

Spring and SpringBoot

Training Agenda

Spring & SpringBoot Introduction

Spring Initializr Project

Dependency Injection

AutoConfig

Powerful Annotations

Maven – Dependency Config

Logging

Documentation

Spring Web

Building RESTFul APIs

H2 Database

JPA

Spring Security

Writing Tests

Monitoring App

Spring Framework

Beans, Components, Coupling and More

Spring Framework

Developed by Rod Johnson in 2003

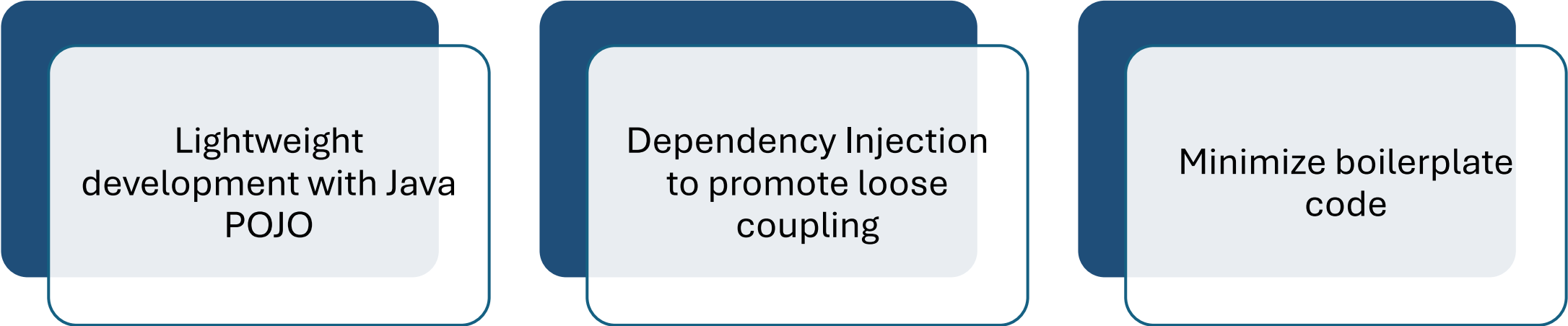
It can be thought of as a *framework of frameworks*

The Spring Framework is a mature, powerful and highly flexible framework focused on building web applications in Java.

Spring project takes care of most of the low-level aspects of building the application to allow us to focus on features and business logic.

Very actively maintained and has a thriving dev community.

Goals of Spring



Lightweight
development with Java
POJO

Dependency Injection
to promote loose
coupling

Minimize boilerplate
code

Why Spring?

Usability

Easy for developers to start and then configure exactly what they need.

Modularity

We have options to use the entire Spring framework or just the modules necessary.

Conformance

Support over the standard specification where necessary. For instance, Spring supports JPA based repositories and hence makes it trivial to switch providers.

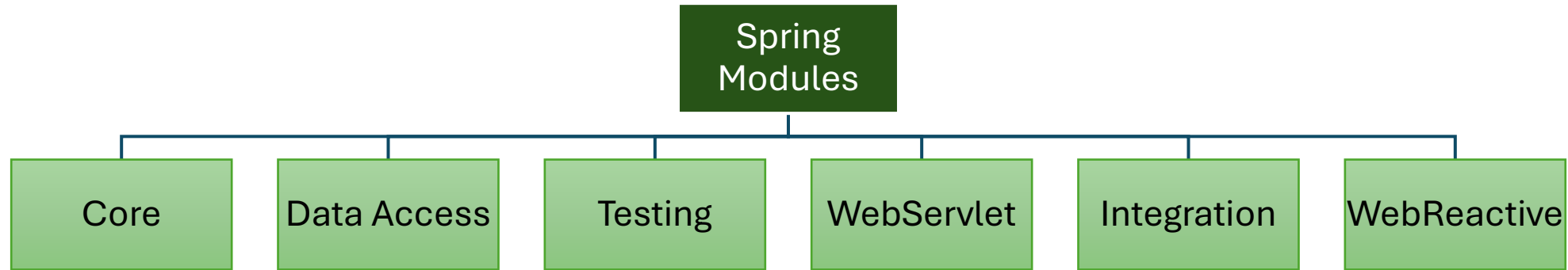
Testability

Spring application is mostly composed of POJOs which naturally makes unit testing relatively much simpler.

