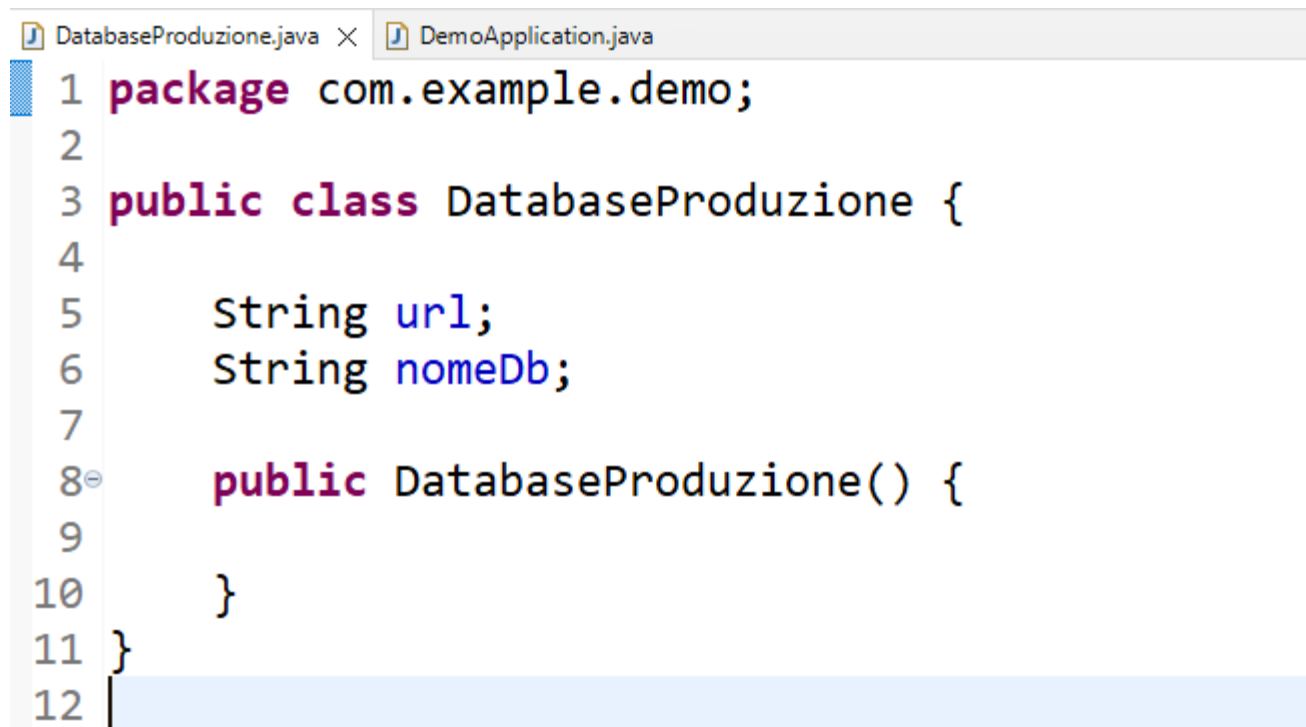


Classe di Configurazione

Finora abbiamo visto come si usa Spring a livello di classe, cioè mettendo l'annotazione sulla classe. C'è un altro modo per «configurare» Spring mettendo l'annotazione non sulla classe, ma sui metodi.

