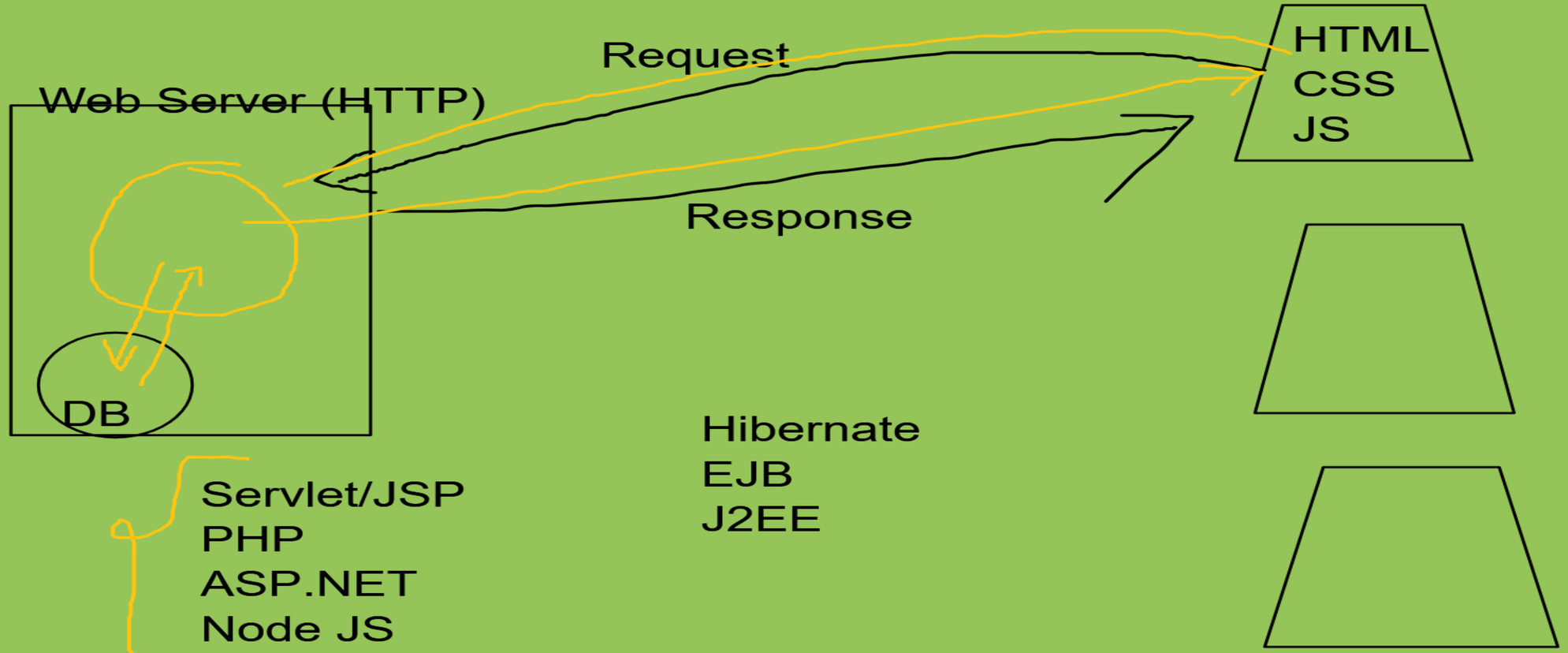
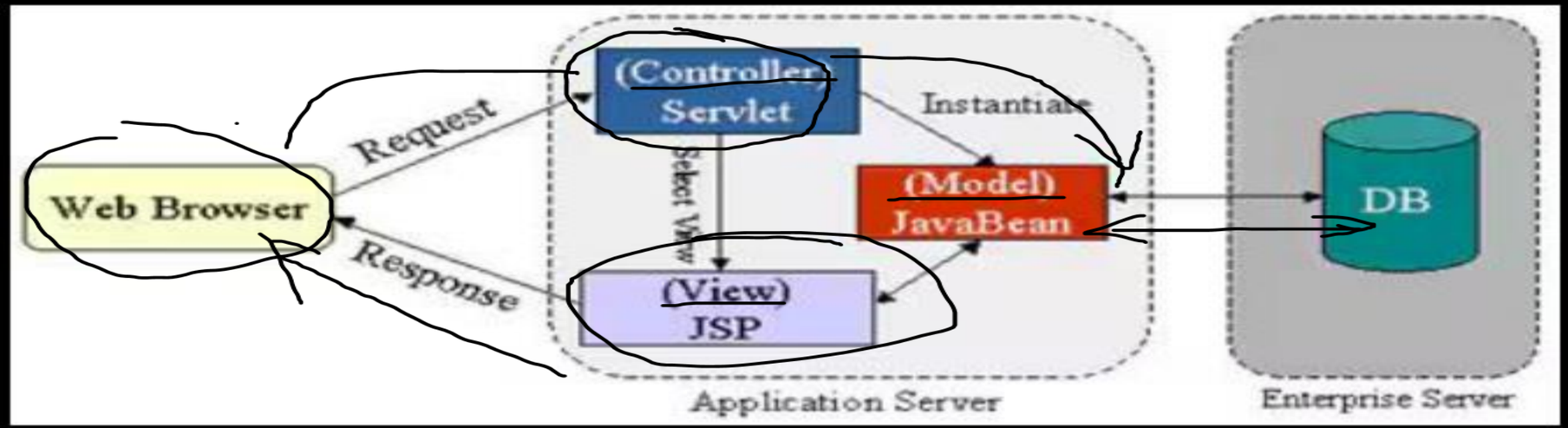
What we will Discuss Today?

- What is Client Server Programming?
- What is Spring Framework?
- Why we use Spring?
- Features of Spring?
- Architecture of Spring Framework


Request
Web Server (HTTP)
HTML
CSS
JS
Response
DB
Hibernate
EJB
J2EE
Servlet/JSP
PHP
ASP.NET
Node JS
etc

Client/Server Programming



What is Spring Framework

From Hibernate to Enterprise Beans, concept is ONLY ONE, why we interact with database, we are java programmers, we knows about classes and Objects not the database or tables or SQL. We knows how to create the Objects or destroy the Objects, we also knows the methods to update the properties of some object, we are not interested in DB.

