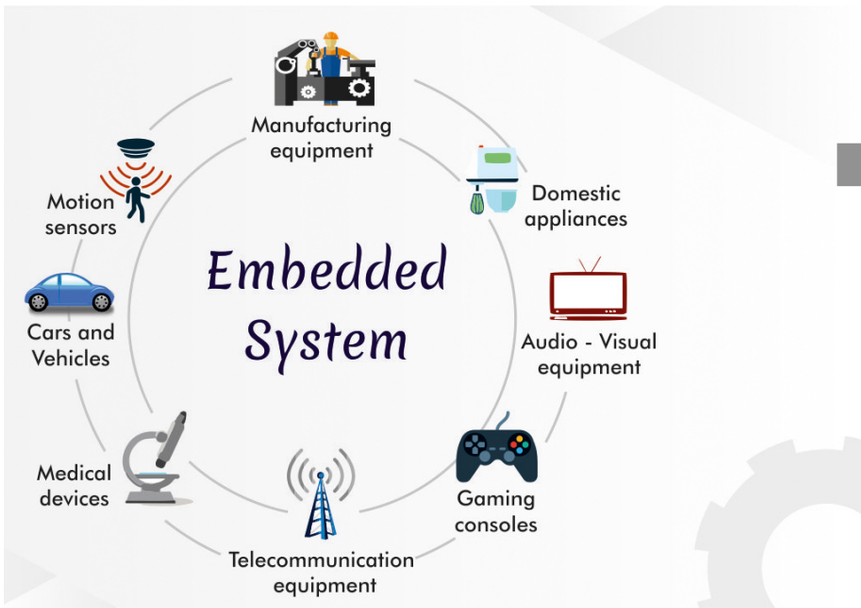
Spring framework

Core Concepts

The Problem

Clients

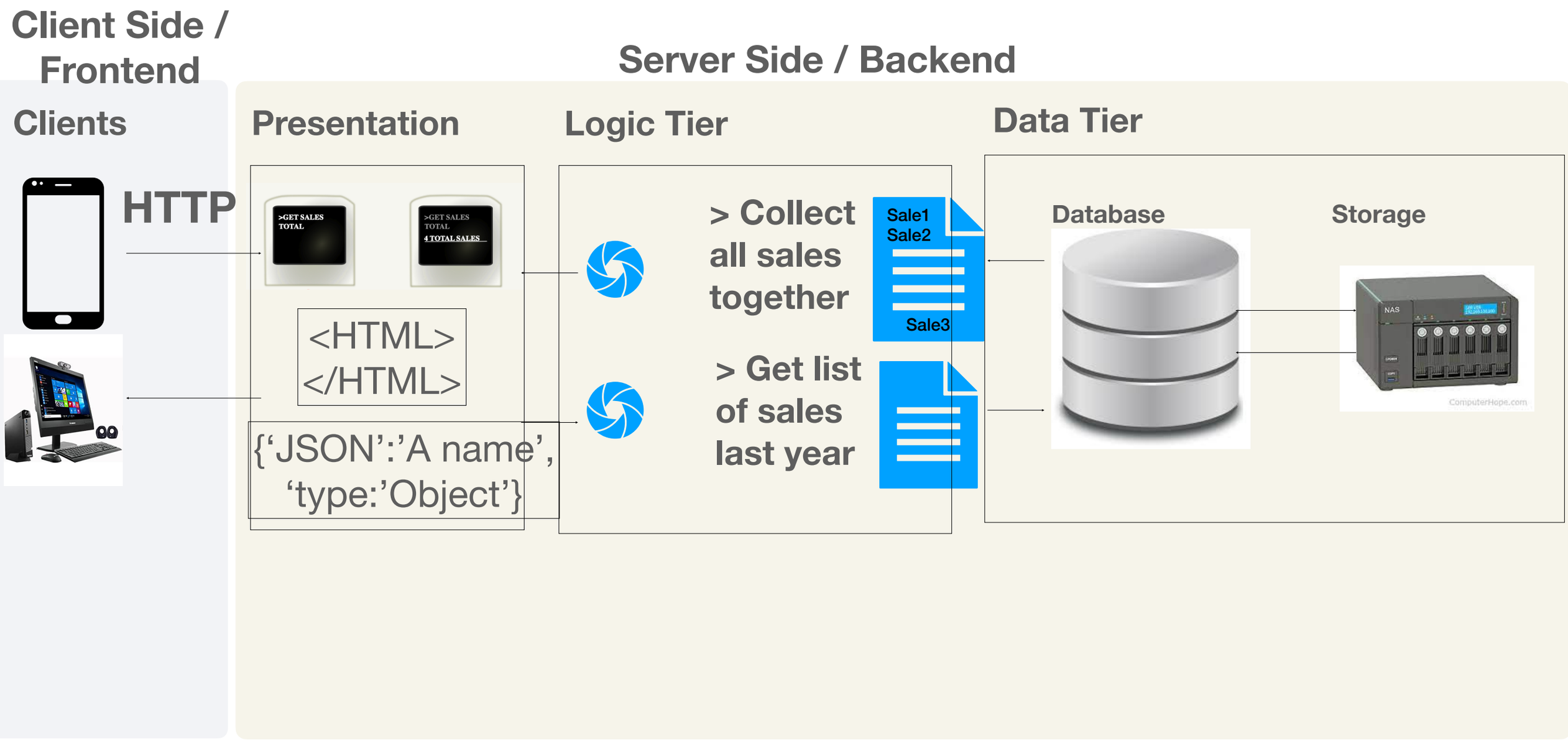