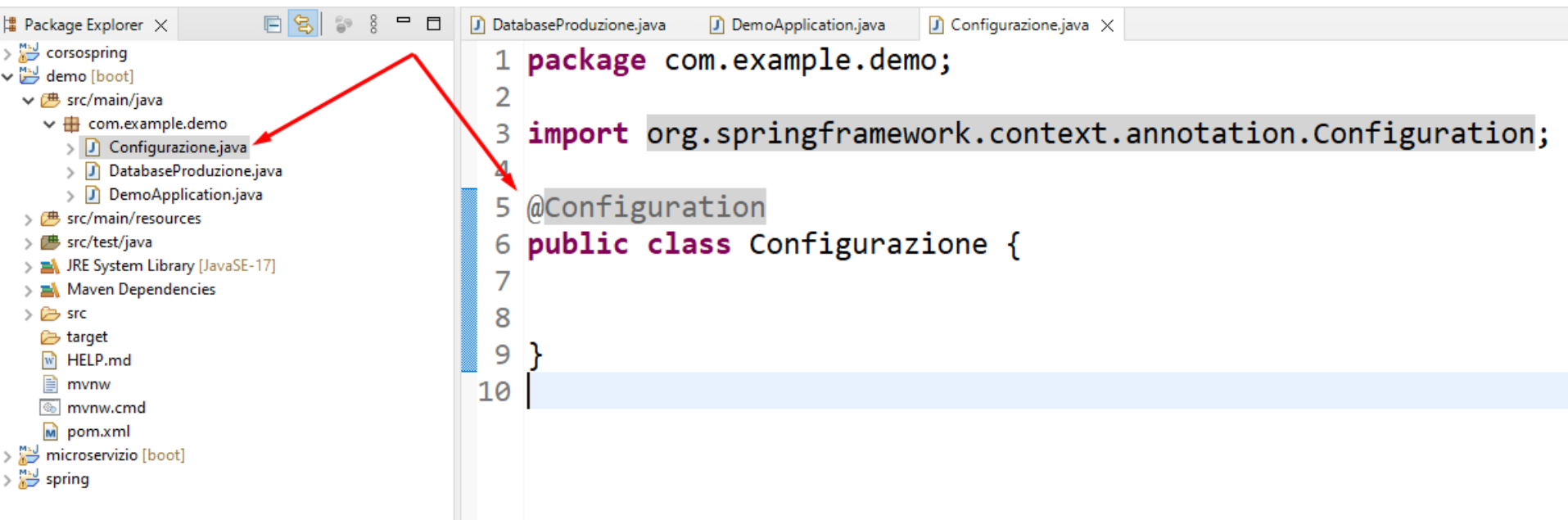
In questo caso si avrà una classe chiamata **classe di Configurazione** dove ci saranno elencati tutti i metodi annotati come @Bean ed ognuno di questo metodo istanzierà un'oggetto, vediamo subito con un esempio il «meccanismo» appena descritto :

1 – Come prima cosa andiamo nella classe DatabaseProduzione e leviamo l'annotazione @Component in modo da eliminare ogni legame assegnato in precedenza



```
DatabaseProduzione.java x DemoApplication.java
1 package com.example.demo;
2
3 public class DatabaseProduzione {
4
5     String url;
6     String nomeDb;
7
8     public DatabaseProduzione() {
9
10    }
11 }
12
```

2 – Creiamo adesso una nuova classe di configurazione. Possiamo chiamarla come vogliamo, quello che conta è l'annotazione che ci mettiamo e cioè **@Configuration**. Per convenzione la chiamiamo Configurazione.



3 – Ora dobbiamo scrivere un metodo che istanzi una variabile di tipo DatabaseProduzione e la restituisce, aggiungendo a questo metodo l'annotazione **@Bean** in questo modo :

```
1 package com.example.demo;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class Configurazione {
8
9     @Bean
10     public DatabaseProduzione configProduzione() {
11
12         DatabaseProduzione dbProd = new DatabaseProduzione();
13         dbProd.nomeDb = "DbDemo";
14         dbProd.url = "https://serverdb:3306";
15
16         return dbProd;
17     }
18 }
19
```

Avviare ora l'Applicazione : tasto destro sul progetto → run as → Spring boot App . Per richiamare quel metodo appena scritto (test()) è sufficiente aprire il browser e richiamare il nostro application server alla root principale :