Maturity

Spring has a long history of innovation, adoption, and standardization. Over the years, it's become mature enough to become a default solution for most common problems faced in the development of large-scale enterprise applications

Spring Modules

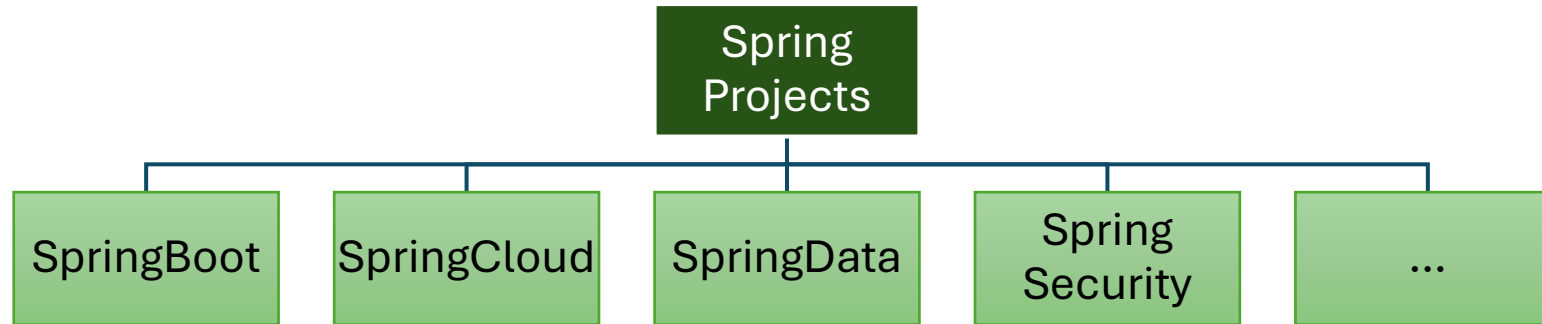


Spring Framework is divided into modules:

- Core: IoC Container etc
- Testing: Mock Objects, Spring MVC Test etc
- Data Access: Transactions, JDBC, JPA etc
- Web Servlet: Spring MVC etc
- Web Reactive: Spring WebFlux etc Integration: JMS etc

Each application can choose the modules they want to make use of.

Spring Projects



Spring Projects: Spring keeps evolving (REST API > Microservices > Cloud)

- Spring Boot: Most popular framework to build microservices
- Spring Cloud: Build cloud native applications
- Spring Data: Integrate the same way with different types of databases : NoSQL and Relational
- Spring Integration: Address challenges with integration with other applications
- Spring Security: Secure your web application or REST API or microservice

Why is Spring Popular?

Loose Coupling: Spring manages beans and dependencies

- Make writing unit tests easy!
- Provides its own unit testing project - Spring Unit Testing

Reduced Boilerplate Code: Focus on Business Logic

- Example: No need for exception handling in each method!
- All Checked Exceptions are converted to Runtime or Unchecked Exceptions

Architectural Flexibility: Spring Modules and Projects

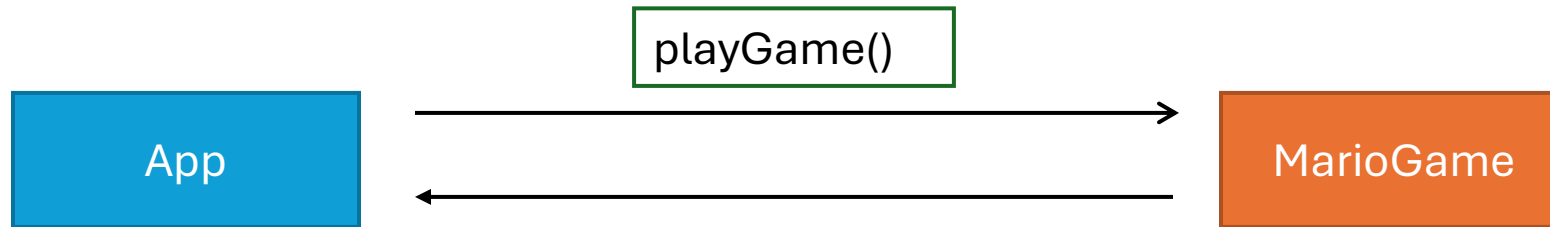
- You can pick and choose which ones to use (You DON'T need to use all of them!)

Evolution with Time: Microservices and Cloud

- Spring Boot, Spring Cloud etc!

Inversion of Control

The approach of outsourcing the construction and management of objects.