Forgot the DB, Tables, and SQL , Hibernate or EJB will take care of it. Just focus on Java Objects, but Hibernate and EJB have some complexity problems , and here comes the Spring Framework.

Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. Spring is the most popular application development framework for enterprise Java. Spring is lightweight when it comes to size and transparency.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model

Features of Spring Framework

Inversion of Control: In Spring Framework Dependency Injection is shifted from the application code to the Framework, known as Inversion of Control. Benefits of IoC are:

- Loose Coupling
- Testability
- Reusability
- Module Design

Dependency Injection: Spring allowing Objects to be injected with their dependencies rather than having to create or manage them explicitly. This promotes easier configuration, better decoupling, and improved testability.

Spring MVC: Model View Controller Architecture simplify the Application Development.

Transaction Management: Very powerful transaction management.

Spring Data: Just use Objects and work with Objects. No requirements of SQL, or DB knowledge.

Spring Security: Powerful Authentication and Authorization framework helps to secure the applications.

Spring Boot: This feature is Optional, but it simplifies the setup and configuration of Spring Applications.

Testing: Spring Applications provides supports for Unit Testing, Integration Testing, etc.

Features of Spring Framework

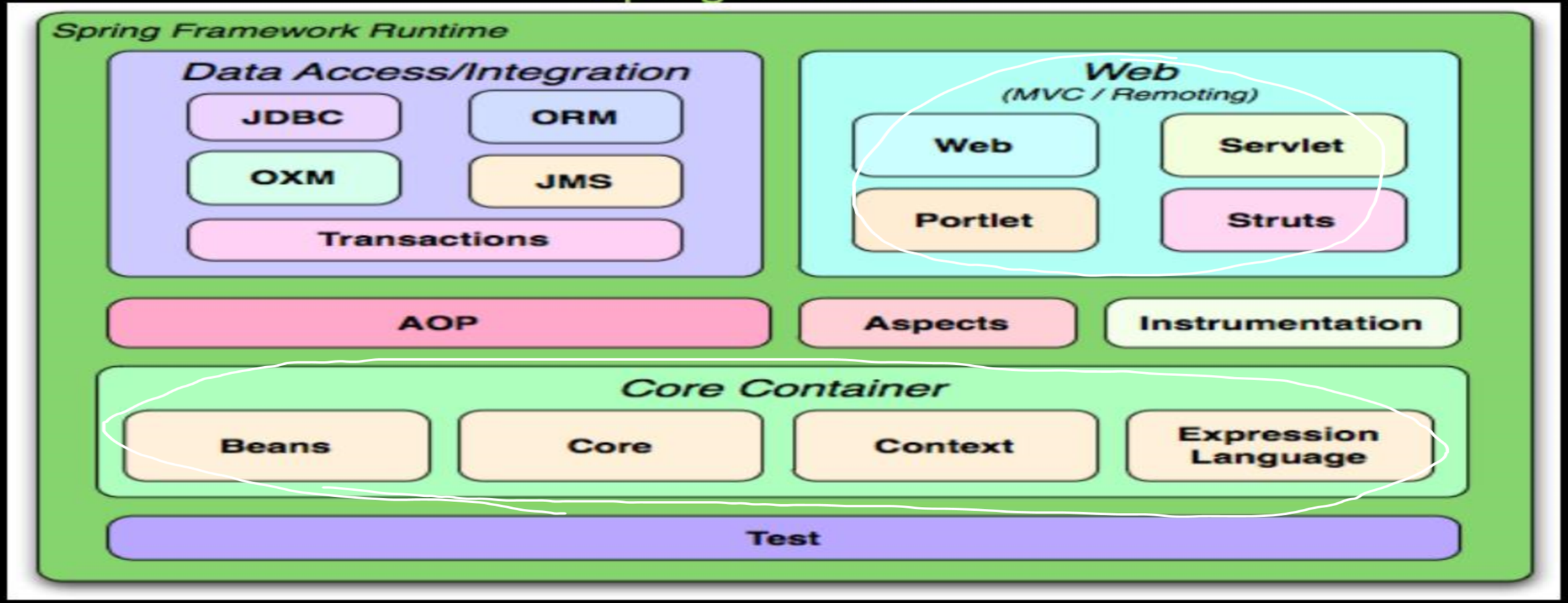
- **Aspect Oriented Programming: One of the key components of Spring is the Aspect Oriented Programming (AOP) framework.**

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect. The AOP module of Spring Framework provides an aspect-oriented programming implementation allowing us to define method-interceptors and point cuts to clearly decouple code that implements functionality that should be separated.

Inversion of Control

- Inversion of Control (IoC) is a principle in software engineering where the control flow of a program is inverted: instead of the programmer controlling the flow of execution, the framework or container controls it. In the context of Spring Boot, IoC is achieved through the Spring IoC container.
- In Spring Boot, IoC is facilitated through Dependency Injection (DI). Dependency Injection is a design pattern where objects are passed their dependencies (i.e., other objects or services) rather than creating them internally.
- The control over the lifecycle and management of these beans is handed over to the Spring framework. The Spring container is responsible for creating, initializing, wiring, and managing the beans throughout their lifecycle.
- By following the IoC principle, Spring Boot promotes loose coupling between components, making the application easier to maintain, test, and extend. It also allows for easier integration with third-party libraries and frameworks, as the control over object instantiation and lifecycle is centralized within the Spring container.

Spring Architecture



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules.

The Core and Beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The Context module builds on the solid base provided by the Core and Beans modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also supports Java EE features such as EJB, JMX ,and basic remoting. 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. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis. Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

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 (plain old Java objects).

Web

The Web layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

Spring's 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. It also contains the web-related parts of Spring's remoting support.

The Web-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The Web-Struts module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

AOP and Instrumentation

Spring's AOP module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes.

The separate Aspects module provides integration with AspectJ.

