Java Technologies

Lecture 2

Spring Boot

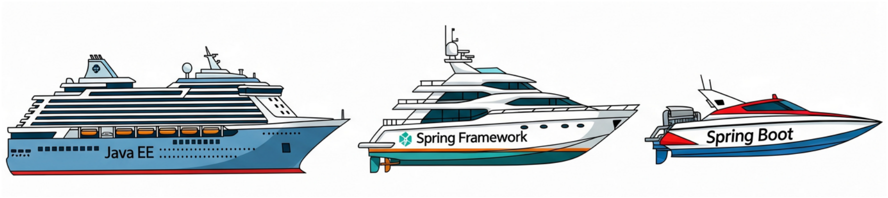
Fall, 2025

Agenda

- Java EE, Spring Framework, Spring Boot
- Bootstrap a Simple Spring Boot Application
- Spring Boot Starters
- Java Annotations Recap
- Application Initialization
- Component Scanning
- Auto-Configuration, Custom/Conditional Configurations
- Externalized Configurations
- Spring Expression Language (SpEL)
- Spring Boot Profiles, Spring Cloud Config
- Command Line Arguments, Spring Boot Actuator
- Properties vs. YAML, Dependency Management

The Context

- **Java EE**: originally called J2EE was released in 1999 by Sun Microsystems to be a standard for enterprise development.
  $\rightarrow$ Heavy and complex, especially EJBs. (Not any more)
- **Spring Framework**: created by Rod Johnson, published in his 2002 book *Expert One-on-One J2EE Design and Development*.
  $\rightarrow$ Simplify Java EE development.
- **The Rise of Spring Boot (2014)**: "Make it easy to create stand-alone, production-grade Spring-based applications with minimal effort."



Spring Framework

Spring Framework is a comprehensive, modular framework for building enterprise-level applications. Its core features are:

- Dependency Injection (DI) and Inversion of Control (IoC) for managing components and their lifecycles.
- Aspect-Oriented Programming (AOP) to handle cross-cutting concerns like logging and transactions.
- MVC framework for building web applications.
- Transaction management and data access.
- Security, messaging, batch processing, and many more modules.
- Requires manual configuration and deployment to a Java EE server, such as Tomcat or Jetty.

Spring Boot

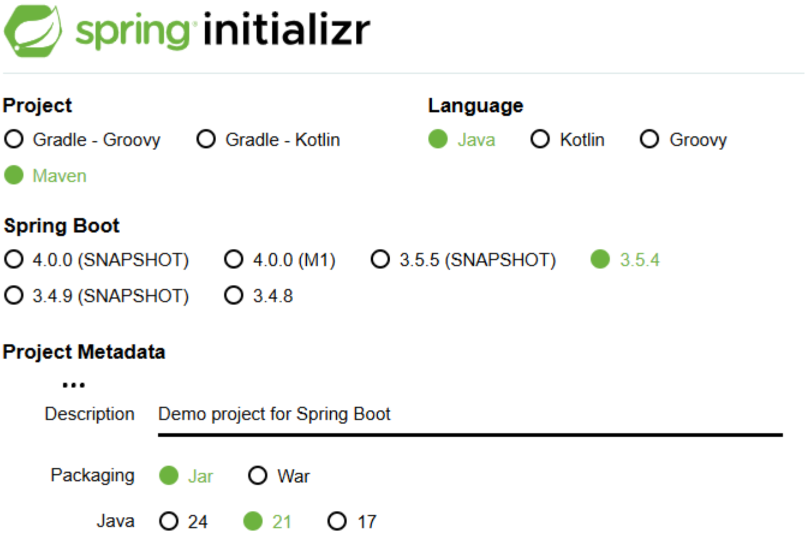
Spring Boot is a **convention-over-configuration** extension of the Spring Framework designed to simplify setup and development of Spring applications. Its core features are:

- **Auto-Configuration**, based on dependencies in your project.
- **Standalone applications** - no need to deploy to external application servers. Can run as a self-contained JAR with an embedded server (Tomcat, Jetty, or Undertow).
- **Starter dependencies** - pre-defined dependency ”starters” to simplify adding libraries: `spring-boot-starter-web` includes everything needed for a web application.
- **Opinionated defaults** - reducing the boilerplate.
- **Production-ready** features, like health checks, metrics. Supports Spring Boot Actuator for monitoring and management.
- Ideal for **building microservices**.