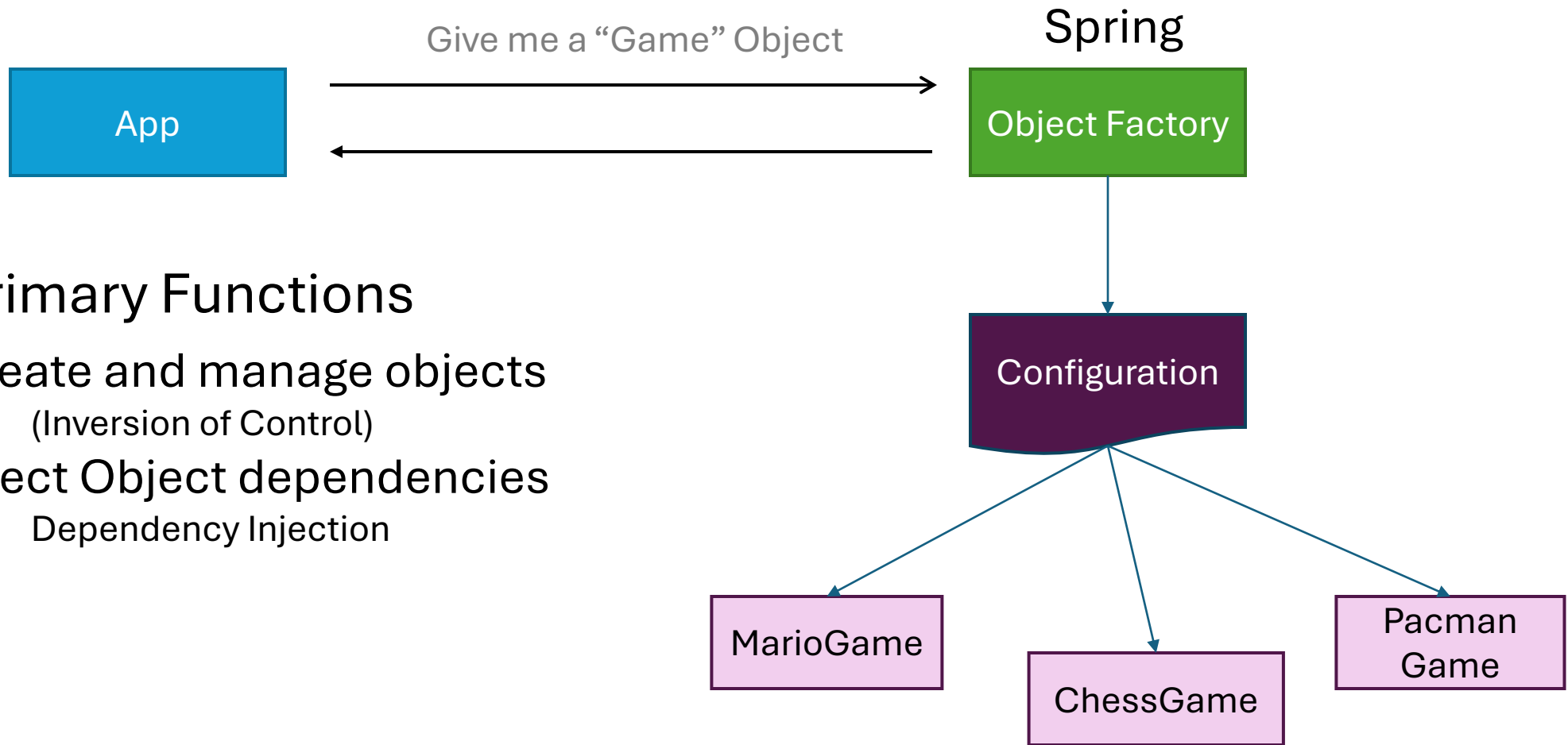


App should be configurable

Easily change the game

- ChessGame, SuperContra, Pacman etc

Spring Container



❑ Primary Functions

- ✓ Create and manage objects
 - (Inversion of Control)
- ✓ Inject Object dependencies
 - Dependency Injection

Dependency Injection/Inversion Principle

The client delegates to another object the responsibility of providing its dependencies.

Spring Dependency Injection Types

Constructor based

- Dependencies are set by creating the Bean using its Constructor

Setter based

- Dependencies are set by calling setter methods on your beans

Field based

- No setter or constructor. Dependency is injected using reflection

Spring team recommends Constructor-based injection as dependencies are automatically set when an object is created.

Spring Autowiring

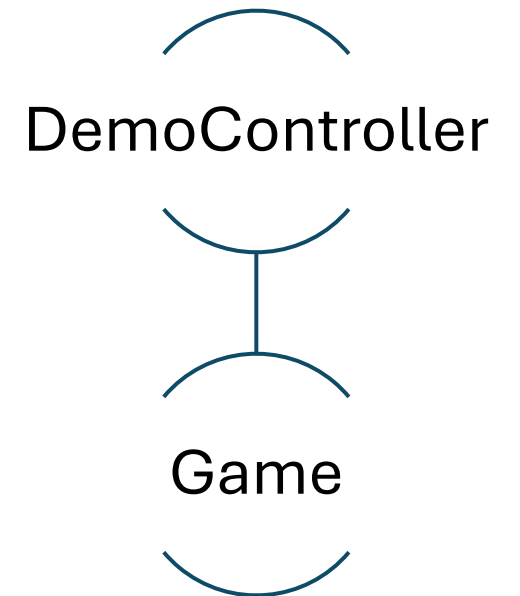
For dependency injection, Spring can use Autowiring.