4 – Il programma va in errore perché non c'è una view associata da visualizzare a video, ma se andiamo sulla console del nostro editor possiamo notare che prima di andare in errore ha stampato a video il nome «DbDemo»

```
2023-12-13T12:34:28.449+01:00 INFO 12724 --- [main] o.apache.catalina.core.StandardEngine
2023-12-13T12:34:28.509+01:00 INFO 12724 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]
2023-12-13T12:34:28.509+01:00 INFO 12724 --- [main] w.s.c.ServletWebServerApplicationContext
2023-12-13T12:34:28.860+01:00 INFO 12724 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer
2023-12-13T12:34:28.868+01:00 INFO 12724 --- [main] com.example.demo.DemoApplication
2023-12-13T12:34:41.095+01:00 INFO 12724 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2023-12-13T12:34:41.096+01:00 INFO 12724 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
2023-12-13T12:34:41.096+01:00 INFO 12724 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
DbDemo
2023-12-13T12:34:41.136+01:00 ERROR 12724 --- [nio-8080-exec-1] o.a.c.c.C.[.][.][dispatcherServlet]
```

```
jakarta.servlet.ServletException: Circular view path [prova]: would dispatch back to the current handler
at org.springframework.web.servlet.view.InternalResourceView.prepareForRendering(InternalResourceView.java:150)
at org.springframework.web.servlet.view.InternalResourceView.renderMergedOutputModel(InternalResourceView.java:125)
at org.springframework.web.servlet.view.AbstractView.render(AbstractView.java:314) ~[spring-webmvc-5.3.13.jar:5.3.13]
at org.springframework.web.servlet.DispatcherServlet.render(DispatcherServlet.java:1431) ~[spring-webmvc-5.3.13.jar:5.3.13]
```

Qualifier Annotation

Riprendendo l'esempio precedente, se volessimo creare un'altra istanza di DatabaseProduzione, dovremmo andare nella classe Configuration e creare un altro metodo che istanzi un altro oggetto... proviamo :

```
6 @Configuration
7 public class Configurazione {
8
9     @Bean
10    public DatabaseProduzione configProduzione() {
11
12        DatabaseProduzione dbProd = new DatabaseProduzione();
13        dbProd.nomeDb = "DbDemo";
14        dbProd.url = "https://serverdb:3306";
15
16        return dbProd;
17    }
18
19    @Bean
20    public DatabaseProduzione configCollaudo() {
21
22        DatabaseProduzione dbColl = new DatabaseProduzione();
23        dbColl.nomeDb = "DbDemoCollaudo";
24        dbColl.url = "https://servercollaudo:3306";
25
26        return dbColl;
27    }
28 }
```

Ma cosa succede adesso se proviamo ad avviare la nostra applicazione ? Ci segnalerà un errore. Perché ? Perché ha trovato due Bean e Spring non sa a chi riferirsi!

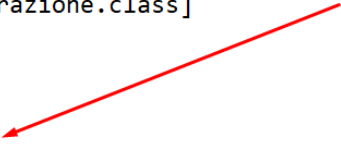
```
*****  
APPLICATION FAILED TO START  
*****
```

Description:

```
Field dbProd in com.example.demo.DemoApplication required a single bean, but 2 were found:  
- configProduzione: defined by method 'configProduzione' in class path resource [com/example/demo/Configurazione.class]  
- configCollaudo: defined by method 'configCollaudo' in class path resource [com/example/demo/Configurazione.class]
```

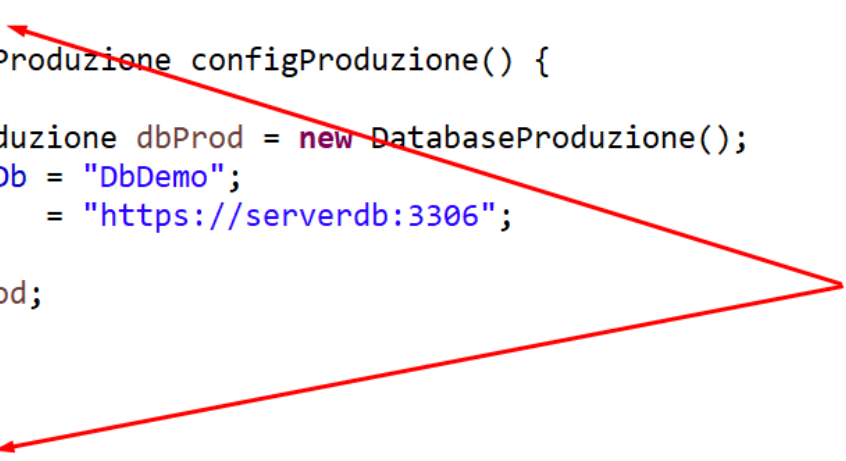
Action:

