**SBOOT Configurazioni:**

* Scansione di classi Context
* Possibilita di escludere delle classi di context da caricare al boot. (exclude param in @SpringBootAnnotation)

**@SpringBootApplication .**

@SpringBootApplication annotation inherits

* @EnableAutoConfiguration
* @Configuration , and
* @ComponentScan ,   
    
  which is why you can use the exclude parameter within the @SpringBootApplication .

**@EnableAutoConfiguration**

@Enable<Module>

You will find that the Spring Framework and some of its modules—like Spring Core, Spring Data,

Spring AMQP, and Spring Integration—provide @Enable<Technology> annotations.  
For example, @EnableTransactionManagement , @EnableRabbit , and @EnableIntegration are part of the modules

mentioned. Within Spring applications, you can use these annotations to follow the pattern “convention

over configuration,” thus making your apps even easier to develop and maintain without worrying too

much about configuration.

Come Sboot li usa:

Spring Boot can also take advantage of these annotations. These annotations are used in the

@EnableAutoConfiguration annotation to do the auto-configuration

the @EnableAutoConfiguration annotation.  
As you already know, this class will attempt to guess and configure the beans that your application will need. The auto-configuration classes are applied based on the classpath and which beans your app has defined, but what this makes more powerful is the org.springframework.boot.autoconfigure.**EnableAutoConfigurationImportSelector** class that

finds all the necessary configuration classes.

The EnableAutoConfigurationImportSelector class has a several methods, but one of the most

important for the auto-configuration to happen is the getCandidateConfiguration method

**Lettura configurazioni di default:**

1. Tramite @SpringBootApplication Sboot
2. Eredita @EnableAutoConfiguration
3. Che eredita da @EnableAutoConfigurationImportSelector ..
4. Il quale attraverso il metodo getCandidateConfigurations
   1. Carica alcune delle classi predefinite in spring-boot-autoconfigure JAR
   2. META-INF/spring.factories (@Enable<Module>)
   3. IL caricamento avviene tramite abilitazione da .properties
      1. @ConditionalOnProperty(prefix = "spring.cloud", name = "enabled", havingValue = "true",matchIfMissing = true)

SpringApplication class api

* Static:
  + SpringApplication.run(<@SpringBootApplication>class)
* Instanza
  + Sbapp.run()
  + Sbapp.setBanner() ...

SpringApplicationBuilder fluent api

Simile al precendente, ma con fluent api.

* Carica contesti con **.child(**MyConfig.class)
* Abilita profili con **.profiles("prod","cloud")**
* Aggiunge event listeners con **.listeners**

**Eventi a livello application Context:**

In addition, you can have these events:

* ApplicationStartedEvent (sent at the start),
* ApplicationEnvironmentPreparedEvent (sent when the environment is known),
* ApplicationPreparedEvent (sent after the bean definitions),
* ApplicationReadyEvent (sent when the application is ready),
* ApplicationFailedEvent (sent in case of exception during the startup

All these events can be useful when you want to set up your application (database, check up for some services,

etc.) before it runs, or if your application fails during a start ( ApplicationFailedEvent ), because you’ll

probably want to send a notification somewhere.

**Runtime e configurazione:**

Application args:

E’ possibile passare args all applicazione da riga di commando.  
Tali parametri sono

* inoltrati tramite SpringApplication.run(class, args)
* disponibili in ogni componente con injection da costruttore con ApplicationArguments

Executable jar da CLI

ApplicationRunner and CommandLineRunner :  
Possibilita di eseguire codice prima di esecuzione classe SpringBootApplication

**Properties:**

Spring Boot has different options for saving your application configuration:

• You can use a file named application.properties , which should be located in the

root classpath of your application (there are more places where you can add this file

that you’ll learn about later).

• You can use a YAML notation file named application.yml that also needs to be

located in the root classpath (there are more places where you can add this file that

you’ll learn about later).

• You can use environment variables. This is becoming the default practices for cloud

scenarios.

• You can use command-line arguments.

**Priorita**

Spring Boot uses an order if you want to override your application configuration properties:

• Command-line arguments

• SPRING\_APPLICATION\_JSON

•JNDI (java:comp/env)

• System.getProperties()

• OS environment variables

• RandomValuePropertySource (random.\*)

• Profile-specific ( application-{profile}.jar ) outside of the package JAR

• Profile-specific ( application-{profile}.jar ) inside of the package JAR

• Application properties ( application.properties ) outside of the package JAR

• Application properties ( application.properties ) inside of the package JAR

• @PropertySource

• SpringApplication.setDefaultProperties

Paths:

Spring Boot has an order to find the application.properties or YAML file. It will look in:

• The /config subdirectory located in the current directory

• The current directory

• A classpath /config package

• The classpath root

**Change config path:**

$./mvnw spring-boot:run -Dspring.config.name=mycfg

Or

$ SPRING\_CONFIG\_NAME=mycfg java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar

**Profiles:**

* Definire application-{profile}.properties
* Setup Profile:
  + Da codice
    - @ActiveProfiles
    - setActiveProfile
  + Da env variable SPRING\_PROFILES\_ACTIVE
  + Da riga comando
    - $ ./mvnw clean spring-boot:run -Dspring.profiles.active=prod

**Custom properties prefix:**

* Creare bean che mappa le properties:
  + Definire prefisso a livello classe
  + Getter/setter