Spring Application Types

- **Servlet-based Application**
  - Classic web applications using Spring MVC.
  - Traditional REST APIs, MVC web apps.
  - Embedded servers supported: Tomcat, Jetty, Undertow.
- **Reactive Application**
  - Non-blocking, reactive applications using Spring WebFlux.
  - High-throughput, event-driven, non-blocking web apps.
  - Embedded servers supported: Netty, Undertow.
- **Non-Web / Command-line Application**
  - CLI apps, batch jobs, or services.
  - Background services, scripts, scheduled jobs.
  - No web server.

https://start.spring.io



spring initializr

Project

Gradle - Groovy Gradle - Kotlin [✓ Maven]

Language

[✓ Java] Kotlin Groovy

Spring Boot

4.0.0 (SNAPSHOT) 4.0.0 (M1) 3.5.5 (SNAPSHOT) [✓ 3.5.4] 3.4.9 (SNAPSHOT) 3.4.8

Project Metadata

...

Description	Demo project for Spring Boot
Packaging	[✓ Jar] War
Java	24 [✓ 21] 17

Add Dependencies

- **Developer Tools**: Spring Boot Dev Tools, Lombok
- **Web**: SpringWeb (REST, MVC, Embedded Tomcat)
- **Template Engines**: Thymeleaf, Freemarker, Mustache
- **Security**: Spring Security, OAuth2, LDAP
- **SQL**: Spring Data JPA, Liquibase, PostgreSQL Driver
- **NoSQL**: MongoDB, Cassandra, Neo4j
- **Messaging**: RabbitMQ, Kafka, WebSocket
- **I/O**: Spring Batch, Java Mail, Quartz, gRPC
- **OPS**: Actuator, Sentry
- **Observability**: Prometheus, Grafana
- **Testing**: Testcontainers
- **Spring Cloud**, **AI**, etc.

pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

A **starter** is as a pre-packaged bundle of all the necessary libraries and configurations for a particular task, all in a single dependency. Other examples: `webflux`, `security`, `actuator`, `data-jdbc`, `data-jpa`, `mail`, `kafka`, `freemarker`.

The Main Class

```java
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
        System.out.println("Hello World!");
    }
} // What does it mean: "the main class"?
```

- `@SpringBootApplication` is a composed annotation indicating a configuration class that triggers **component scanning** and **auto-configuration**.
- `SpringApplication.run` is the bootstrapping engine:
  - Prepares environment (reads application.properties)
  - Creates application context (servlet-based, reactive, non-web)
  - Loads beans and auto-configurations
  - Starts web server (if needed) and runs startup callbacks
  - Keeps running until the JVM is shut down

Running the Application

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.5.4)
Starting DemoApplication using Java 21.0.2 with PID 18176
No active profile set, falling back to 1 default profile: "default"
Devtools property defaults active!
Tomcat initialized with port 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/10.1.43]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 1226 ms
Cannot find template location: classpath:/templates/ (check Thymeleaf)
LiveReload server is running on port 35729
Tomcat started on port 8080 (http) with context path '/'
Started DemoApplication in 2.286 seconds (process running for 2.709)
Hello World!
```

Creating a Simple Web Page

- Create the class `com.example.demo.WebController`

```
@Controller // ← This class is a web component
public class WebController {
    @GetMapping("/welcome")
    public String welcome(Model model) {
        model.addAttribute("message", "Hello from Spring Boot!");
        return "welcome"; // The name of a template
    }
} // Controllers are managed by the Spring IoC container.
```

- Create a page `/resources/templates/welcome.html`

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<body>
    <h1>Welcome!</h1>
    <p th:text="${message}">Dynamic text</p>
</body>
</html>
```

```
http://localhost:8080/welcome
```