Consider marking one of the beans as `@Primary`, updating the consumer to accept multiple beans, or using `@Qualifier` to identify the bean




A questo punto ci viene in aiuto l'annotazione `@Qualifier` che ci permette di assegnare un nome ad ogni Bean. Ad esempio :

```
7 @Configuration
8 public class Configurazione {
9
10     @Bean
11     @Qualifier("1")
12     public DatabaseProduzione configProduzione() {
13
14         DatabaseProduzione dbProd = new DatabaseProduzione();
15         dbProd.nomeDb = "DbDemo";
16         dbProd.url = "https://serverdb:3306";
17
18         return dbProd;
19     }
20
21     @Bean
22     @Qualifier("2")
23     public DatabaseProduzione configCollaudo() {
24
25         DatabaseProduzione dbColl = new DatabaseProduzione();
26         dbColl.nomeDb = "DbDemoCollaudo";
27         dbColl.url = "https://servercollaudo:3306";
28
29         return dbColl;
30     }
31 }
```



A questo punto all'interno della nostra applicazione, dove viene effettuato l'Autowired inseriremo anche qui il nostro Qualifier e indicheremo quale Bean vogliamo associare :

```
10 @SpringBootApplication
11 @Controller
12 public class DemoApplication {
13
14     @Autowired
15     @Qualifier("1") DatabaseProduzione dbProd;
16
17
18     public static void main(String[] args) {
19         SpringApplication.run(DemoApplication.class, args);
20     }
21
22     @GetMapping("/prova")
23     public void test() {
24         System.out.println(dbProd.nomeDb);
25     }
26
27 }
```