Internet

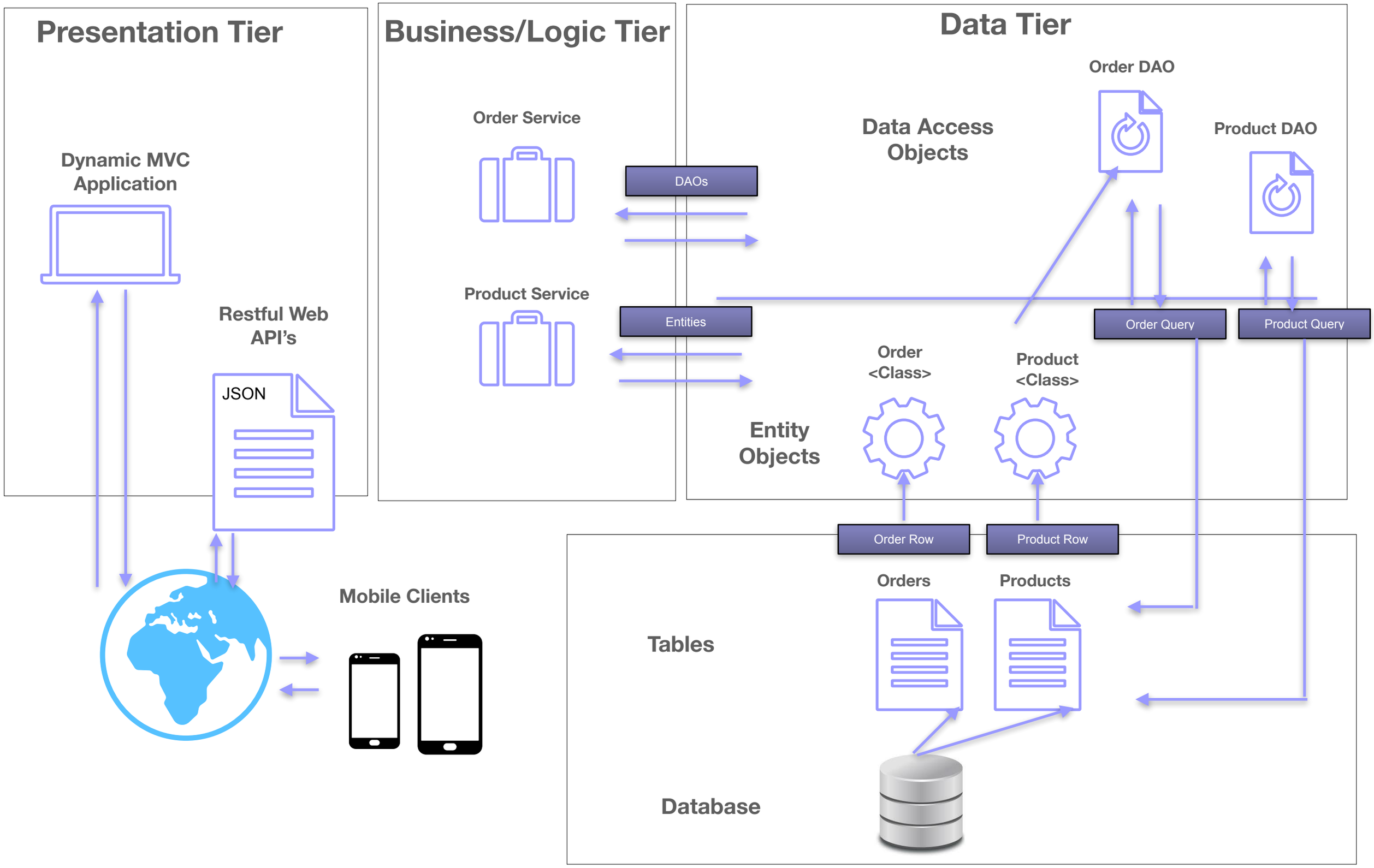
Datasource

MySql
MongoDb
MSSQL
Oracle 11g
PostgreSql
...