Creating a Simple REST Controller

- Create the class `HelloRestController`

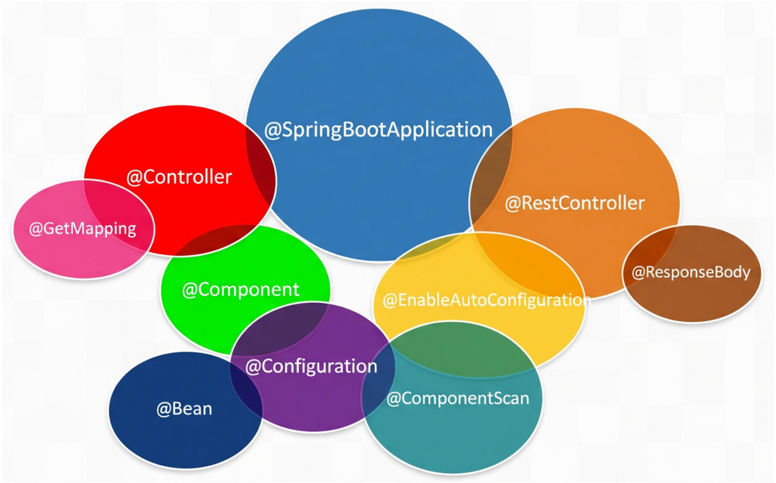
```
@RestController // ← This class handles web requests
public class HelloRestController {

    @GetMapping("/api/hello")
    public String sayHello(String name) {
        return String.format("Hello, %s!", name);
    }
}
```

- `RestController` is a **composed annotation** that is itself annotated with `@Controller` and `@ResponseBody`.
- Send an HTTP GET request

```
http://localhost:8080/api/hello?name=Joe
```

- ”Why did @RestController break up with @Controller? 😄 Because @Controller couldn’t return the right response body.”

What's With All These Annotations?



Java Annotations

- **Annotations are a form of metadata** that you can add to your Java code. They provide information to the compiler, development tools, or build frameworks.
- Can be applied to declarations of classes, methods, fields, constructors, parameters, local variables, or packages.

```
@Target(value = {ElementType.TYPE}) // ← meta-annotation
@Retention(value = RetentionPolicy.RUNTIME)
@Documented // ← marker annotation
public @interface WebServlet {
    public String[] urlPatterns() default {};
    public int loadOnStartup() default -1;
    public boolean asyncSupported() default false;    //...
}
```

```
@WebServlet(urlPatterns = {"/hello", "/demo"}, loadOnStartup=1)
public class HelloWorldServlet extends HttpServlet {...}
```

- Under the hood, an annotation is a special kind of interface, extending `java.lang.annotation.Annotation`.

Annotation Inheritance

- By default, **annotations are not inherited**.

```
@Controller
class Parent {}

class Child extends Parent {} // This is not a @Controller class
```

- Applied on another annotation, the marker meta-annotation `@Inherited` allows inheritance, **but only at class level**.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Inherited
public @interface Transactional { ... }

@Transactional
class Parent {}

class Child extends Parent {} // This is also @Transactional
```

- An annotation cannot extend another annotation.

Composed Annotations

- Composed annotations are higher-level annotations that combine multiple annotations into a single, reusable annotation.

```
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {
    @AliasFor(annotation = Controller.class)
    public String value() default ""; // single-element
}
```

```
...
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {...}
```

- Reduce boilerplate (fewer repeated annotations) and improve clarity and consistency by enforcing standard configurations.

How Are Annotations Used?

- By itself, an annotation **does nothing**.
- It needs an **annotation processor** to discover it and take it into account. This depends on the **retention policy**:
    - SOURCE: @Override, Lombok annotations;
    - CLASS: needed for bytecode analyzers;
    - RUNTIME: most of them.
- At runtime, annotations are discovered using **Reflection API**.