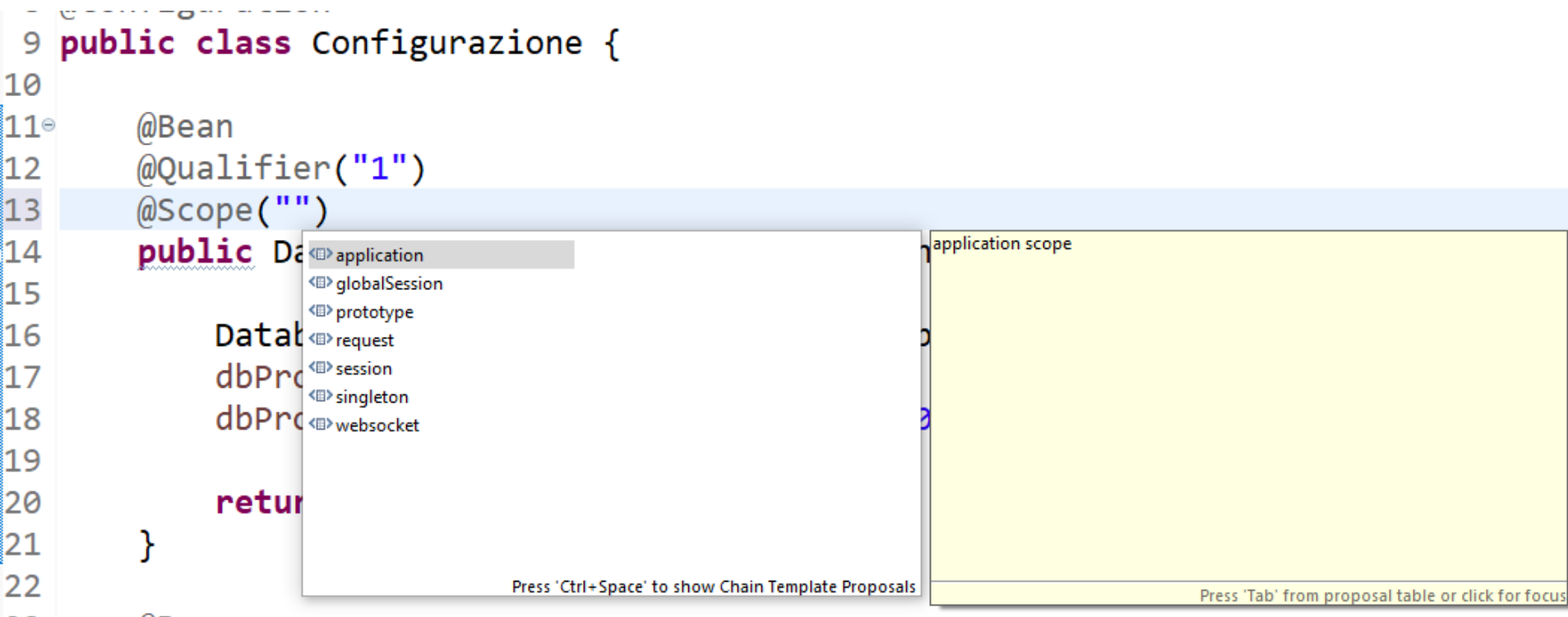


@Scope Annotation

IMPORTANTE : Quando inizializziamo un'applicazione, l'Autowired va a «cercare» l'istanza della Classe, che di default è **Singleton**, ciò vuol dire che chiama sempre la stessa istanza.

Ma possiamo decidere di cambiare questo comportamento attraverso l'annotazione **@Scope** sulla classe di appartenenza (nel nostro caso sulla Classe DatabaseProduzione):

Dopo aver scritto l'annotazione, all'interno delle parentesi se premete CTRL + SPAZIO eclipse ci aiuta e ci suggerisce quali tipi di scope abbiamo :



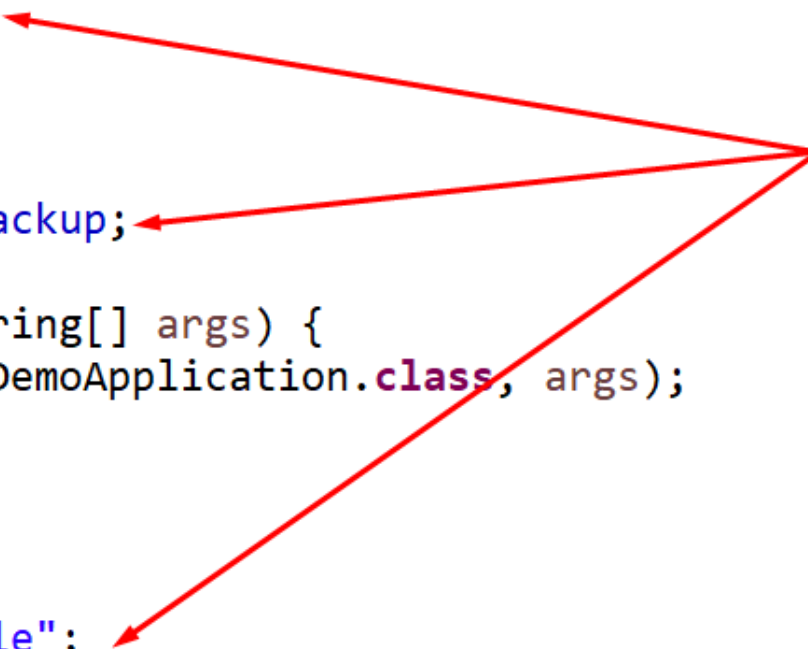
Di default è **Singleton**, ma se mettiamo ad esempio **prototype** stiamo dicendo a Spring che ogni volta che facciamo l'Autowired, deve creare una nuova istanza di quell'oggetto.