Solution: Multi-tier Architecture



Multi-tier Backend



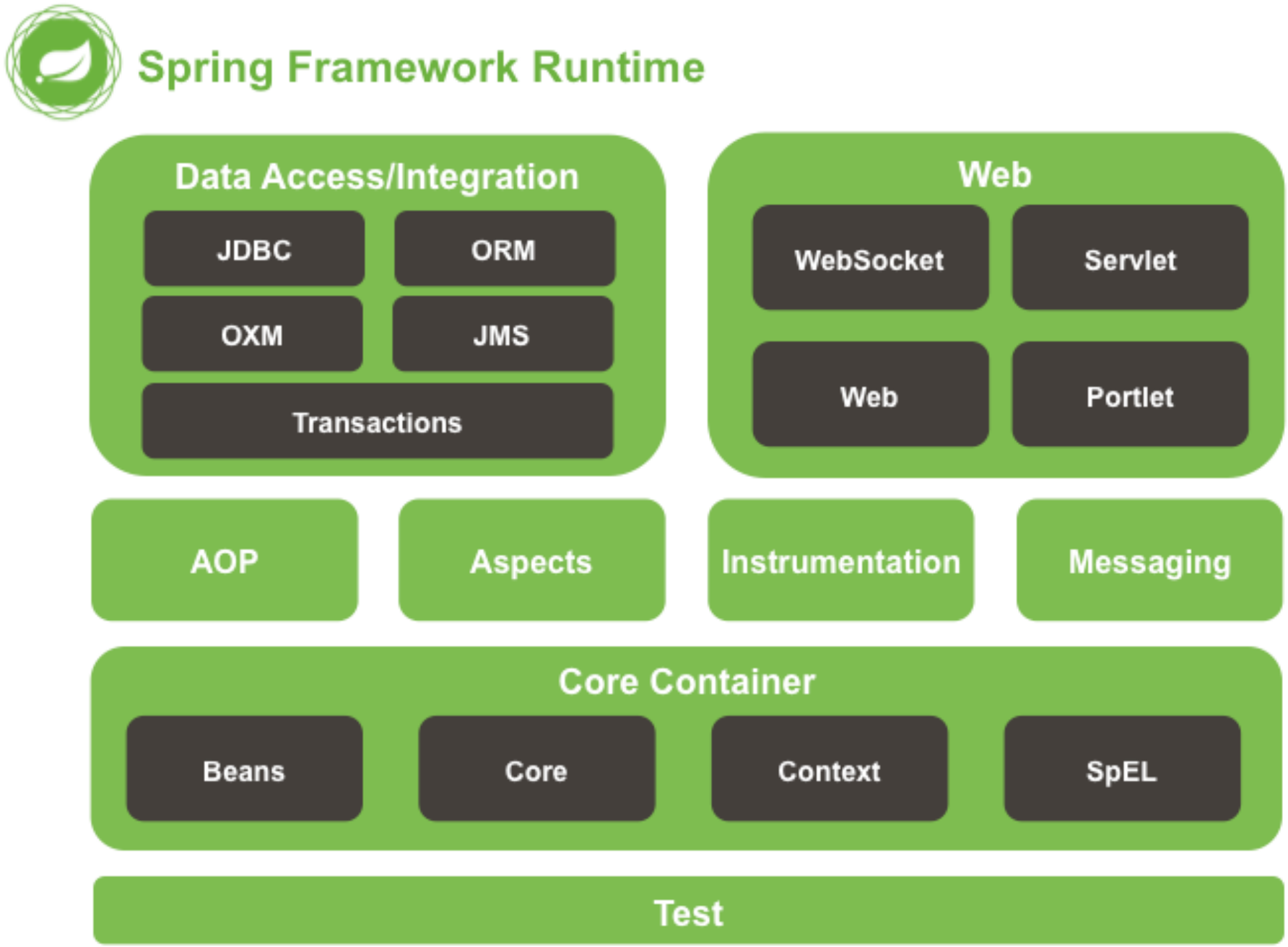
What is Spring Framework

- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- Spring handles the infrastructure so you can focus on your application.
- POJO’s are used both to build applications and enterprise services

Advantages Using Spring

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

Spring Modules



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules whose detail is as follows:

- The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.

- The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The **ApplicationContext** interface is the focal point of the Context module.

- The **Expression Language** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows:

- The **JDBC module** provides a JDBC-abstraction layer that removes the need to do tedious JDBC related coding.

- The **ORM module** provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.

- The **OXM module** provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

- The **Java Messaging Service** JMS module contains features for producing and consuming messages.

- The **Transaction module** supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web

The Web layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules whose detail is as follows:

- The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The Web-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications.
- The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application.
