



# Spring Boot

# Introduction



- Spring Boot is a Framework from “The Spring Team” to ease the bootstrapping and development of new Spring Applications.
- Spring Boot Framework is not implemented from the scratch by The Spring Team
- Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".
- It provides defaults for code and annotation configuration to quick start new Spring projects within no time.
- It follows an approach to avoid lot of boilerplate code , configuration to improve Development, Unit Test and Integration Test Process.

# Uses



- To ease the Java-based applications Development, Unit Test and Integration Test Process.
- To reduce Development, Unit Test and Integration Test time by providing some defaults.
- To increase Productivity.

# Advantages



- It reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
- It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.
- It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
- It provides lots of plugins to work with embedded and in-memory Databases very easily.


# Limitations

- It is somewhat time consuming process to convert existing or legacy Spring Framework projects into Spring Boot Applications but we can convert all kinds of projects into Spring Boot Applications. It is very easy to create brand new/Greenfield Projects using Spring Boot.

# Spring Boot Starters



Starter name	Description
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML
spring-boot-starter-jdbc	Starter for using JDBC with the HikariCP connection pool
spring-boot-starter-data-jdbc	Starter for using Spring Data JDBC
spring-boot-starter-data-jpa	Starter for using Spring Data JPA with Hibernate
spring-boot-starter-security	Starter for using Spring Security
spring-boot-starter-web	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
spring-boot-starter-actuator	Starter for using Spring Boot's Actuator which provides production ready features to help you monitor and manage your application



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

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

# Spring Boot Annotation



## ■ **@SpringBootApplication:**

This annotation is used on the application class while setting up a Spring Boot project and must be kept in the base package. The one thing that *@SpringBootApplication* does is a component scan. But it will scan only its sub-packages.

■ The *@SpringBootApplication* is a convenient annotation that adds all the following:

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan



# Spring Boot Annotation

## ■ @EnableAutoConfiguration

This annotation is usually placed on the main application class. The @EnableAutoConfiguration annotation implicitly defines a base package to search. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

# Spring Boot - Quick Start

- Spring Boot CLI
- Spring Initializr
- Spring Tool Suite

## ■ Application Runner

Application Runner is an interface used to execute the code after the Spring Boot application started.

`@SpringBootApplication`

```
public class DemoApplication implements ApplicationRunner {  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
    @Override  
    public void run(ApplicationArguments arg0) throws Exception {  
        System.out.println("This is an Application Runner");  
    }  
}
```

## ■ Command line Runner

It is used to execute the code after the Spring Boot application started.

```
@SpringBootApplication
```

```
public class DemoApplication implements CommandLineRunner {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(DemoApplication.class, args);
```

```
    }
```

```
@Override
```

```
public void run(String... args) throws Exception {
```

```
    System.out.println("Hello world from Command Line Runner");
```

```
}
```

```
}
```

# Spring Boot -Application Properties

## ■ Command Line Properties

Spring Boot application converts the command line properties into Spring Boot Environment properties. Command line properties take precedence over the other property sources. By default, Spring Boot uses the 8080 port number to start the Tomcat. E.g. changing port number.

## ■ Properties File

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the **application.properties** file under the classpath.

## ■ YAML files

## ■ External properties file

# Spring Boot -Application Properties

## ■ Command Line Properties

```
F:\training\FirstDemo\target>java -jar FirstDemo-0.0.1-SNAPSHOT.jar  
--server.port=9090
```

## ■ Properties File

```
server.port = 9090  
spring.application.name = FirstDemo
```

## ■ YAML File

```
spring:  
  application:  
    name: FirstDemo  
  server:  
    port: 9090
```

## ■ External properties file

# Working with @value

## ■ Through program

```
@Value("${spring.application.owner}")  
private String owner;
```

```
@Value("${spring.application.owner:abhijeet}")
```

## ■ Properties File

```
spring.application.owner=abhijeet
```

# Profiles

## **application.properties**

spring.application.owner = demoservice

## **application-dev.properties**

spring.application.owner = demoservice

## **application-prod.properties**

spring.application.owner = demoservice



# Selecting profiles

## ■ Through properties file (application.properties)

`spring.profiles.active=dev`

## ■ Runtime

Select “Run as Configuration” →

**VM Arguments:** `-Dspring.profiles.active=dev`

# Removing Banner in boot

- `spring.main.banner-mode=off`

# Defining Context

```
server.servlet.context-path=/secondrestdemo
```

# Spring boot JPA

- What is Spring Boot Data JPA
- Why Spring Boot Data JPA
- Architecture or API
- Implementation

# Spring boot JPA

- Repository
- CrudRepository
- PagingAndSortingRepository
- JpaRepository

# CrudRepository

- count
- save(Entity)
- delete(Entity)
- deleteAll()
- deleteById(ID id)
- findAll()
- findAllById(ID id)



**CrudRepository** provides CRUD functions

**PagingAndSortingRepository** provides methods to do pagination and sort records

**JpaRepository** provides JPA related methods such as flushing the persistence context and delete records in a batch

# Steps

1. Create a spring starter application

2. Add mysql and JPA repository

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

3.



