1.

Creation of a Spring Boot Application

To build a Spring Boot application from scratch, we typically use **Spring Initializr**, which provides a quick and efficient way to generate the project structure with the required dependencies.

Step-by-Step Process:

1. **Go to Spring Initializr**:
   Visit https://start.spring.io, which is an online project generator tool for Spring Boot applications.
2. **Configure Project Metadata**:
   - **Project**: Maven or Gradle (commonly Maven)
   - **Language**: Java
   - **Spring Boot Version**: Choose the latest stable release
   - **Group**: e.g., com.example
   - **Artifact**: e.g., demo
   - **Name & Package**: Set your application name and base package
   - **Dependencies**: Add the required modules like Spring Web, Spring Data JPA, H2 Database, etc.
3. **Download and Import Project**:
   After clicking **Generate**, a .zip file will be downloaded. Extract and open the folder in your IDE (like IntelliJ or Eclipse).
4. **Main Class**:
   The main class is annotated with @SpringBootApplication and acts as the entry point of the application.

```
package com.example.demo;


import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication

public class DemoApplication {

  public static void main(String[] args) {

    SpringApplication.run(DemoApplication.class, args);

  }
```

}

- @SpringBootApplication is a combination of three annotations:
    - @Configuration – Marks this class as a source of bean definitions.
    - @EnableAutoConfiguration – Automatically configures your Spring application based on dependencies.
    - @ComponentScan – Scans the package for Spring components.

5. **Running the Application**:
   Use the IDE or the command mvn spring-boot:run to start the server. By default, it runs on port **8080**.

2.

Need and Benefits of Spring Boot

Why Spring Boot?

Spring Boot was introduced to simplify the development of Java-based enterprise applications. Traditional Spring development often required a lot of XML configurations, dependency management, and setup before writing actual logic. Spring Boot solves these issues by providing a production-ready setup with minimal effort.

Key Benefits of Spring Boot:

1. **Simplified Development**
   Spring Boot reduces complexity by auto-configuring commonly used settings, letting developers focus on writing business logic instead of infrastructure setup.
2. **No Boilerplate Code**
   You don't have to write repetitive configuration or XML files — Spring Boot handles that for you with annotations and conventions.
3. **Faster Development Time**
   Built-in tools and starter dependencies reduce the time taken to set up, develop, and deploy applications.
4. **Embedded Web Server**
   Comes with an embedded **Tomcat** (or Jetty/Undertow), so you can run your application as a standalone JAR — no need to deploy on external servers.
5. **Production Ready Features**
   Spring Boot provides features like health checks, metrics, and monitoring out of the box using **Spring Boot Actuator**.

6. **Easy Dependency Management**
   With pre-defined starter packages (like spring-boot-starter-web), it handles all internal dependencies required for building applications.
7. **Minimal XML Configuration**
   Most of the configuration is handled through annotations like @SpringBootApplication, reducing the need for verbose XML files.

3.

Loading Bean from Spring Configuration File

In traditional Spring (non-Spring Boot) applications, beans are defined in an **XML configuration file**. This file helps the **IoC (Inversion of Control) container** manage object creation, configuration, and lifecycle.

1. Spring Configuration XML

The XML file uses the spring-beans.xsd schema to define beans:

```
<?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">

  <bean id="studentBean" class="com.example.Student">
    <constructor-arg name="name" value="Rekha"/>
    <constructor-arg name="age" value="21"/>
  </bean>

</beans>
```

2. Bean Configuration Elements

- <bean>: Declares a Spring-managed bean.
- id: Unique identifier for the bean.

- class: Fully qualified class name.
- <constructor-arg>: Used for **constructor injection**.
- <property>: Used for **setter injection**.

3. Loading Bean in Java

Use **ApplicationContext** or **ClassPathXmlApplicationContext** to load beans from the configuration file:

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.example.Student;


public class MainApp {

   public static void main(String[] args) {

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

      Student student = (Student) context.getBean("studentBean");

      System.out.println(student.getName() + " - " + student.getAge());

   }

}
```

4. Bean Scopes

- singleton (default): A single instance is shared throughout the application.
- prototype: A new instance is created every time getBean() is called.

Example:

```
<bean id="studentBean" class="com.example.Student" scope="prototype"/>
```

5. Dependency Injection Types

- **Constructor Injection**: Uses <constructor-arg> to pass values at the time of bean creation.
- **Setter Injection**: Uses <property name="..." value="..."/> to set values using setter methods.

5. Inclusion of Logging in Spring Boot Application

Spring Boot comes with built-in support for logging using SLF4J (Simple Logging Facade for Java) and Logback as the default logging implementation. You can customize the logging behavior easily through the application.properties file or programmatically using LoggerFactory.

1. Basic Logger Setup in Code

Use LoggerFactory to create a logger and print messages at various levels.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class LoggingDemoApplication implements CommandLineRunner {

    private static final Logger logger = LoggerFactory.getLogger(LoggingDemoApplication.class);

    public static void main(String[] args) {
        SpringApplication.run(LoggingDemoApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        logger.trace("This is a TRACE message");
        logger.debug("This is a DEBUG message");
        logger.info("This is an INFO message");
        logger.warn("This is a WARN message");
        logger.error("This is an ERROR message");
    }
}
```

2. Configure Logging in application.properties

```
# Set the root logging level
logging.level.root=info

# Set logging level for specific package
logging.level.com.example=debug

# Customize the log message format
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n

# Change server port (optional)
server.port=8081
```

3. Log Levels in Spring Boot

Spring Boot supports the following logging levels, from lowest to highest:

- TRACE – Very detailed logs (for troubleshooting)
- DEBUG – Debugging information
- INFO – General application events
- WARN – Potential issues
- ERROR – Serious problems

Only logs at or above the configured level are printed.

4. Default Behavior

- Spring Boot uses **Logback** by default.
- Logs are printed to the console.
- You don't need to add any external logging dependencies unless you want to switch to **Log4j2** or others.