- The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules whose detail is as follows:

- The AOP module provides aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.

- The Aspects module provides integration with AspectJ which is again a powerful and mature aspect oriented programming (AOP) framework.

- The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.

- The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

Environment Setup

- In a standard Java environment, major Spring framework jar and dependent jars must be included in the project (current release 5.3.3)
- In order to get started quickly as possible following tools will be used:
  - Spring Tool Suite 4 (An Eclipse Version)
  - Maven (For dependency management and build)
  - Spring Boot (For testing and deploying)

Spring Boot

- Spring Boot is basically an extension of the Spring framework, which eliminates the boilerplate configurations required for setting up a Spring application. It provides:

- Opinionated ‘starter' dependencies to simplify the build and application configuration

- Embedded server to avoid complexity in application deployment

- Metrics, Health check, and externalized configuration

- Automatic config for Spring functionality – whenever possible

Setup Spring Boot

- Spring Initializer: https://start.spring.io/ , then import into STS as Maven Project.
- Or use STS New Spring Starter Project

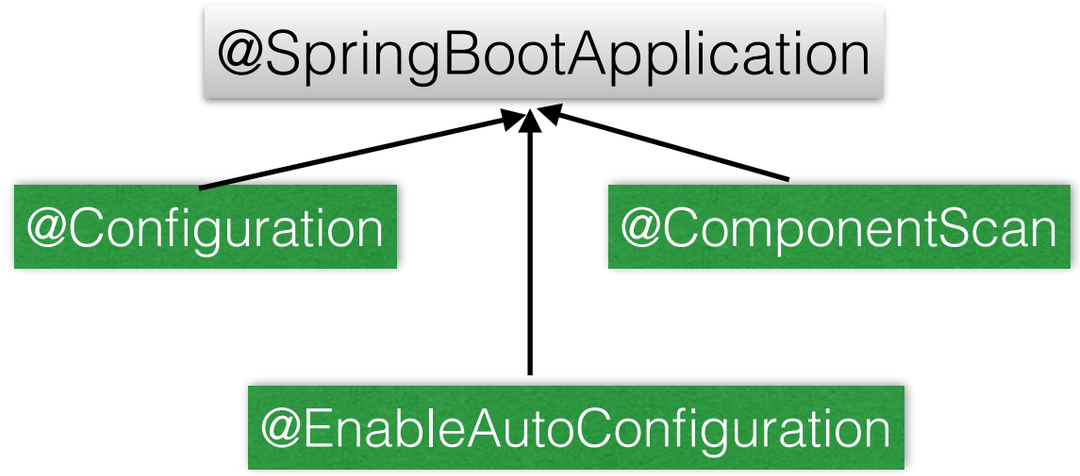
Maven Dependencies

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.2</version>
    <relativePath />
</parent>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Application Configuration