```
public class MyProgram {
  @Test public void m1() { }
  @Test public void m2() { throw new RuntimeException("Boom!"); }
}
```

```
for (Method m : Class.forName("MyProgram").getMethods()) {
    if (m.isAnnotationPresent(Test.class)) {
        // Do something
    }
}
```

Application Initialization

```
ApplicationContext ctx = SpringApplication.run(MyApp.class, args);
// It starts the "bootstrapping".
```

1. **Initialization**: It creates the `SpringAppplication` instance, determines the app type (Servlet, Reactive, or None). 💡
2. **Environment Preparation**: Reads properties, profiles, command-line args, application listeners are notified.
3. **Context Creation**: It creates the `ApplicationContext` object, based on the application type.
4. **Bean Definition Registration**: Main class is registered, then auto-config classes, then component scanning kicks in.
5. **Context Refresh**: Bean post-processing, instantiation of singleton beans, web server starts (if web app).
6. **Application Ready**: the "runners" are launched. 🏃

Component Scanning

- **Component scanning** is the process where Spring automatically detects and registers *beans* in the application context by looking for certain annotations in the classpath.

```
@Component
@Service, @Repository, @Controller, @RestController
Any custom annotation marked with @Component
@Configuration, @Bean
```

- The scanning is performed by classes annotated with `@ComponentScan`, such as the main class. By default, such a class scans the package where it is declared and all subpackages.

```
com.example.demo
    MyApplication.java    (@SpringBootApplication)
    service
        MyService.java    (@Service)
    controller
        MyController.java (@RestController)
com.example.other
```

Customizing Component Scan

- If your components are outside the default package tree, you need to **manually set scan base packages**:

```
@SpringBootApplication(
  scanBasePackages = {"com.example.demo", "com.example.other"})
public class MyApplication { ... }
```

- Using @ComponentScan directly:

```
@SpringBootApplication
@ComponentScan(
  basePackages = {"com.example.demo", "com.example.other"})
public class MyApplication { ... }
```

- Excluding specific components:

```
@ComponentScan(excludeFilters = @ComponentScan.Filter(
    type = FilterType.REGEX,
    pattern = "com.example.demo.test.*"))
```

- **Best practice**: Place the main class in a root package to avoid missing sub-packages.

How Is Component Scanning Implemented?

- Spring has to scan the classpath of the project looking for classes annotated with `@Component`, `@Service`, or similar.
- **Class Loading Is Expensive**: Naively, Spring could just explore all the files, load them in memory, and check the annotations.

```
Class.forName("com.example.MyComponent"); // ← Expensive!
```

- **The Solution: Bytecode Analysis**: Spring uses ASM, a fast Java bytecode manipulation/inspection library.
  Does this class have `@Component` annotation?
  1. Open the .class file.
  2. Read the bytecode.
  3. Parses the metadata with ASM.
  4. Extract annotation info, method signatures, etc.
- **Performance, Lazy Loading, Memory Savings**

Auto-Configuration

- What is the default port of the server?
- Where should I put the Thymeleaf templates?
- How to set up the connection to a database?
- How to configure things depending on runtime conditions?

- Auto-configuration is designed to solve a major problem in traditional Spring development: the extensive and often complex manual configuration required to set up an application.
- It is based on the **dependencies in the classpath and your own configuration**.
    - Add `spring-boot-starter-web`
      → Spring MVC with an embedded Tomcat.
    - Add `spring-boot-starter-data-jpa`
      → Hibernate, DataSource, JPA repositories.

How Does Auto-Configuration Work?

- **Classpath Inspection**: Checks what libraries are available.
- **Configuration Classes**: Spring Boot looks for classes annotated with `@EnableAutoConfiguration`, such as:

```
WebMvcAutoConfiguration
JacksonAutoConfiguration
ThymeleafAutoConfiguration
```

  These classes are configured by default in application.properties