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

Test

The Test module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

Dependency Injection and Inversion of Control

Java applications -- a loose term that runs the gamut from constrained applets to n-tier server-side enterprise applications -- typically consist of objects that collaborate to form the application proper. Thus the objects in an application have dependencies on each other.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. True, you can use design patterns such as Factory, Abstract Factory, Builder, Decorator, and Service Locator to compose the various classes and object instances that make up an application. However, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that you must implement yourself in your application.

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. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s). Numerous organizations and institutions use the Spring Framework in this manner to engineer robust, maintainable applications.

Scope of Beans

There are 6 different types of bean scopes in the Spring framework.

The scope of a bean defines the life cycle and visibility of that bean in the contexts we use it.

1. singleton
2. prototype
3. request
4. session
5. application
6. websocket

The last four scopes mentioned, request, session, application and websocket, are only available in a web-aware application.

Singleton Scope

When we define a bean with the singleton scope, the container creates a single instance of that bean; all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.

Let's create a Person entity to exemplify the concept of scopes:

```
public class Person {
   private String name;

   // standard constructor, getters and setters
}
```

Afterwards, we define the bean with the singleton scope by using the @Scope annotation:

```
@Bean
@Scope("singleton")
public Person personSingleton() {
   return new Person();
}
```

Prototype Scope

A bean with the prototype scope will return a different instance every time it is requested from the container. It is defined by setting the value prototype to the @Scope annotation in the bean definition:

```
@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

Web aware Scopes

Web Aware Scopes

As previously mentioned, there are four additional scopes that are only available in a web-aware application context.

The request scope creates a bean instance for a single HTTP request, while the session scope creates a bean instance for an HTTP Session.

The application scope creates the bean instance for the lifecycle of a ServletContext, and the websocket scope creates it for a particular WebSocket session.

Thank You

What we will Discuss Today?

- How to Configure Beans in Spring Boot?
- What is Auto Wiring?
- How its Different with Old Methods?
- Benefits of Auto Wiring?
- Life Cycle of Beans
- Callback Life Cycle

Beans Configuration

In Spring, there are several styles for configuring beans, each catering to different preferences and requirements:

1) XML Configuration: This is the traditional approach where beans and their dependencies are configured using XML files. Beans are defined using <bean> tags, and their dependencies are injected using attributes like ref and value. XML configuration provides a clear separation between configuration and code, making it easy to understand and manage.

```
<bean id="myService" class="com.example.MyService">
   <property name="myRepository" ref="myRepository"/>
</bean>
<bean id="myRepository" class="com.example.MyRepository"/>
```

Beans Configuration

2) Annotation-Based Configuration: This style uses annotations such as @Component, @Service, @Repository, and @Autowired to configure beans and their dependencies. Beans are automatically detected and registered by Spring based on these annotations. This approach reduces XML configuration overhead and promotes a more concise and readable codebase.

```
@Service
public class MyService {
    @Autowired
    private MyRepository myRepository;
}
```

Beans Configuration

3) Java Configuration (Programmatic): With Java configuration, beans are configured using Java classes annotated with @Configuration and @Bean. This approach allows for more type safety and refactoring support compared to XML configuration. Beans and their dependencies are defined programmatically within these configuration classes.

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService(myRepository());
    }

    @Bean
    public MyRepository myRepository() {
        return new MyRepository();
    }
}
```

Beans Configuration

4) Java Configuration (Lambda-Based): This is a more concise version of Java configuration introduced in Spring 4, leveraging lambda expressions to define beans. It provides a more functional programming style and reduces boilerplate code.

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService(MyRepository myRepository) {
        return new MyService(myRepository);
    }

    @Bean
    public MyRepository myRepository() {
        return new MyRepository();
    }
}
```

Beans Configuration

Each bean configuration style has its advantages and is suitable for different scenarios. XML configuration offers flexibility and clarity, annotation-based configuration reduces boilerplate code, Java configuration provides type safety and refactoring support, lambda-based configuration offers a more concise syntax.

What is Auto wiring?

We had discussed Old Traditional Methods of Beans Configuration in Spring Boot.

Auto-wiring helps in reducing manual code and makes your application more maintainable and flexible by decoupling components and managing their dependencies automatically.

In Spring Boot, auto-wiring is a feature provided by the Spring framework for automatically injecting dependencies into beans. This feature eliminates the need for explicit bean wiring in configuration files by allowing Spring to automatically wire up the dependencies based on their types.

How its works?

1) **Dependency Injection**: Spring Boot uses dependency injection to manage the dependencies between beans. Instead of manually creating and managing instances of objects, you define the dependencies between them, and Spring handles the instantiation and injection of those dependencies.

Auto wiring

2) **Auto-wiring**: When you annotate a field, setter method, or constructor with @Autowired in a Spring-managed component (such as a controller, service, or repository), Spring will automatically search for a bean of the corresponding type in its application context and inject it into the annotated field or method parameter.

```
@Service
public class MyService
{
    private final MyRepository repository;

    @Autowired
    public MyService(MyRepository repository) {
        this.repository = repository;
    }

    // Service methods that use repository...
}
```

In this example, the MyService class has a dependency on MyRepository, and it's being injected via constructor auto-wiring.

Auto wiring

3) Component Scanning: Spring Boot automatically scans the packages for classes annotated with @Component, @Service, @Repository, etc., and registers them as beans in the application context. This allows Spring to know about the beans and their dependencies, making auto-wiring possible.

Bean Life Cycle

The lifecycle of a bean in Java Spring consists of several phases:

- **Instantiation**: At this stage, the Spring container creates an instance of the bean. Depending on the bean scope, this could happen at different times, such as during application startup for singleton beans or upon request for prototype beans.
- **Population of Properties**: After instantiation, the container sets the bean's properties and dependencies, either through constructor injection, setter injection, or field injection.
- **Initialization Callbacks**: Before the bean is ready for use, any initialization callbacks are invoked.
- **Bean Ready for Use**: At this point, the bean is fully initialized and ready for use by other beans or components in the application.
- **Usage**: The bean can now be used throughout the application context, either directly or through dependency injection.
- **Destruction Callbacks**: When the application context is shut down or when the bean is explicitly destroyed, any destruction callbacks are invoked.
- **Bean Destruction**: After destruction callbacks are invoked, the container releases the resources held by the bean and removes it from the application context.

Callback Life Cycle

Lifecycle callbacks in Spring allow you to perform actions during different phases of a bean's lifecycle, such as initialization and destruction. There are several ways to define lifecycle

1) **InitializingBean and DisposableBean Interfaces**: Beans can implement InitializingBean and DisposableBean interfaces, which require them to implement afterPropertiesSet() and destroy() methods, respectively. These methods are invoked by the Spring container after the bean has been instantiated and before it is destroyed.