Spring will look for a class that matches -

- Match by type: class or interface

Spring will inject it automatically... hence it is Autowired.

- Inject a Game Implementation
- Spring will scan for @Components
- Anyone implements the Game Interface?
- If so, let's inject them. For example : PacmanGame



Configuring Spring Container

Java
Annotations
(Modern)

Java Source
Code
(Modern)

XML
Configuration
file
(Legacy)

Maven

A Tool For Project Management

Maven

Maven is a Project
Management tool

Most popular use of Maven
is for *build management*
and dependencies

What Problem Does Maven Solve?

Approach 01

When building your Java project, you may need additional JAR files. For example: Spring, Hibernate, Commons Logging, JSON etc...

One approach is to download the JAR files from each project web site

usually add the JAR files to your build path / classpath

Approach 02 – Maven Solution

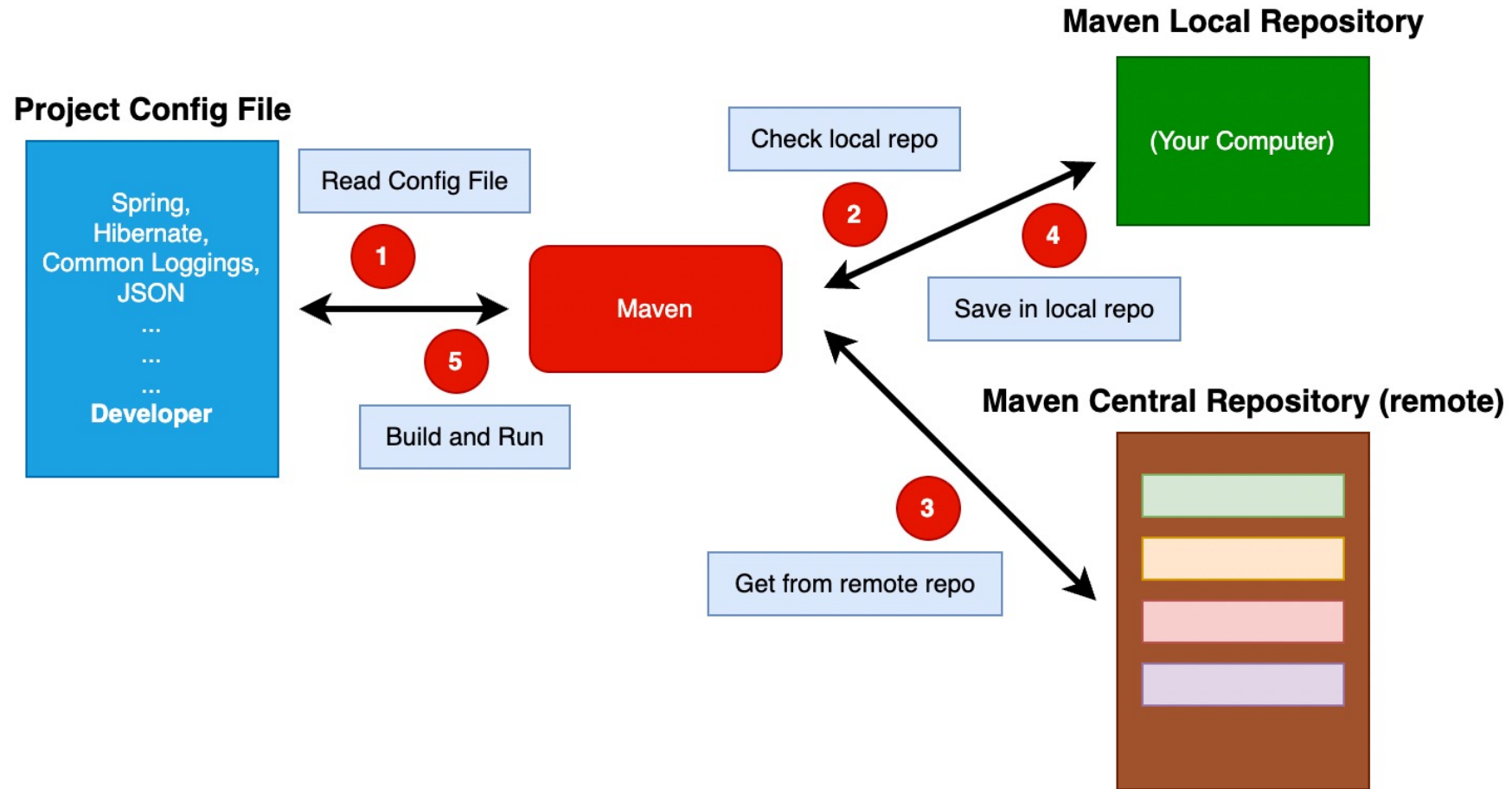
Tell Maven the projects you are working with (dependencies) - Spring, Hibernate etc

Maven will go out and download the JAR files for those projects for you

And Maven will make those JAR files available during compile/run

Think of Maven as your friendly helper / personal shopper

Maven – How It Works?