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {

SpringApplication.run(Application.class,
args);
    }
}
```



Spring Configuration Metadata

- Methods to provide configuration metadata to the Spring Container:

1. XML based configuration file.
2. Annotation-based configuration
3. Java-based configuration

Java and Annotation Based Configuration

- Java based configuration option enables you to write most of your Spring configuration **without XML** but with the help of few Java-based annotations

- Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.

- The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

Java and Annotation Based Configuration

In Spring Framework:

```
@ComponentScan(basePackages = {"package1,package2"})
@Configuration
@EnableAutoConfiguration
public class ASpringApp {

    public static void main(String[] args) {
        ApplicationContext ctx =
                SpringApplication.run(ASpringApp.class, args);
        Book b = ctx.getBean(Book.class);
        System.out.println(b.getTitle());
    }
    @Bean
    @Scope("prototype")
    public Book book() {
        return new Book("Grapes of Wrath");
    }
}
```

@Configuration

Defines the current class as a Bean declaration class

@ComponentScan

Enables Spring to scan for things like configurations, controllers, services, and other components we define.

@EnableAutoConfiguration

Enables Spring to auto-configure the application context. Therefore, it automatically creates and registers beans based on both the included jar files in the classpath and the beans defined by us.

In Spring Boot:

```
@SpringBootApplication
public class ASpringApp {

    //Rest is the same as above
}
```

@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiguration

Spring Bean Scopes

- Do you need a new instance of a bean for each call?

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

Singleton Scope

- If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition.

- This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

- It is the default scope

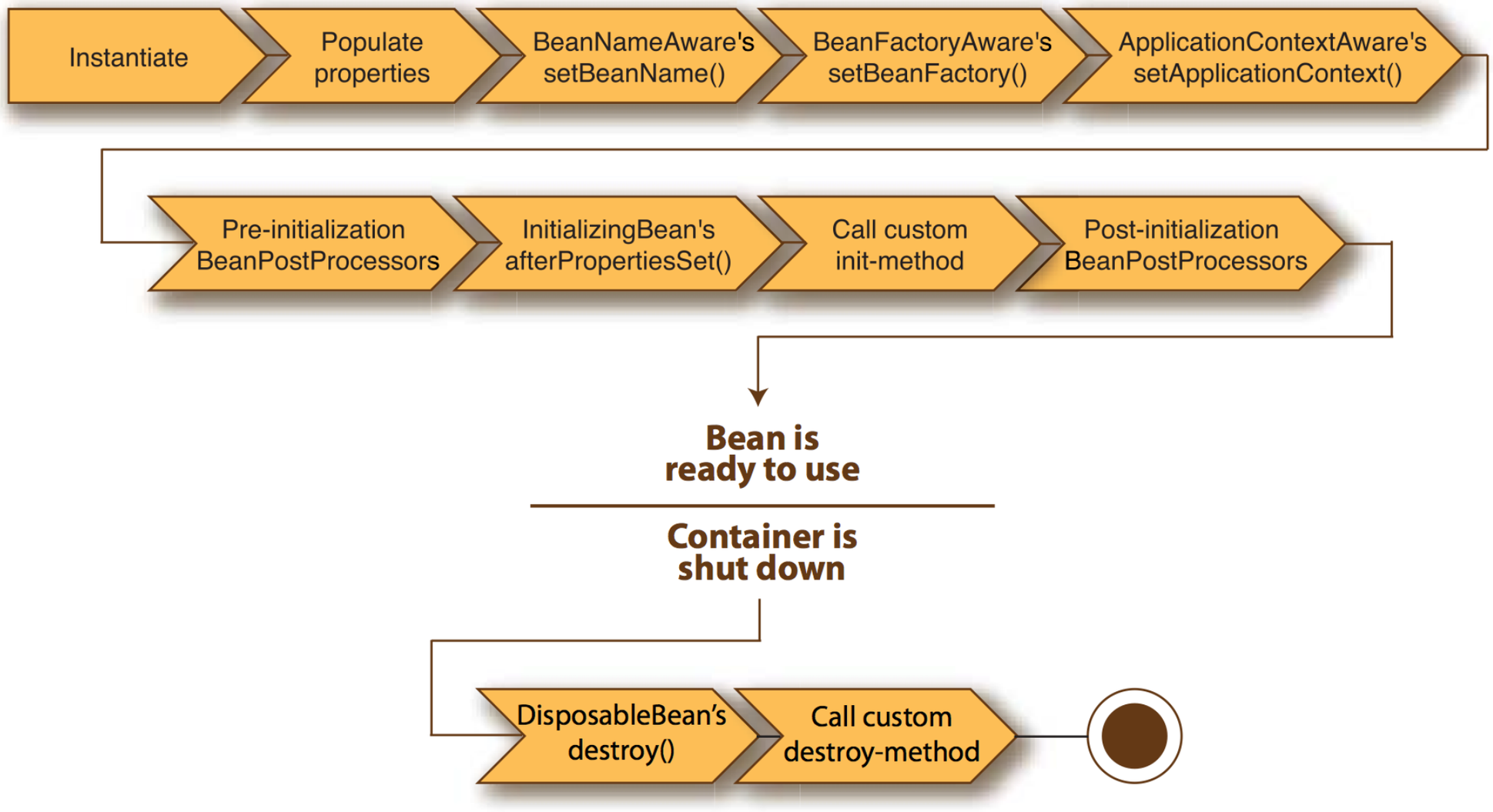
Prototype Scope

- If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made.

- As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

Spring Beans Lifecycle

- A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring



Bean Lifecycle

- When a bean is instantiated, it may be required to perform some initialization to get it into a usable state.
- When the bean is no longer required and is removed from the container, some cleanup may be required
- How to control lifecycle:
  - Implement **InitializingBean** and/or **DisposableBean** interfaces
  - Use JSR-250 **@PostConstruct** and/or **@PreDestroy** methods