```
public class MyBean implements InitializingBean, DisposableBean {
    @Override
    public void afterPropertiesSet() throws Exception {
        // Initialization logic
    }

    @Override
    public void destroy() throws Exception {
        // Destruction logic
    }
}
```

Callback Life Cycle

2) @PostConstruct and @PreDestroy Annotations: You can use the @PostConstruct and @PreDestroy annotations to define initialization and destruction methods directly in your bean class. These methods will be invoked by the Spring container at the appropriate lifecycle phases.

```
public class MyBean {
    @PostConstruct
    public void init() {
        // Initialization logic
    }

    @PreDestroy
    public void cleanup() {
        // Destruction logic
    }
}
```

Callback Life Cycle

3) Custom Initialization and Destruction Methods: You can define custom initialization and destruction methods in your bean class and configure them in the Spring XML configuration or through Java-based configuration.

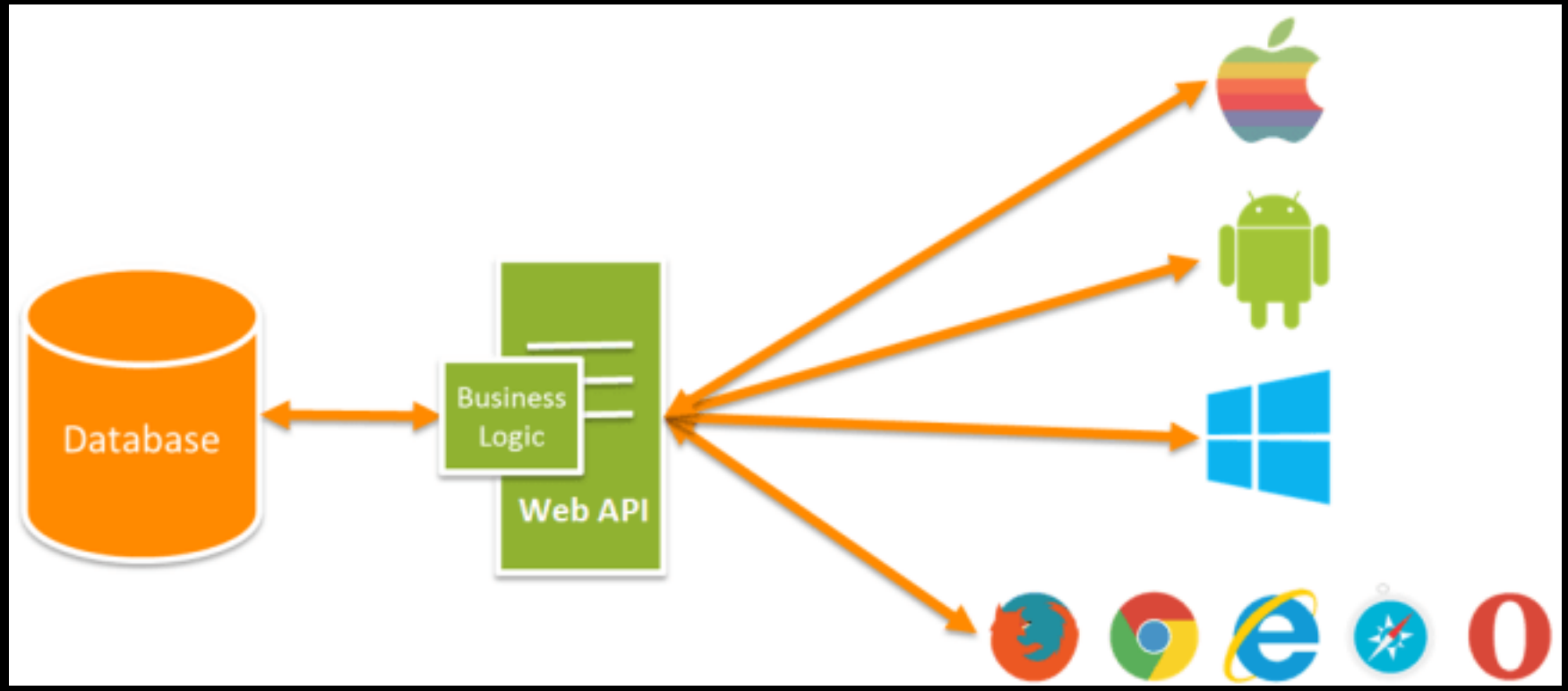
```
public class MyBean {
    public void customInitMethod() {
        // Initialization logic
    }