```
server.port=8081
spring.thymeleaf.prefix=classpath:/my-templates/
```

- **Override Mechanism**: All these classes can be overriden. If you define your own bean, auto-config backs off.
- **Conditional Rules**: A config bean could be created only if:
  - a certain class exists on the classpath, or
  - no bean of that type already exists, or
  - some property is set in application.properties.

Custom Configuration

```
@Configuration
public class ThymeleafConfig {

  @Bean
  public SpringResourceTemplateResolver
    templateResolver(SpringResourceTemplateResolver resolver) {
    resolver.setPrefix("classpath:/my-templates/");
    return resolver;
  }
} // Overriding beans that Spring Boot auto-configures.
```

```
@Configuration
public class JacksonConfig {

  @Bean
  public ObjectMapper objectMapper() {
    ObjectMapper mapper = new ObjectMapper();
    mapper.enable(SerializationFeature.INDENT_OUTPUT); //pretty
    return mapper;
  }
} // Auto-config classes are in spring-boot-autoconfigure JAR
```

Conditional Configuration

Auto-configuration can be applied only if certain conditions are met: on property, on class, on missing bean, on web app / on not web app.

```
@Configuration
public class GreetingConfiguration {
    @Bean
    @ConditionalOnProperty(
        name = "greeting-enabled", havingValue = "true")
    public String greetingMessage() {
        return "Hello, Spring Boot!";
    }
} // Create this bean only if greeting-enabled=true in application.yml
```

```
@Configuration
public class CacheConfiguration {
    @Bean
    @ConditionalOnClass(org.ehcache.Cache.class)
    public String cacheService() {
        return "Ehcache-based CacheService is enabled!";
    }
} // Create this bean only if Cache class exists on classpath
```

What Exactly Is a Bean?

- A **bean** is an object **managed by the Spring IoC** container. Spring keeps track of these beans, handles their lifecycle, and can inject them into other beans.
- @Bean is a **method-level** annotation.

```java
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
        // the returned object becomes a Spring bean
        // by default, the bean name is the method name.
    }
}
```

- The default **scope is singleton**. 💡
  We'll talk more about this topic later.

Externalized Configuration

- **Externalized configuration** means putting configuration values outside of your compiled code, so you can change them without rebuilding the app.
- The configuration is applied from highest to lowest priority:
    1. Command-line args
    2. Java system properties (-D)
    3. Environment variables
    4. application-{profile}.properties
    5. application.properties
    6. Default values in your code

```
# application.properties
welcome-message=Welcome from config!
```

```
// Reading configuration in code
// "Hello" is the default message, if not set externally
@Value("${welcome-message:Hello}")
private String welcomeMessage;
```

application.properties

- Central configuration file for Spring Boot applications
- Format: key=value pairs

```
server.port=8080
spring.application.name=my-app
logging.level.org.springframework=DEBUG
```

- Location: `src/main/resources/application.properties` or external paths.
- Support property placeholders

```
app.home=/home/user/app
app.temp=${app.home}/temp
```

- Supports profiles and overrides from other externalized configuration sources.

Spring Expression Language (SpEL)

- **Query and manipulate** objects at runtime.
- Perform **conditional logic** within Spring configurations.
- Similar to **Unified EL**, used in Java EE, or **OGNL** (Object-Graph Navigation Language), used in Apache Struts.

```
@Component
public class MyComponent {
    @Value("#{systemProperties['user.name']}")
    private String currentUser;

    @Value("#{person.age > 18 ? 'Adult' : 'Minor'}")
    private String category;

    @Value("#{person?.address?.city ?: 'City not available'}")
    private String city; // Elvis operator ?:

    @Value("#{T(java.lang.Math).random() * 100}")
    private double employeeCode;   // Invoke static methods
}
```

Configuring Bean Definitions

SpEL can be used inside your bean definitions (via annotations like `@Value`) to dynamically compute values instead of hardcoding them.