Maven: Advantages

Dependency Management

- Maven will find JAR files for you
- No more missing JARs

Building and Running your Project

- No more build path / classpath issues

Standard directory structure

Spring Boot

Production Ready Applications... Quickly!

Spring Boot

Spring Boot is a framework that *simplify the development and deployment* of Java applications, including Microservices. With Spring Boot, you can create *self-contained, executable JAR files* instead of traditional WAR or EAR files. These JAR file contains all the *dependencies and configurations* required to run the application. This approach eliminates the need for external application or web server.

Why Spring Boot?

Provides a range of built-in features and integrations such as auto-configuration, dependency injection and support for various cloud platforms.

Provides an embedded server which can run the application directly without the need for external server installation.

In-built support for production ready features such as metrics, health check and externalized configurations.

We can quickly bootstrap the project and start coding with a range of starter dependencies that provide pre-configured settings for various components such as databases, queues etc.

Well suited for cloud native development, It integrated smoothly with cloud platforms like Kubernetes, provides support for containerization and enable seamless deployment to popular cloud providers.

World Before SpringBoot

Setting up Spring Web Projects before SpringBoot was NOT easy!

- Define **maven dependencies** and manage versions for frameworks
 - spring-webmvc, jackson-databind, log4j etc
- Define **web.xml** (/src/main/webapp/WEB-INF/web.xml)
 - Define Front Controller for Spring Framework (DispatcherServlet)
- Define a **Spring context XML file** (/src/main/webapp/WEB-INF/todo-servlet.xml)
 - Define a Component Scan (<context:component-scan base-package="base.package.name" />)
- Install **Tomcat** or use tomcat7-maven-plugin plugin (or any other web server)
- **Deploy** and **Run** the application in Tomcat

How does SpringBoot do its Magic?

- Spring Boot Starter Projects
- Spring Boot Auto Configuration

SpringBoot Starter Project

Help you get a project up and running quickly

- Web Application - Spring Boot Starter Web
- REST API - Spring Boot Starter Web
- Talk to database using JPA - Spring Boot Starter Data JPA
- Talk to database using JDBC - Spring Boot Starter JDBC
- Secure your web application or REST API - Spring Boot Starter Security

Manage list of maven dependencies and versions for different kinds of apps:

- SpringBoot Starter Web: Frameworks needed by typical web applications
 - spring-webmvc, spring-web, spring-boot-starter-tomcat, spring-boot-starter-json

Spring Boot Auto Configuration

Spring Boot provides Auto-Configuration

- **Basic configuration** to run your application using the frameworks defined in your maven dependencies
- **Auto Configuration is decided based on:**
 - Which frameworks are in the Class Path?
 - What is the existing configuration (Annotations etc)?
- **An Example:** (Enable debug logging for more details): If you use Spring Boot Starter Web, following are auto configured:
 - Dispatcher Servlet (DispatcherServletAutoConfiguration)
 - Embedded Servlet Container - Tomcat is the default (EmbeddedWebServerFactoryCustomizerAutoConfiguration)
 - Default Error Pages (ErrorMvcAutoConfiguration)
 - Bean to/from JSON conversion (JacksonHttpMessageConvertersConfiguration)

```
spring-boot-autoconfigure-2.4.4.jar - /Users/rangakaranam/.m2/re
└─ org.springframework.boot.autoconfigure
   └─ org.springframework.boot.autoconfigure.admin
      └─ org.springframework.boot.autoconfigure.amqp
         └─ org.springframework.boot.autoconfigure.aop
            └─ org.springframework.boot.autoconfigure.availability
               └─ org.springframework.boot.autoconfigure.batch
                  └─ org.springframework.boot.autoconfigure.cache
                     └─ org.springframework.boot.autoconfigure.cassandra
                        └─ org.springframework.boot.autoconfigure.codec
                           └─ org.springframework.boot.autoconfigure.condition
                              └─ org.springframework.boot.autoconfigure.context
                                 └─ org.springframework.boot.autoconfigure.couchbase
                                    └─ org.springframework.boot.autoconfigure.dao
                                       └─ org.springframework.boot.autoconfigure.data
                                          └─ org.springframework.boot.autoconfigure.data.cassandra
                                             └─ org.springframework.boot.autoconfigure.data.couchbase
                                                └─ org.springframework.boot.autoconfigure.data.elasticsearch
                                                   └─ org.springframework.boot.autoconfigure.data.jdbc
                                                      └─ org.springframework.boot.autoconfigure.data.jpa
                                                         └─ org.springframework.boot.autoconfigure.data.ldap
                                                            └─ org.springframework.boot.autoconfigure.data.mongo
                                                               └─ org.springframework.boot.autoconfigure.data.neo4j
                                                                  └─ org.springframework.boot.autoconfigure.data.r2dbc
                                                                     └─ org.springframework.boot.autoconfigure.data.redis
                                                                        └─ org.springframework.boot.autoconfigure.data.rest
                                                                           └─ org.springframework.boot.autoconfigure.data.solr
                                                                              └─ org.springframework.boot.autoconfigure.data.web
                                                                                 └─ org.springframework.boot.autoconfigure.diagnostics.analyzer
                                                                                    └─ org.springframework.boot.autoconfigure.domain
                                                                                       └─ org.springframework.boot.autoconfigure.elasticsearch
                                                                                          └─ org.springframework.boot.autoconfigure.elasticsearch.rest
                                                                                             └─ org.springframework.boot.autoconfigure.flyway
                                                                                                └─ org.springframework.boot.autoconfigure.freemarker
                                                                                                   └─ org.springframework.boot.autoconfigure.groovy.template
                                                                                                      └─ org.springframework.boot.autoconfigure.gson
                                                                                                         └─ org.springframework.boot.autoconfigure.h2
                                                                                                            └─ org.springframework.boot.autoconfigure.hateoas
                                                                                                               └─ org.springframework.boot.autoconfigure.hazelcast
                                                                                                                  └─ org.springframework.boot.autoconfigure.http
                                                                                                                     └─ org.springframework.boot.autoconfigure.http.codec
```