    public void customDestroyMethod() {
        // Destruction logic
    }
}
```

Thank You

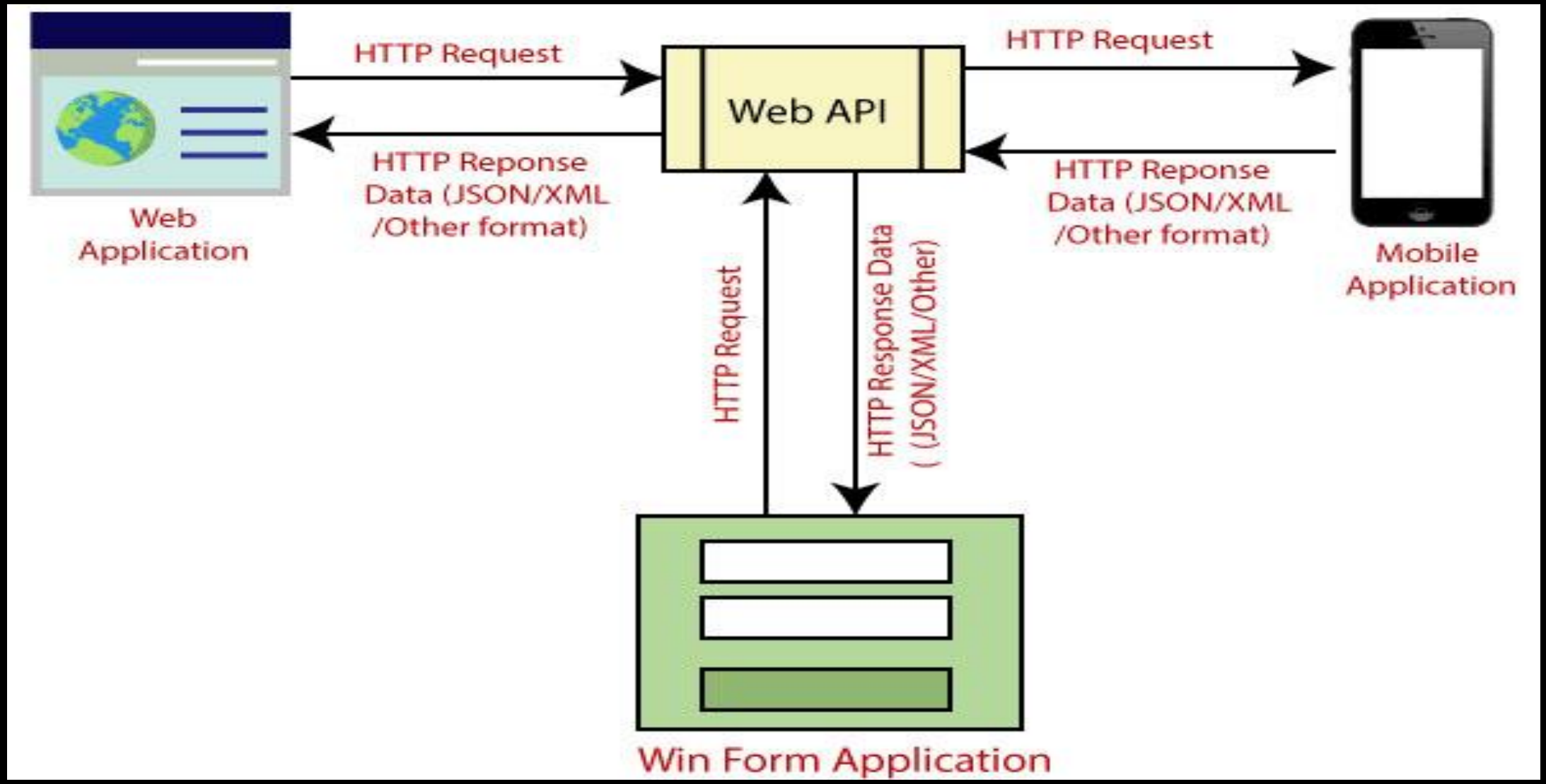
What we will Discuss Today?

- What is REST API?
- Why we use REST API?
- GET, POST, PUT, DELETE APIs?
- How to build Web Applications in Spring boot?
- Types of Spring Boot Build Systems.
- Structure of Spring Boot Code.
- Spring Boot Runners
- Rest Controller, Request Mapping, Request Body, Path Variable, Request Parameter

REST API



REST API


HTTP Request
Web API
HTTP Request
HTTP Reponse
Data (JSON/XML
/Other format)
HTTP Reponse
Data (JSON/XML
/Other format)
Web
Application
Mobile
Application
HTTP Request
HTTP Response Data
((JSON/XML/Other)
Win Form Application

REST API

REST (Representational State Transfer) API is an architectural style for designing networked applications. It relies on a stateless communication protocol, usually HTTP, and standard operations (GET, POST, PUT, DELETE) to manipulate resources (data or objects) on a server. RESTful APIs are characterized by their simplicity, scalability, and the ability to leverage existing protocols and infrastructure.

Here are some key characteristics of RESTful APIs:

- **Statelessness**: Each request from a client to the server must contain all the necessary information to understand and fulfill the request. The server should not store any client context between requests.
- **Resource-based**: RESTful APIs are centered around resources, which can be any entity that can be uniquely identified. Resources are manipulated using standard HTTP methods.
- **Uniform Interface**: REST APIs use a uniform interface to access resources. This typically involves using standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources. Additionally, resources are identified by URIs (Uniform Resource Identifiers).

REST API

Client-Server Architecture: The client and server are separate concerns, and the client is not concerned with the data storage or business logic on the server. This allows for separation of concerns and improved scalability.

Cacheability: Responses from the server can be labeled as cacheable or non-cacheable. This allows clients to cache responses to improve performance.

Layered System: REST APIs can be layered, allowing for intermediaries such as proxies or gateways to be inserted between clients and servers to improve scalability, security, or other concerns.

RESTful APIs have become the standard for building web services due to their simplicity, flexibility, and compatibility with existing web technologies. They are commonly used for building APIs for web applications, mobile applications, and IoT devices, among other use cases.

GET/ POST/ PUT/ DELETE API

GET: The GET method is used to request data from a specified resource. It's a safe and idempotent operation, meaning it doesn't modify the resource on the server, and making multiple identical requests should have the same effect as making a single request.

GET requests are typically used for retrieving data, such as fetching a webpage, getting user details, or fetching a list of items from a database.

Example: GET /users retrieves a list of all users.

POST: The POST method is used to submit data to be processed to a specified resource.

It's not idempotent, meaning making multiple identical requests may have different effects each time.

POST requests are commonly used for creating new resources, such as submitting a form, uploading a file, or creating a new record in a database.

Example: POST /users creates a new user with the provided data in the request body.

GET/ POST/ PUT/ DELETE API

PUT: PUT method is used to update or replace an existing resource with new data at a specified URI. It's idempotent, meaning making multiple identical requests should have the same effect as making a single request. PUT requests are commonly used for updating existing resources, such as modifying a user's details or updating a record in a database. When using PUT, the client typically sends the entire updated resource in the request body, not just the fields that have changed.

Example: PUT /users/123 updates the user with ID 123 with the provided data in the request body.

DELETE: The DELETE method is used to request the removal of a specified resource from the server. It's idempotent, meaning making multiple identical requests should have the same effect as making a single request. DELETE requests are commonly used for deleting resources, such as removing a user account, deleting a file, or deleting a record from a database.