```
@Configuration
public class AppConfig {

    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Alice");
        return p;
    }

    @Bean
    public Employee employee(
            @Value("#{person.name}") String managerName) {

        Employee emp = new Employee();
        emp.setManagerName(managerName); // gets "Alice"
        return emp;
    }
}
```

Storing Configuration in the Environment

Everything that is likely to vary between deployments — e.g., database URLs, API keys, credentials, feature flags, etc. should be **stored in the environment, not hard coded**.

- No secret credentials in Git.
- Easy to deploy the same artifact to multiple environments.
- No rebuild required to change config.
- Works well in cloud-native CI/CD pipelines (Docker env vars, Kubernetes ConfigMaps/Secrets).

```
// Bad
String dbPassword = "hardcodedSecret";

// Good
@Value("${DB_PASSWORD}")
String dbPassword;
```

Using @ConfigurationProperties

- How to load many related properties into a POJO object?

```
app.name=DemoApp
app.version=1.0.0
```

- `@ConfigurationProperties` binds external configuration properties directly to a Java class, instead of using multiple `@Value` annotations.

```java
@Component
@ConfigurationProperties(prefix = "app")
public class AppProperties {

    private String name;
    private String version;

    // Getters and setters
}
```

Spring Boot Profiles

- **Problem**: You don't want the same DB URL, credentials, or logging level in development as in production.
- **Profiles** provide a way to group and activate different configurations for different environments (dev, test, prod).
- Profile-specific **property files**

```
# application.properties
spring.profiles.active=dev,test   # active profile(s)
```

```
# application-dev.properties      # overrides for 'dev'
spring.datasource.url=jdbc:h2:mem:test-db
logging.level.root=DEBUG
```

```
# application-prod.properties     # overrides for 'prod'
spring.datasource.url=jdbc:postgresql://db.server.com/prod-db
logging.level.root=INFO
```

- Profiles can be activated using command line arguments, environment variables or annotation in test classes.

Conditional Values Based on Profiles

```
#application-dev.properties
app.message=Hello from DEV

#application-prod.properties
app.message=Hello from PROD
```

```
@Component
public class ProfileBasedBean {

    @Value("${app.message}")
    private String message;

    public String getMessage() {
        return message;
    }
} // You can also use SpEL: "#{@environment.getActiveProfiles()}"
```

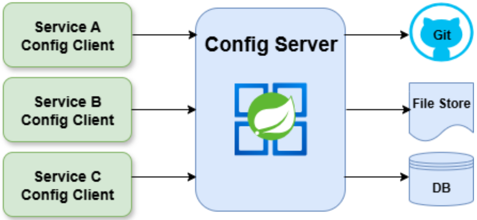
```
java -jar myapp.jar --spring.profiles.active=dev  → Hello from DEV
java -jar myapp.jar --spring.profiles.active=prod → Hello from PROD
```

Conditional Beans with @Profile

```
@Configuration
public class AppConfig {

    @Bean
    @Profile("dev")  // Only in dev
    public DataSource devDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .build();
    }

    @Bean
    @Profile("prod")  // Only in prod
    public DataSource prodDataSource() {
        return DataSourceBuilder.create()
            .url("jdbc:postgresql://db.server.com/prod-db")
            .username("user")
            .password("secret")
            .build();
    }
} // @Profile can also be used on @Component classes.
```

Managing Configurations in Distributed Systems

- What happens if we have to deploy services on multiple machines and we have to change their configurations?
- **Spring Cloud Config** solves the ”configuration management problem” in the microservice architectures.
- A **Config Server** is a centralized service that manages and distributes configuration settings.
- A **Config Client** is a service or application that fetches its configuration settings from a Config Server instead of local files.



Command Line Arguments & Runners

- A Spring Boot app starts once we call the run method.

```
ApplicationContext ctx = SpringApplication.run(MyApp.class, args);
```

- The application may have **command line arguments**.