```
public class ExampleBean {
     @PostConstruct
    public void init() {
       // do some initialization work
    }
}
```

```
public class ExampleBean {
     @PreDestroy
    public void end() {
       // do some pre destroy tasks
    }
}
```

Dependency Injection (DI) and Inversion control (IOC)

- Objects in an application are dependent on each other (composition, aggregation, etc)

- Lifecycle of objects must be considered in relation with their dependencies

- **The Spring Framework Inversion of Control (IoC)** component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use.

What are DI and IOC?

- Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. Advantages:
    - decoupling the execution of a task from its implementation
    - making it easier to switch between different implementations
    - greater modularity of a program
    - greater ease in testing
- We can achieve Inversion of Control through Dependency Injection

Dependency Injection

Traditional Programming

```
public class Store {
    private Item item;

    public Store() {
        item = new ItemImpl1();
    }
}
```

- Dependency is tightly coupled
- Store owns control of item instance

Dependency Injection

```
public class Store {
    private Item item;
    public Store(Item item) {
        this.item = item;
    }
}
```

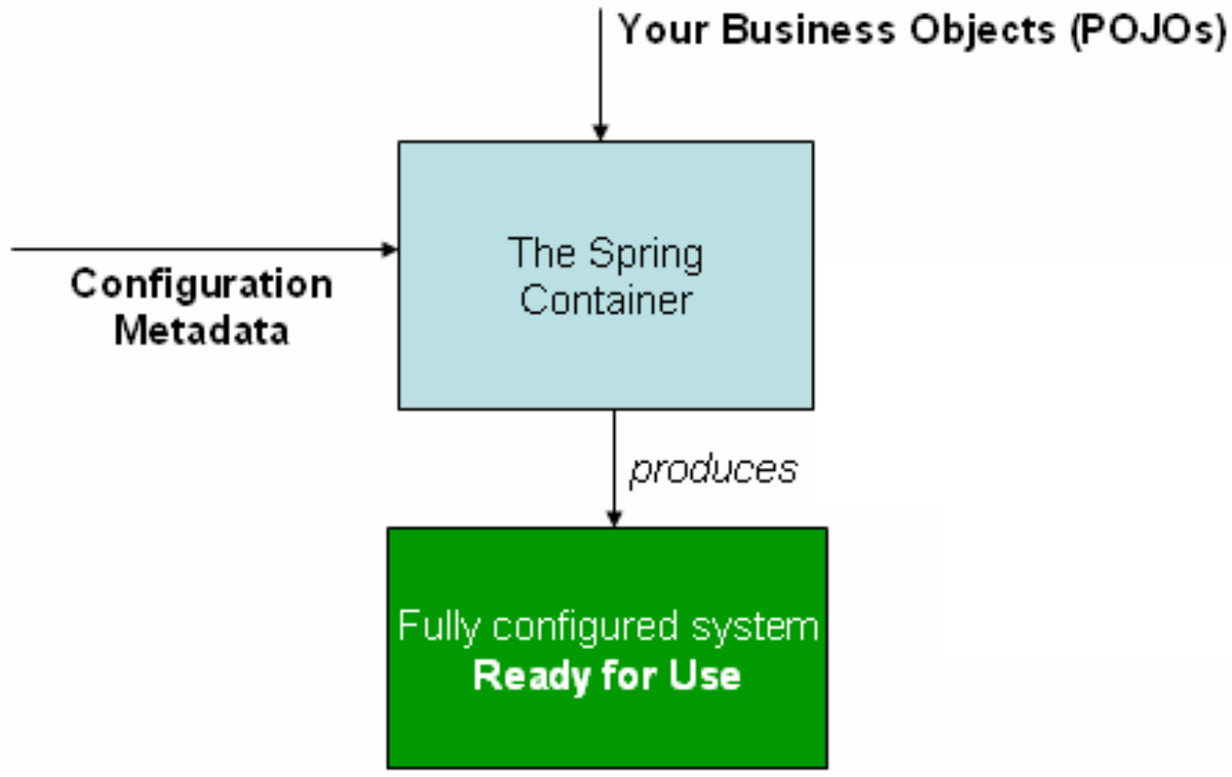
- We can rewrite the example without specifying the implementation of the Item that we want

The IOC Container and Beans

- **IOC** (Inversion of Control) provides dependency injection
  - Manage objects, others that work with those objects, constructor arguments and properties
- The container then **injects** those dependencies when it creates the bean.
- The interface ApplicationContext represents the IoC container.
- In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called ***beans***

Container Overview

- The Spring container is at the core of the Spring Framework.
    - Creates objects, wire them, configure them, manage complete lifecycle
- Container gets instructions by reading configuration meta-data
- Configuration can be represented by **XML**, **Java Annotations** or **Java Code**



Spring ApplicationContext Container