Example: DELETE /users/123 deletes the user with ID 123 from the system.

Some Important Terms Related to the WEB API

1. Entity:

- - An Entity represents a table in a relational database. It typically maps to a database table, with each instance of the Entity representing a row in that table.
- - In Spring Boot applications, Entities are usually annotated with `@Entity` from the JPA (Java Persistence API) specification. This annotation tells Spring Boot that the class is a JPA entity, and it should be mapped to a corresponding database table.
- - Entities contain fields that represent the columns in the corresponding database table, along with getter and setter methods to access and modify these fields.
- - Example: In a simple, you might have an `Products` entity with fields like `id`, `name`, `price`, and `qty`.

Some Important Terms Related to the WEB API

2. Repository:

- - A Repository is responsible for managing data access logic, such as querying, saving, updating, and deleting data in a database.
- - In Spring Boot applications, Repositories are typically interfaces that extend `JpaRepository` or a similar interface provided by Spring Data JPA. These interfaces provide methods for common CRUD operations without requiring explicit implementation.
- - Repositories allow developers to interact with the database using object-oriented concepts rather than writing native SQL queries.
- - Example: For the `Products` entity mentioned earlier, you might have an `ProductsRepository` interface that allowing you to perform operations like `save`, `findById`, `findAll`, etc., on `Products` entities.

Some Important Terms Related to the WEB API

3. Service:

- - A Service contains the business logic of an application. It co-ordinates the interactions between multiple components, such as Entities and Repositories, to perform specific operations or fulfill business requirements.
- - Services encapsulate the application's logic into reusable and testable components, promoting modular and maintainable code.
- - In Spring Boot applications, Services are typically annotated with `@Service`. This annotation tells Spring that the class is a service component and should be managed by the Spring container.
- - Example: In an application, you might have an `ProductService` that contains methods for creating, retrieving, updating, and deleting products. The `ProductService` would use methods from the `ProductRepository` to interact with the database.

Creating a NEW Web Application for REST API-1

What we need:

1) My Sql Database (We will use My Sql with XAMPP)
2) VS Code
3) JDK 17 or higher
4) POSTMAN tool to check the APIs
5) Knowledge of Basic SQL Commands.

Before continue , please install these software on your computer and create a table with following fields:

Products (id,product,details,price,qty) in a database let “DB1”

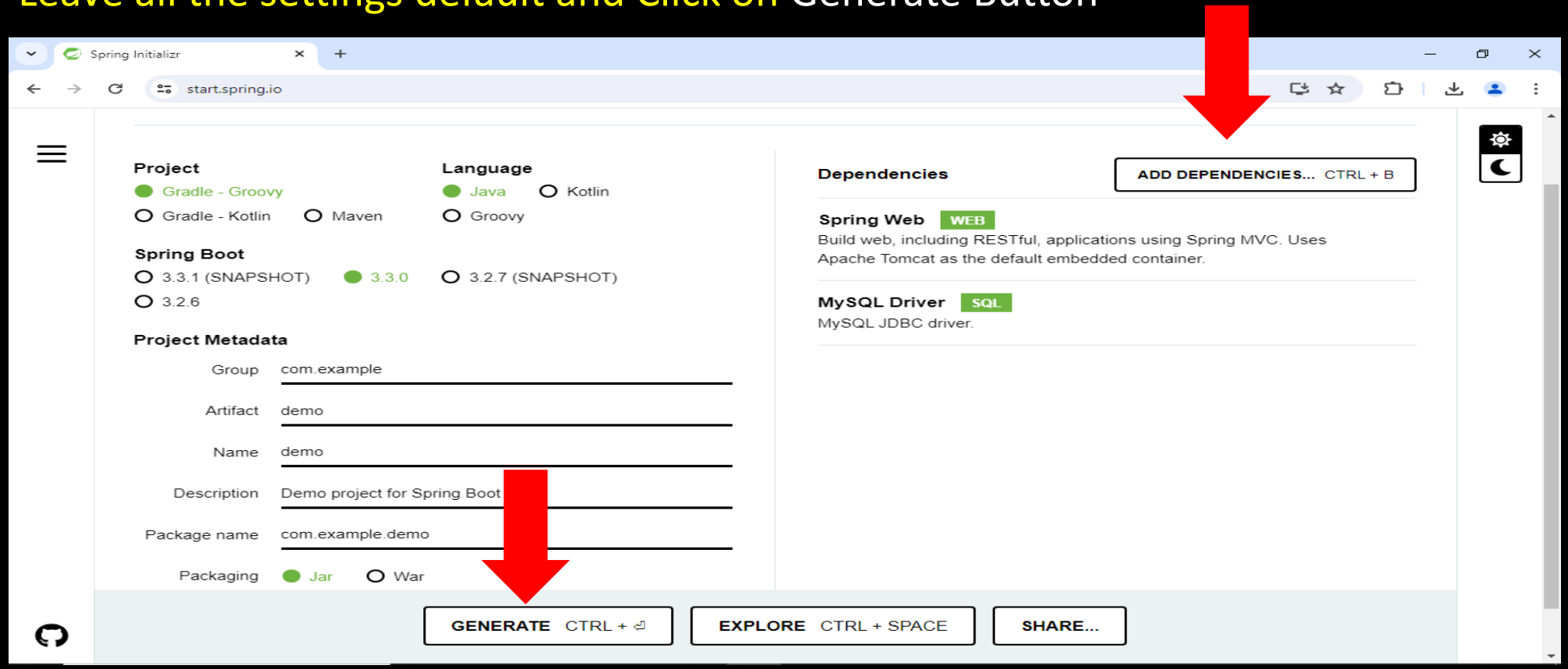
```
Create Table Products
(
        id int primary key,
        product varchar(30),
        details varchar(100),
        price int,
        qty int
);
```

Creating a NEW Web Application for REST API-2

Open following website : https://start.spring.io

Add Following Dependencies:

1) Spring WEB
2) My SQL Driver
3) Leave all the settings default and Click on Generate Button



Creating a NEW Web Application for REST API-3

Download the Generated zip file and extract it into some folder, and then open this folder in VS Code.

And do the following…….. Lets do it

Creating a NEW Web Application for REST API

Hope that now you have clearly understand the concept of Services, Repository, and Entities along with the concept of REST API. Now we will discuss following topics:

- Types of Spring Boot Build Systems.
- Structure of Spring Boot Code.
- Spring Boot Runners

Spring Boot Build Systems

Spring Boot primarily supports two build systems: Maven and Gradle. Both are powerful tools for managing dependencies, building, testing, and packaging Java applications. Here are the details:

1. Maven:

Maven is a widely-used build automation tool primarily used for Java projects. It relies on XML for its configuration, which can be a bit verbose but is also highly configurable. Here's how Maven works with Spring Boot:

- - pom.xml: In Maven, project configuration is stored in a file called `pom.xml` (Project Object Model). This file includes information about the project and its dependencies.