Per fare una prova inseriamo nella nostra classe dove richiamiamo il bean un'altra variabile dello stesso tipo e proviamo a stampare il relativo nomeDb :


```
8 @Configuration
9 public class Configurazione {
10
11     @Bean
12     @Qualifier("1")
13     @Scope("singleton")
14     public DatabaseProduzione configProduzione() {
15
16         DatabaseProduzione dbProd = new DatabaseProduzione();
17         dbProd.nomeDb = "DbDemo";
18         dbProd.url     = "https://serverdb:3306";
19
20         return dbProd;
21     }
```



```
10 @SpringBootApplication
11 @Controller
12 public class DemoApplication {
13
14     @Autowired
15     @Qualifier("1")
16     DatabaseProduzione dbProd;
17
18     @Autowired
19     @Qualifier("1")
20     DatabaseProduzione dbProdBackup;
21
22     public static void main(String[] args) {
23         SpringApplication.run(DemoApplication.class, args);
24     }
25
26     @GetMapping("/prova")
27     public void test() {
28         dbProd.nomeDb = "DbReale";
29         System.out.println(dbProd.nomeDb);
30         System.out.println(dbProdBackup.nomeDb);
31     }
32
33 }
```



Che cosa succede se eseguiamo la nostra applicazione ? In questo caso avendo messo lo Scope come **singleton** ogni volta che Spring «incontra» un Autowired **utilizza la stessa istanza**, quindi in questo caso stamperà DbReale per due volte.

Mentre se metto a scope il valore di **prototype**, quando eseguo l'applicazione la variabile dbProd.nome vale DbReale... mentre la variabile dbProdBackup.nomeDb vale DbDemo... provate...

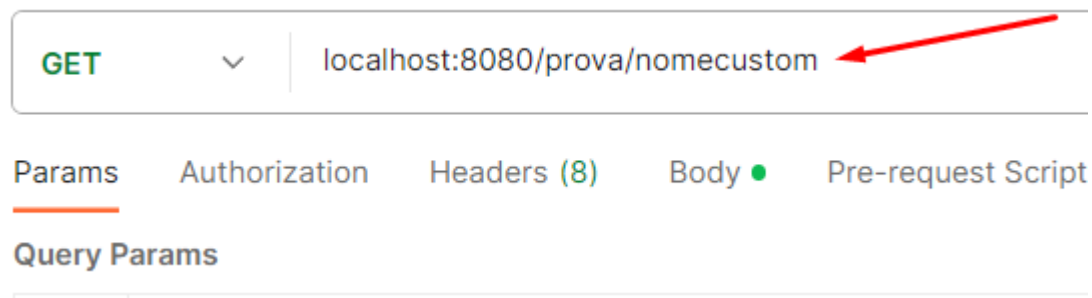
@PathVariable Annotation

Vediamo ora un aspetto particolarmente importante dei metodi di un controller: i **parametri**. Per far gestire a Spring i parametri possiamo usare tre annotazioni :

- @PathVariable
- @RequestParam
- @RequestBody


```
@GetMapping("/prova/{nome}")  
public void test(@PathVariable String nome) {  
    dbProd.nomeDb = nome;  
    System.out.println(dbProd.nomeDb);  
    System.out.println(dbProdBackup.nomeDb);  
}
```


Attraverso l'annotation **@PathVariable** messa davanti al tipo del parametro e l'aggiunta del parametro all'interno delle parentesi graffe nel GetMapping posso passare parametri tramite la url:



@RequestParam annotation

```
@GetMapping("/prova")
public void test(@RequestParam String nome) {
    dbProd.nomeDb = nome;
    System.out.println(dbProd.nomeDb);
    System.out.println(dbProdBackup.nomeDb);
}
```



GET ▼ localhost:8080/prova?nome=nometest 

Params ● Authorization Headers (8) Body ● Pre-request Script Tests Settings

Query Params

	Key
--	-----

La differenza principale è che nel primo caso (con `@PathVariable`) gli passo il valore direttamente nel path, mentre nel secondo caso sfrutto sempre la url ma in modo diverso, faccio riferimento al nome della variabile :

<http://localhost:8080/1> (PathVariable)

<http://localhost:8080?a=1> (RequestParam)