Annotations

@SpringBootApplication is composed of the following annotations -

Annotation	Description
@EnableAutoConfiguration	Enable Spring Boot's auto-configuration support
@ComponentScan	Enable component scanning of the current package. Also recursively scans sub-packages
@Configuration	Able to register extra beans with @Bean or import other configuration classes

@Component Annotation

@Component marks the class as a Spring Bean

- Spring Bean is just a regular Java class which is managed by spring

@Component also make the bean available for dependency injection

SpringBoot Embedded Servers

How do you deploy your Application?

- Step 1 : Install Java
- Step 2 : Install Web/Application Server
 - Tomcat/WebSphere/WebLogic etc
- Step 3 : Deploy the application WAR (Web ARchive)
 - This is the OLD WAR Approach
 - Complex to setup!

Embedded Server – Simpler Approach

- Step 1 : Install Java
- Step 2 : Run JAR file
- Make JAR not WAR (Credit: Josh Long!)
- Embedded Server
 - Examples: `spring-boot-starter-tomcat` `spring-boot-starter-jetty` `spring-boot-starter-undertow`

Spring Boot vs Spring MVC vs Spring

Spring Framework

Core feature – Dependency Injection

- @Component, @Autowired, IOC Container, ApplicationContext, Component Scan etc..
- Spring Modules and Spring Projects: Good Integration with Other Frameworks (Hibernate/JPA, JUnit & Mockito for Unit Testing)

Spring MVC

Build web applications in a decoupled approach

- Dispatcher Servlet, ModelAndView and View Resolver etc

Spring Boot

Build production ready applications quickly

- Starter Projects - Make it easy to build variety of applications
- Auto configuration - Eliminate configuration to setup Spring, Spring MVC and other projects
- Enable production ready nonfunctional features

REST API - Architectural Style for the Web

Resource: Any information (Example: Courses)

URI: How do you identify a resource?
(/courses, /courses/1)

You can perform actions on a resource-
(Create/Get/Delete/Update)

Representation: How is the resource represented?
(XML/JSON/Text/Video etc..)

Different HTTP Request Methods
are used for different operations:

- GET - Retrieve information
- POST - Create a new resource
- PUT - Update/Replace a resource
- PATCH - Update a part of the resource
- DELETE - Delete a resource

Different Annotations & Classes That Supports Building REST Services

@RestController

Can be used to put on top of a class. This will expose your methods as REST APIs. Developers can also use @Controller + @ResponseBody for same behaviour

@ResponseBody

Can be used on top of a method to build a REST API when we are using @Controller on top of a class.

@ControllerAdvice

Along with @ExceptionHandler, this can be used to handle exceptions globally.

We have another annotation @RestControllerAdvice which is same as @ControllerAdvice + @ResponseBody

@RequestHeader & @RequestBody

Is used to receive request header and body individually

ResponseEntity<T> Class

Allows developer to send response body, status and headers on the HTTP response

RequestEntity<T> Class

Allows developer to receive the request body and headers in a HTTP request

Resolving issue for multiple implementations

In case of multiple class implementations -

- Resolve it using @Primary with class
- Resolve it using @Qualifier on constructor injections

@Qualifier has the higher priority. Also recommended.