- - Dependencies: Dependencies are managed in the `pom.xml` file. Spring Boot dependencies can be easily included by adding them to the dependencies section with appropriate version numbers.
- - Plugins: Maven plugins are used to execute specific goals, such as compiling code, running tests, and packaging applications. Spring Boot Maven plugin is particularly useful for packaging Spring Boot applications into executable JAR or WAR files.
- - Build Lifecycle: Maven has a predefined set of lifecycle phases (e.g., clean, compile, test, package) that dictate the order in which goals are executed. These phases make it easy to perform common tasks during the build process.

Spring Boot Build Systems

2. Gradle:

Gradle is another popular build automation tool that offers more flexibility and a Groovy-based DSL (Domain Specific Language) for defining build scripts. Here's how Gradle works with Spring Boot:

- - build.gradle: In Gradle, project configuration is stored in a file called `build.gradle`. This file is typically written in Groovy (though Kotlin DSL is also supported) and defines project structure, dependencies, and tasks.
- - Dependencies: Similar to Maven, Gradle manages dependencies, but it uses a more concise syntax. Dependencies are declared in the `build.gradle` file, and Spring Boot dependencies can be included using Gradle's `implementation` or `compile` configuration.
- - Plugins: Gradle has a rich ecosystem of plugins that provide additional functionality. The Spring Boot Gradle plugin enhances Gradle's capabilities for building and packaging Spring Boot applications.
- - Task-based: Gradle builds are organized around tasks. Each task performs a specific action, such as compiling code, running tests, or creating a distribution. Tasks can be customized and extended as needed.

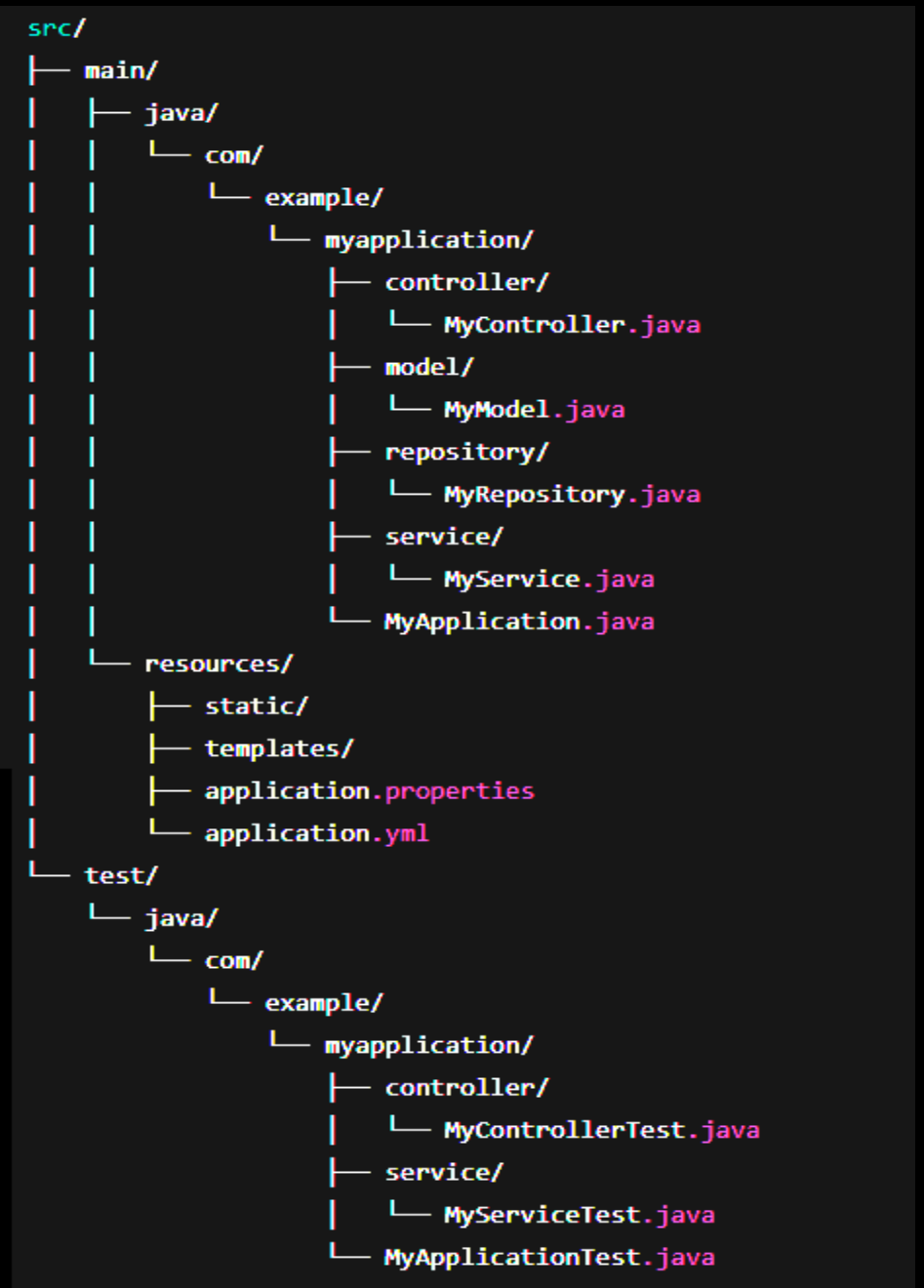
Spring Boot Build Systems

2. Gradle:

Gradle is another popular build automation tool that offers more flexibility and a Groovy-based DSL (Domain Specific Language) for defining build scripts. Here's how Gradle works with Spring Boot:

- - build.gradle: In Gradle, project configuration is stored in a file called `build.gradle`. This file is typically written in Groovy (though Kotlin DSL is also supported) and defines project structure, dependencies, and tasks.
- - Dependencies: Similar to Maven, Gradle manages dependencies, but it uses a more concise syntax. Dependencies are declared in the `build.gradle` file, and Spring Boot dependencies can be included using Gradle's `implementation` or `compile` configuration.
- - Plugins: Gradle has a rich ecosystem of plugins that provide additional functionality. The Spring Boot Gradle plugin enhances Gradle's capabilities for building and packaging Spring Boot applications.
- - Task-based: Gradle builds are organized around tasks. Each task performs a specific action, such as compiling code, running tests, or creating a distribution. Tasks can be customized and extended as needed.

Structure of Spring Boot Code.



Spring Boot applications typically follow a common structure to organize their code.

src/main/java: This directory contains the main Java source code for your application.

src/main/resources: This directory contains non-Java resources used by your application.

src/test/java: This directory contains the unit and integration tests for your application.

Spring Boot Runners

Spring Boot Runners are special types of classes in a Spring Boot application that are executed during the application startup process. These runners allow developers to perform specific tasks or initialize certain components before the application is fully up and running.

In Spring Boot, runners are implemented using the CommandLineRunner and ApplicationRunner interfaces. These interfaces provide a single run method that is called by the Spring Boot framework at the appropriate time during application startup.

1) CommandLineRunner
2) ApplicationRunner

Spring Boot Runners

CommandLineRunner: This interface provides a run method with a String... args parameter. The args parameter allows access to any command-line arguments passed to the application. Developers can implement this interface to execute tasks that require access to command-line arguments, such as initializing components, loading data, or performing other startup tasks.

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
@Component
public class MyCommandLineRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        // Perform startup tasks here
    }
}
```