```
java -jar myapp.jar --version=1.0 hello world
version=1.0 is a key/value option
hello world are non-option arguments
```

- Specific pieces of code may need to be executed after **the application context has been fully initialized**: configuration checks, running scripts, or starting background tasks.
- After the initial component scan, Spring executes the code in any component that implements the functional interfaces `CommandLineRunner` or `ApplicationRunner`.

The "Running" Beans

- **CommandLineRunner**

```
@Component
@Order(1) // optional, used if there are multiple runners
public class MyAppRunner implements CommandLineRunner {
  @Override
  public void run(String... args) throws Exception {
    System.out.println("Running startup logic...");
  }
}
```

- **ApplicationLineRunner** - for better handling of arguments.

```
@Component
public class MyAppRunner implements ApplicationRunner {
  @Override
  public void run(ApplicationArguments args) {
    if (args.containsOption("version")) {
      System.out.println(args.getOptionValues("version"));
    }
  }
}
```

Spring Boot Actuator

- *Actuator*: a device that acts on a system to make something happen, like a lever, motor, or valve that causes motion or change in response to some input.
- The Spring Boot Actuator module provides endpoints that allow you to "act" on your application or observe its internal state. → It provides features for **monitoring and managing** your app.
- **Monitoring**: Exposes runtime info: health, metrics, beans, environment, heap dumps, thread dumps.
- **Management**: Allows safe operational actions: enable/disable features, trigger refresh, shutdown, or restart (in dev/test).
- You can trigger changes or retrieve state programmatically (using `Endpoint` beans) or via HTTP endpoints.
- It is essential for observability and operational control in microservice architectures.

Using Spring Boot Actuator

- Add the dependency: `spring-boot-starter-actuator`.
- Decide which endpoints to expose and enable them (if required).

```
management.endpoints.web.exposure.include=info,health,metrics
management.info.env.enabled=true
```

- Access built-in endpoints, via HTTP or JMX.

```
http://localhost:8080/actuator
    /info
    /metrics/jvm.memory.used
```

- Additional endpoints may be added.

```
@Component
@Endpoint(id = "other") // exposed at /actuator/other
public class OtherEndpoint {
    @ReadOperation
    public Map<String, Object> getOtherValues() {
        return Map.of("greeting", "Hello Actuator!");
    }
}
```

Very Short Introduction to YAML

- YAML = **Y**et **A**nother **M**arkup **L**anguage
  YAML = **Y**AML **A**in't **M**arkup **L**anguage
- Human-friendly format for configuration files.
- Uses indentation, no braces or commas.
- Key–Value Pairs

```
# server.port = 8081
# becomes
server:
  port: 8081
```

- Hierarchical / Nested Structure, Lists / Arrays

```
# my.servers[0]=host1.example.com
# my.servers[1]=host2.example.com
# becomes
my:
  servers:
    - host1.example.com
    - host2.example.com
```

Properties or YAML?

- **application.properties** (simple key-value pairs)

```
server.port=8081
spring.datasource.url=jdbc:postgresql://localhost/mydb
spring.datasource.username=root
spring.datasource.password=secret
logging.level.org.springframework=DEBUG
```

- **application.yml** (hierarchical, recommended for nested configs)

```
server:
  port: 8081  # usually use 2 spaces for indentation (no tabs!)

spring:
  datasource:
    url: jdbc:postgresql://localhost/mydb
    username: user
    password: secret

logging:
  level:
    org.springframework: DEBUG
```

Dependency Management

- **Dependency management** refers to how your project manages external libraries (dependencies) and their versions.
- **Explicitly declare and isolate dependencies.**
- Spring Boot uses a **Bill of Materials (BOM)** approach.
  A BOM is a Maven POM file (pom.xml) that lists dependency versions but does not actually include the dependencies itself.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>3.5.4</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- This **locks versions** for all common dependencies.

Inheriting from a Parent POM

- A **parent POM** is a POM that your project inherits from. It provides **default dependency versions**, plugin configurations, build settings and other shared configurations.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.5.4</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

- When using a **starter**, you don't need to specify version.

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