Bean Initialization

By default, all beans are initialized, when your application starts. For example, `@Component` etc....

Spring will create an instance of all beans and make them available.

Instead of creating all beans upfront, we can specify lazy initialization by adding `@Lazy` annotation to a given class

Lazy Initialization

A bean will only be initialized in the following cases –

- It is needed for dependency injection
- Or it is explicitly requested

All beans can be configured lazy using global configuration-

- `spring.main.lazy-initialization=true`

Bean Scope

Scope refers to the life cycle of a bean

- How long does the bean live?
- How many instances are created?
- How is the bean shared?

Default scope of a bean is Singleton

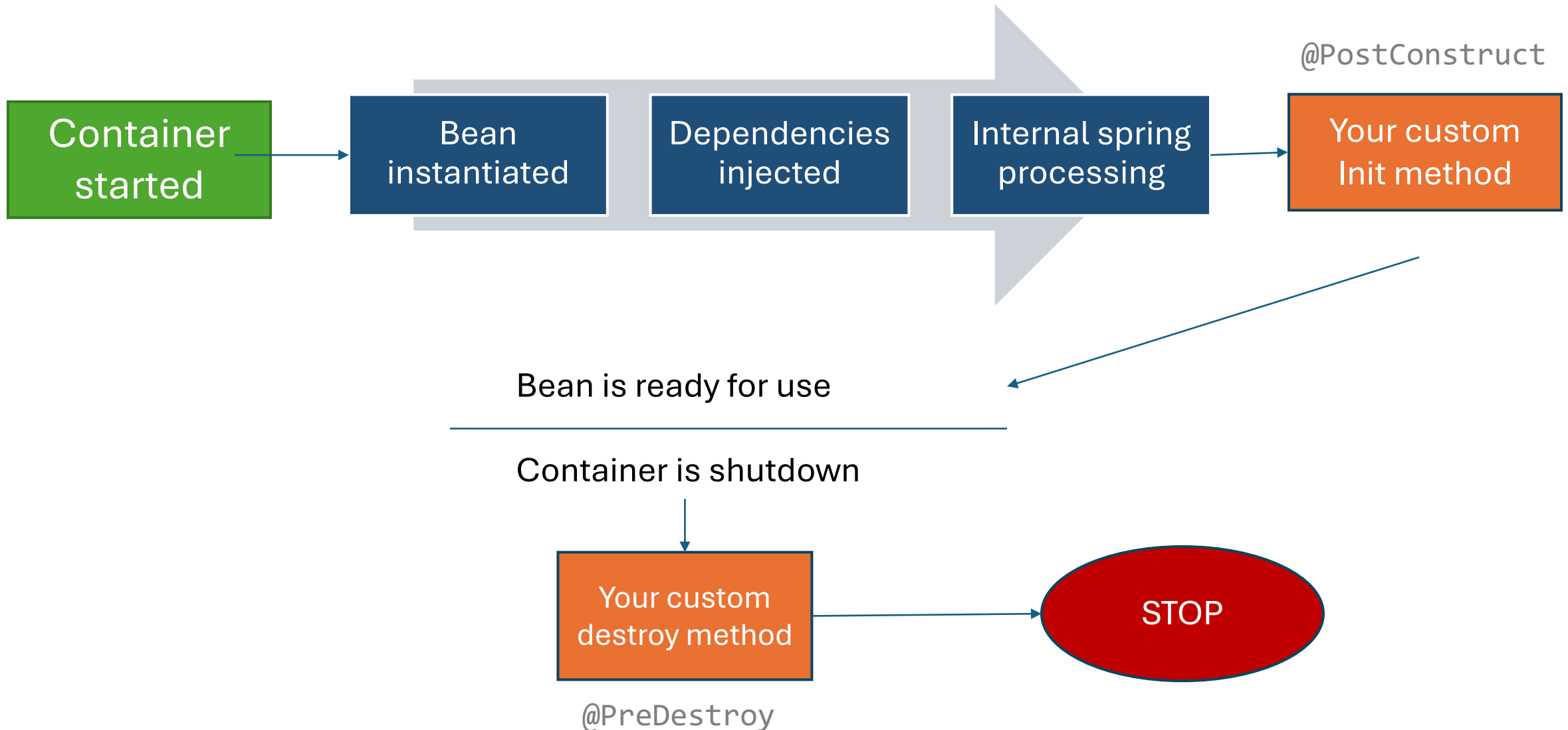
- Spring container created only one instance of the bean, by default.
- It is cached in memory
- All dependency injections for the bean will reference the SAME bean