- This container adds more enterprise-specific functionality such as the **ability to resolve textual messages** from a properties file and the ability to **publish application events to interested event listeners.**

- This container is defined by the **org.springframework.context.ApplicationContext** interface.

- Recommended over BeanFactory

- Subtype of BeanFactory

- Common Implementations:

  - **FileSystemXMLApplicationContext**

  - **ClassPathXmlApplicationContext**

  - **WebApplicationContext**

  - **AnnotationConfigApplicationContext**

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

      ApplicationContext context = new
FileSystemXmlApplicationContext
            ("C:/Users/ZARA/workspace/
HelloSpring/src/Beans.xml");

      HelloWorld obj = (HelloWorld)
context.getBean("helloWorld");
      obj.getMessage();
   }
}
```

Constructor Based Dependency Injection

- Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

```
public class Computer {
   private Processor processor;
   public Computer(Processor processor) {
      this.processor = processor;
   }
    public void process(){
        processor.processData();
    }
}
```

The order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor

Setter Based Dependency Injection

- Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

```
public class Computer {
   private Processor processor;

    public void process(){
        processor.processData();
    }

    public Processor getProcessor(){
        return this.processor;
    }

    public void setProcessor(Processor p){
        this.processor = p;
    }
}
```

Field Based DI

- In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation.

- While constructing the Store object, if there's no constructor or setter method to inject the Item bean, the container will use reflection to inject Item into Store.

```
public class Store {
    @Autowired
    private Item item;
}
```

@Autowired

- The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished.
- The @Autowired annotation can be used to autowire bean on the setter method, constructor, a property or methods with arbitrary names and/or multiple arguments.
- Beans still must be defined in configuration file.