# Steps

## 3. Add the database configuration to application.properties

```
spring.jpa.hibernate.ddl-auto=  
spring.datasource.url=  
spring.datasource.username=  
spring.datasource.password=
```

## 4. Create an entity class. This class must have @Entity annotation, and Id (Note the data type of ID)

## 5. Create repository interface, which should extend

CrudRepository OR JPA repository

CrudRepository<Entity\_Class, Data\_type\_of\_ID>

JpaRepository<Entity\_class, Data\_Type\_of\_ID>

# Steps

6. Declare additional methods in this interface and also initiate a query for that if you want to make sure to happen it implicitly. Else provide implementation in Service class.
7. In Application or service layer **autowire** the repository
8. Invoke Methods

# Using Custom Queries

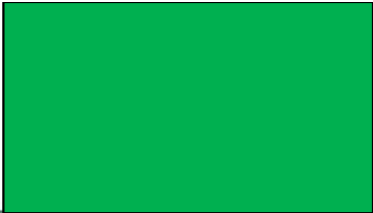
- Using Structured Method Names
- Using Query Annotation
- Using Named Queries


# Creating Structured Methods

- must start with prefixes: *find...By*, *read...By*, *query...By*, *count...By*, and *get...By*.
- Limiting number of results: *findTopBy*, *findTop1By*, *findFirstBy*, and *findFirst1By*
- Unique Results: *findTitleDistinctBy* or *findDistinctTitleBy* means that we want to select all unique titles that are found from the database.
- We must add the search criteria of our query method after the first *By* word
- If our query method specifies x search conditions, we must add x method parameters to it
- Add the return type without fail

- **Example 1:** If we want to create a query method that returns the todo entry whose id is given as a method parameter, we have to add one of the following query methods to our repository interface:

- `public Todo findById(Long id);`

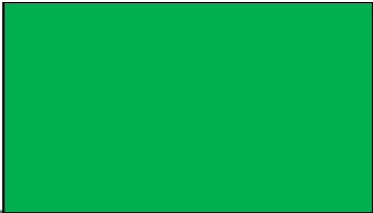
- 
- **Example 2:** If we want to create a query method that returns todo entries whose title or description is given as a method parameter, we have to add the following query method to our repository interface:
    - `public List<Todo> findByTitleOrDescription(String title, String description);`

- 
- **Example 3:** If we want to create a query method that returns the number of todo entries whose title is given as a method parameter, we have to add the following query method to our repository interface:

- `public long countByTitle(String title)`

- **Example 4:** If we want to return the distinct todo entries whose title is given as a method parameter, we have to add the following query method to our repository interface:
  - `public List<Todo> findDistinctByTitle(String title);`



- 
- **Example 5:** If we want to return the first 3 todo entries whose title is given as a method parameter, we have to add one of the following query methods to our repository interface:
  - `public List<Todo> findFirst3ByTitleOrderByTitleAsc(String title);`

# Using Query Annotation

- Limitations of Method approach
- Apply @Query annotation followed by Query to execute

- **JPQL**

```
@Query("SELECT t FROM Todo t WHERE t.title = 'title' ")  
public List<Todo> findById();
```

- Creating SQL query

```
@Query(value = "SELECT * FROM todos t WHERE t.title = 'title'",  
      nativeQuery=true)  
public List<Todo> findByTitle();
```

# Using Named Query

- Annotate the entity with the `@NamedNativeQuery` annotation.
- Set the name of the named query (`Todo.findByTitles`) as the value of the `@NamedNativeQuery` annotation's `name` attribute.
- Set the SQL query (`SELECT * FROM todos t WHERE t.title = 'title'`) as the value of the `@NamedNativeQuery` annotation's `query` attribute.
- Set the returned entity class (`Todo.class`) as the value of the of the `@NamedNativeQuery` annotation's `resultClass` attribute.

# Using Named Query

```
@Entity
@NamedNativeQuery(name = "Todo.findByTitles",
    query="SELECT * FROM todos t WHERE t.title = 'title'",
    resultClass = Todo.class
)
@Table(name = "todos")
final class Todo {

}
```

# Using Named Query

## ■ Type 1

```
interface TodoRepository extends Repository<Todo, Long> {  
    public List<Todo> findByTitles();  
}
```

## ■ Type 2

```
@Query(name="")  
interface TodoRepository extends Repository<Todo, Long> {  
    public List<Todo> findByTitles();  
}
```