Additional Spring Bean Scopes

Scope	Description
Singleton	Creates a single shared instance of the bean. Default Scope.
Prototype	Creates a new bean instance for each container request/injection.
Request	Scoped to an HTTP request. Used for Web Apps only.
Session	Scoped to an HTTP web session. Used for Web Apps only.
Global-session	Scoped to a global HTTP web session. Used for Web Apps only.

```
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

Bean Life Cycle Methods



Note on Prototype Scope

In contrast to the other scopes, Spring does NOT manage the complete life cycle of a prototype bean -

- For “prototype” scoped beans, spring does not call the destroy method

Prototype beans are lazy by default.

- There is no need to use the `@Lazy` annotation for prototype scoped beans

What is JPA?

Jakarta Persistence API(JPA) ... previously known as Java Persistence API

- Standard API for Object-to-Relational-Mapping(ORM)

JPA is only specification

- Defines a set of interfaces
- Requires an implementation to be usable
 - You are NOT locked to vendor implementation
 - If one vendor stops supporting their product, you can switch to another vendor easily

Maintain portable, flexible code by coding to JPA spec(interfaces)

Hibernate / JPA uses JDBC for all database communications.

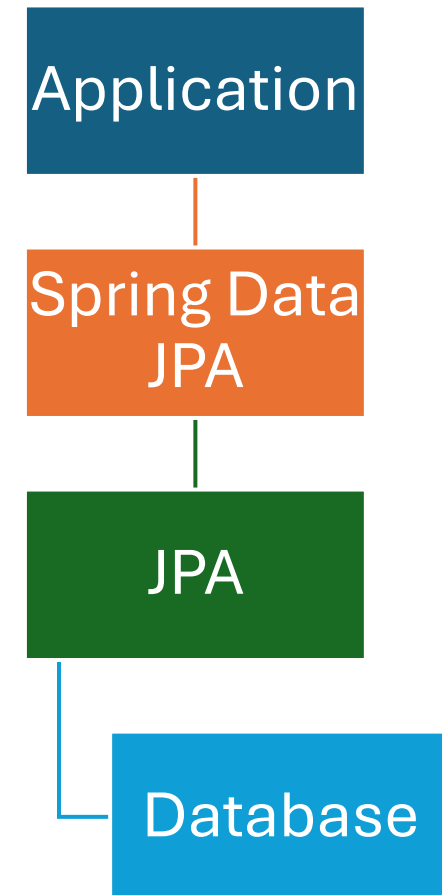
Spring Boot Auto Configuration Magic – Data JPA

Just add data JPA and h2 dependencies

- Spring Boot Auto Configuration does some magic:
 - Initialize JPA and Spring Data JPA frameworks
 - Launch an in memory database (H2)
 - Setup connection from App to in-memory database
 - Launch a few scripts at startup (example: data.sql)

Remember - H2 is in memory database

- Does NOT persist data
- Great for learning
- BUT NOT so great for production



JDBC to Spring JDBC to JPA to Spring Data JPA

JDBC

- Write a lot of SQL queries!
- And write a lot of Java code

Spring JDBC

- Write a lot of SQL queries
- BUT lesser Java code

JPA

- Do NOT worry about queries
- Just Map Entities to Tables!

Spring Data JPA

- Let's make JPA even more simple!
- I will take care of everything!

Documentation

REST API Documentation

Documentation - OpenAPI Specification

OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API:

Available endpoints (/books) and operations on each endpoint (GET /books, POST /books)

Operation parameters Input and output for each operation

Authentication methods

Contact information, license, terms of use, and other information.

Swagger

An open-source framework for API development

Provides a standardized way to describe, produce, consume, and visualize RESTful web services

The Swagger ecosystem includes several tools: Swagger UI, Swagger Editor, Swagger Codegen

Include Maven Dependency:

- `springdoc-openapi-starter-webmvc-ui`

URL to View swagger:

`http://localhost:8080/swagger-ui/index.html`

Distributed Tracing

Let's follow the trails

Distributed Tracing

Zipkin is an open-source distributed tracing system that helps gathering timing data

Distributed tracing allows tracking the entire journey of a request and identifying bottlenecks

Setting Up Zipkin Server

Download Zipkin server from -

- <https://zipkin.io/pages/quickstart>

Run the zipkin server by running following in terminal -

- `java -jar zipkin-server-exec.jar`

Add following dependencies in POM file -

- `spring-boot-starter-actuator`
- `micrometer-tracing-bridge-otel`
- `opentelemetry-exporter-zipkin`

Add following properties in application properties file -

- `Management.tracing.sampling.probability=1.0`
- `management.zipkin.tracing.endpoint=http://localhost:9411/api/v2/spans`

URL for accessing Zipkin server:

- `http://127.0.0.1:9411/zipkin/`