Can be over constructor, field or setter

```
class Student{
    @AutoWired
    private Address address;

    public Address getAddress(){
        return this.address;
    }
    public void setAddress(Address address){
        this.address=address;
    }
}
```

Autowiring Modes

Mode	Description
no	This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter.
byName	Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
byType	Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown.
constructor	Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

@Qualifier

- There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

```
public class Profile {
    @Autowired
    @Qualifier("student1")
    private Student student;

     //more code

}
```

Managed Components

- Candidate components can be implicitly detected by scanning the classpath
- Instead of declaring beans in XML format certain annotations can be used
- All beans can be represented with **@Component** stereotype
- **@Service**
  - Implements @Component, for service level interfaces
- **@Repository**
  - Implements @Component, for data access (DAO) operations
- **@Controller**
  - Implements @Component, for creating view level controllers

Examples for Managed Components

```
@Service
@Scope(“singleton”)
public class SimpleMovieLister {
private MovieFinder movieFinder;
@Autowired
public SimpleMovieLister(MovieFinder movieFinder)
{ this.movieFinder = movieFinder;
} }
```

```
@Repository
public class JpaMovieFinder implements MovieFinder { //
implementation elided for clarity
}
```

More Examples for XML'less Configuration

```
@Configuration
@Import(ConfigA.class)
public class ConfigB {
    @Bean
    public B a() {
        return new A();
    }
}
```

@Import annotation imports other configuration classes

```
public class Foo {
    public void init() {// initialization logic}
    public void cleanup() {// destruction logic}
}

@Configuration
public class AppConfig {
    @Bean(initMethod = "init", destroyMethod = "cleanup" )
    public Foo foo() {
        return new Foo();
    }
}
```

Lifecycle callback attributes

```
@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public Foo foo() {
        return new Foo();
    }
}
```

Specifying bean scope

Spring Expression Language

- Expressions are evaluated or executed at bean creation time.
- Can be used with setters, constructors, fields
- Methods can be called using EL

```
public class Customer {
    private Item item;
    private String itemName;
    //getters&setters
}
```

```
public class Item {
    private String name;
    private int quantity;
    //getters&setters
}
```

Expression Language with Annotations

- @Value annotation is used (JSR 330)
- @Value(“[VALUE]”) automatically sets a String value
- @Value(“#{EXPRESSİON}”) evaluates expression outcome

```
@Component("customerBean")
public class Customer {

    @Value("#{itemBean}")
    private Item item;

    @Value("#{itemBean.name}")
    private String itemName;

    @Value(“Some Text Value”)
    private String text;

    //getters&setters

}
```

Referring to Beans

- In Spring EL, you can reference a bean, and nested properties using a ‘dot (.)‘ symbol. For example, “bean.property_name“.

```
@Component("customerBean")
public class Customer {

    @Value("#{addressBean}")
    private Address address;

    @Value("#{addressBean.country}")
    private String country;

    @Value("#{addressBean.getFullAddress('henry')}")
    private String fullAddress;

    //getter and setter methods

    @Override
    public String toString() {
        return "Customer [address=" + address + "\n, country=" + country
                + "\n, fullAddress=" + fullAddress + "]";
    }

}
```

EL Method Invocation

- Spring expression language (SpEL) allow developer uses expression to execute method and inject the method returned value into property, or so called “SpEL method invocation“.

```
@Component("customerBean")
public class Customer {

    @Value("#{'sometext'.toUpperCase()}")
    private String name;

    @Value("#{priceBean.getSpecialPrice()}")
    private double amount;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    @Override
    public String toString() {
        return "Customer [name=" + name + ", amount=" + amount + "]";
    }

}
```

EL Operators

- Spring EL supports most of the standard mathematical, logical or relational operators. For example,
  - **Relational operators** – equal (**==, eq**), not equal (**!=, ne**), less than (**<, lt**), less than or equal (**<= , le**), greater than (**>, gt**), and greater than or equal (**>=, ge**).
  - **Logical operators** – **and**, **or**, and **not** (**!**).
  - **Mathematical operators** – addition (**+**), Subtraction (**-**), Multiplication (*****), division (**/**), modulus (**%**) and exponential power (**^**).

EL Ternary Operator (if-then-else)

```
condition ? true : false
```

```
@Value("#{itemBean.qtyOnHand < 100 ? true : false}")
   private boolean warning;
```