Spring Boot Runners

2) ApplicationRunner: Similar to CommandLineRunner, this interface provides a run method but with an ApplicationArguments parameter instead of a String... args. The ApplicationArguments object provides access to parsed command-line arguments as well as application properties. Developers can use this interface to perform tasks that require access to both command-line arguments and application properties.

```
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component
public class MyApplicationRunner implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        // Perform startup tasks here
    }
}
```

Thanks
&
Best of Luck

// Products.java

```
package com.example.demo.model;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
//mark class as an Entity
@Entity
//defining class product as Table product
@Table
public class Products
{
   @Id
   @Column
   int id;
   @Column
   String product;
   @Column
   String details;
   @Column
   int price;
   @Column
   int qty;
   public Products() {
   }
   public Products(int id,String product, String details, int price, int qty)
{
      this.id = id;
      this.product = product;
      this.details = details;
      this.price = price;
      this.qty = qty;
   }
   public int getId() {
      return id;
```

```
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getProduct() {
        return product;
    }
    public void setProduct(String product) {
        this.product = product;
    }
    public String getDetails() {
        return details;
    }
    public void setDetails(String details) {
        this.details = details;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
    public int getQty() {
        return qty;
    }
    public void setQty(int qty) {
        this.qty = qty;
    }
}
```

// ProductsRepository.java

```
package com.example.demo.repository;
import org.springframework.data.repository.CrudRepository;
import com.example.demo.model.*;
//repository that extends CrudRepository
public interface ProductsRepository extends
CrudRepository<Products,Integer>
{

}
```

```
// ProductsService.java


package com.example.demo.services;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.demo.repository.*;
import com.example.demo.model.*;
//defining the business logic
@Service
public class ProductsService
{
   @Autowired
   ProductsRepository productsRepository;
   //getting all Products record by using the method findaAll() of
CrudRepository
   public List<Products> getAllProducts()
   {
      List<Products> products = new ArrayList<Products>();
      for(Products b:productsRepository.findAll())
      {
         products.add(b);
```

```
        }
        return products;
    }
    //getting a specific record by using the method findById() of
CrudRepository
    public Products getProductsById(int id)
    {
        return productsRepository.findById(id).get();
    }
    //saving a specific record by using the method save() of
CrudRepository
    public void saveOrUpdate(Products products)
    {
        productsRepository.save(products);
    }
    //deleting a specific record by using the method deleteById() of
CrudRepository
    public void delete(int id)
    {
        productsRepository.deleteById(id);
    }
    //updating a record
    public void update(Products products, int pid)
    {
        productsRepository.save(products);
    }
}
```

// ProductsController.java

```
package com.example.demo;
import java.util.List;
import java.util.ArrayList;
import java.util.Optional;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.services.*;

import com.example.demo.model.Products;
import com.example.demo.model.*;
//mark class as Controller
@RestController
public class ProductsController
{
   //autowire the ProductsService class
   @Autowired
   ProductsService productsService;

   //creating a get mapping that retrieves all the Products detail from the
database
   @GetMapping("/products")
   private List<Products> getAllProducts()
   {
      return productsService.getAllProducts();
   }
   //creating a get mapping that retrieves the detail of a specific book
   @GetMapping("/products/{id}")
   private Products getProducts(@PathVariable("id") int id)
   {
      return productsService.getProductsById(id);
   }
   //creating a delete mapping that deletes a specified book
```

```
    @DeleteMapping("/products/{id}")
    private void deleteProducts(@PathVariable("id") int id)
    {
        productsService.delete(id);
    }
    //creating post mapping that post the book detail in the database
    @PostMapping("/products")
    private String saveProducts(@RequestBody Products products)
    {
        productsService.saveOrUpdate(products);
        int id=products.getId();
        return "OK";
    }
    //creating put mapping that updates the book detail
    @PutMapping("/products")
    private Products update(@RequestBody Products Products)
    {
        productsService.saveOrUpdate(Products);
        return Products;
    }
}
```