Day7: Overview of Enterprise application, Spring Core introduction

Overview of J2EE

The **Java EE** stands for **Java Enterprise Edition**
, which was earlier known as J2EE and is currently known as Jakarta EE. It is a set of specifications wrapping around Java SE (Standard Edition). The Java EE provides a platform for developers with enterprise features such as distributed computing and web services. Java EE applications are usually run on reference run times such as A**pplication servers**. Examples of some contexts where Java EE is used are e-commerce, accounting, banking information systems.

J2EE provides a solution to the problems encountered by companies moving to a multi-tier computing model. The problems addressed include reliability, scalability, security, application deployment, transaction processing, web interface design, and timely software development. It builds upon the Java 2 Platform, Standard Edition (J2SE) to enable Sun Microsystems' "Write Once, Run Anywhere" paradigm for multi-tier computing.

Java EE has several specifications which are useful in making web pages, reading and writing from database in a transactional way, managing distributed queues. The Java EE contains several APIs which have the functionalities of base Java SE APIs such as Enterprise JavaBeans, connectors, Servlets, Java Server Pages and several web service technologies.

What is the J2EE Application Model?

The J2EE application model is a multi-tier application model. Application components are managed in the middle tier by containers. A container is a standard runtime environment that provides services, including life cycle management, deployment, and security services, to application components. This container-based model separates business logic from system infrastructure.

What is an Application Server?

An application server is software that runs between web-based client programs and back-end databases and legacy applications. They help separate system complexity from business logic, enabling developers to focus on solving business problems. They help reduce the size and complexity of client programs by enabling these programs to share capabilities and resources in an organized and efficient way.

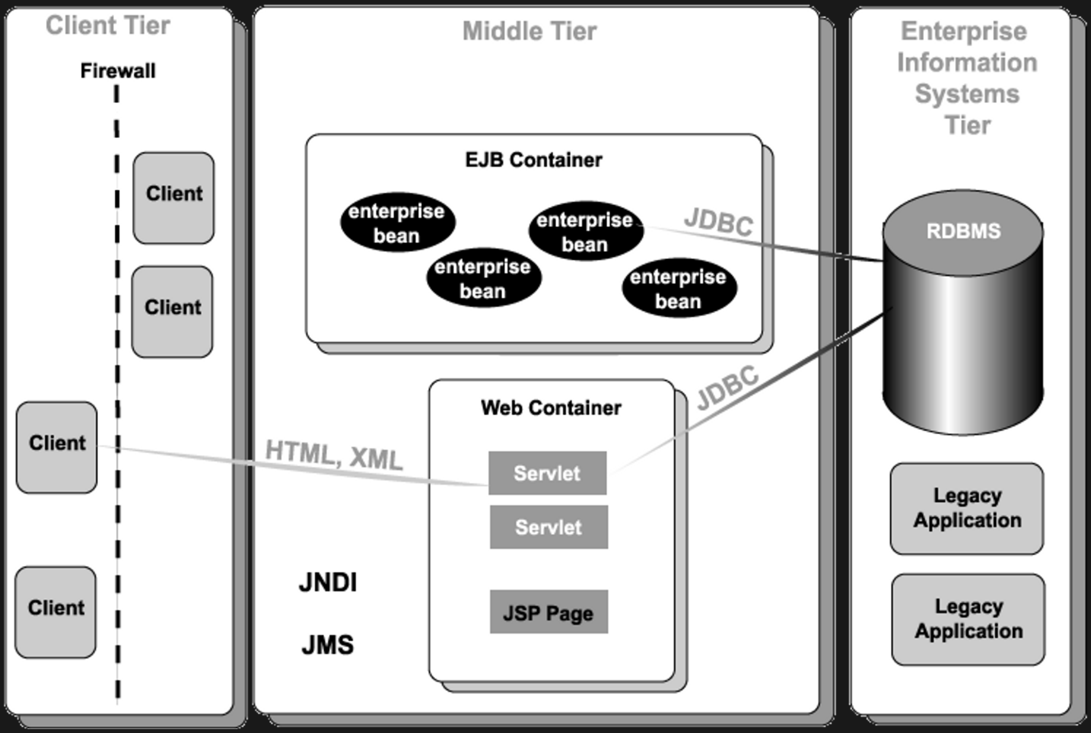
Application servers provide benefits in the areas of usability, flexibility, scalability, maintainability, and interoperability.

J2EE Application Architecture

The J2EE platform provides a multi-tiered distributed application model. Central to the J2EE component-based development model is the notion of containers. Containers are standardized runtime environments that provide specific services to components. Thus, Enterprise Java Beans (EJB) developed for a specific purpose in any organization can expect generic services such as transaction and EJB life cycle management to be available on any J2EE platform from any vendor.

Containers also provide standardized access to enterprise information systems; for example, providing RDBMS access through the JDBC API. Containers also provide a mechanism for selecting application behavior at assembly or deployment time.

As shown in Figure, the J2EE application architecture is a multi-tiered application model. In the middle tier, components are managed by containers: For example, J2EE Web containers invoke servlet behavior, and EJB containers manage life cycle and transactions for EJBs. The container-based model separates business logic from system infrastructure.



Spring Introduction:

In early days of Java based business application development, programmer used multiple Java bean classes (normal Java classes) to build the business logic layer/service layer.

The Business logic layer only required the enterprise capabilities like security, transaction-management, logging, mailing, messaging, etc. these enterprise capabilities are also known as extra service to the main business logic or the middleware services to make our main business logic perfect.

With the Java bean classes, developers are only responsible to define and add these middleware services to the main business logic, it increases burden to the developers.

To overcome the above burden to develop a business logic/service logic, sun-microsystem has released EJB technology as part of J2EE specification in 1998.

In EJB technology, programmer develop the main business logic and EJB-container provides these extra middleware services.

EJB reduced the middleware service development from the programmer, but it increased the complexities to access these middleware services.

EJB components are heavy weight components (Here our Java classes need to be developed as EJB component by implementing EJB technology related interfaces, need to override lost of unnecessary methods inside our Java classes and need to register these Java classes inside various xml files and deploy our EJB component inside the the Application Server software)

EJB has been very powerful but very complex to the build the Business Logic layer.

To the build the business logic layer, simplicity of Java bean classes + Power of EJB - complexities of EJB was realized in the industry.

Rod Johnson developed a framework called **Interface 21** to address the above need and rename it to the **Spring** and released in mid of 2004.

Spring is an application framework software, to develop an Enterprise application.

The software community treats spring as a framework of frameworks because it gives the support of various other frameworks also like Hibernate, Struts, JSF, etc.

Spring is an open-source, lightweight application f that can be used in all the layers of a java based business application (i.e. Presentation Layer, Business Logic Layer/Service Layer, Data Access Layer)

Spring makes programming Java quicker, easier, and safer for everybody. Spring’s focus on speed, simplicity, and productivity has made it the world's most popular Java framework.

Spring framework allows to write the business logic of a business application in a POJO class, and its **Spring container** provides the other services with less processing overhead.

Spring framework is considered a lightweight application framework by taking the following things into the consideration:

- Size of the spring container. (it is a simple java class, which can be activated inside any java application)
- Processing overhead.
- POJO and POJI programming model.
- No need to install any server to develop and run our business logic.

The Architecture of Spring framework:

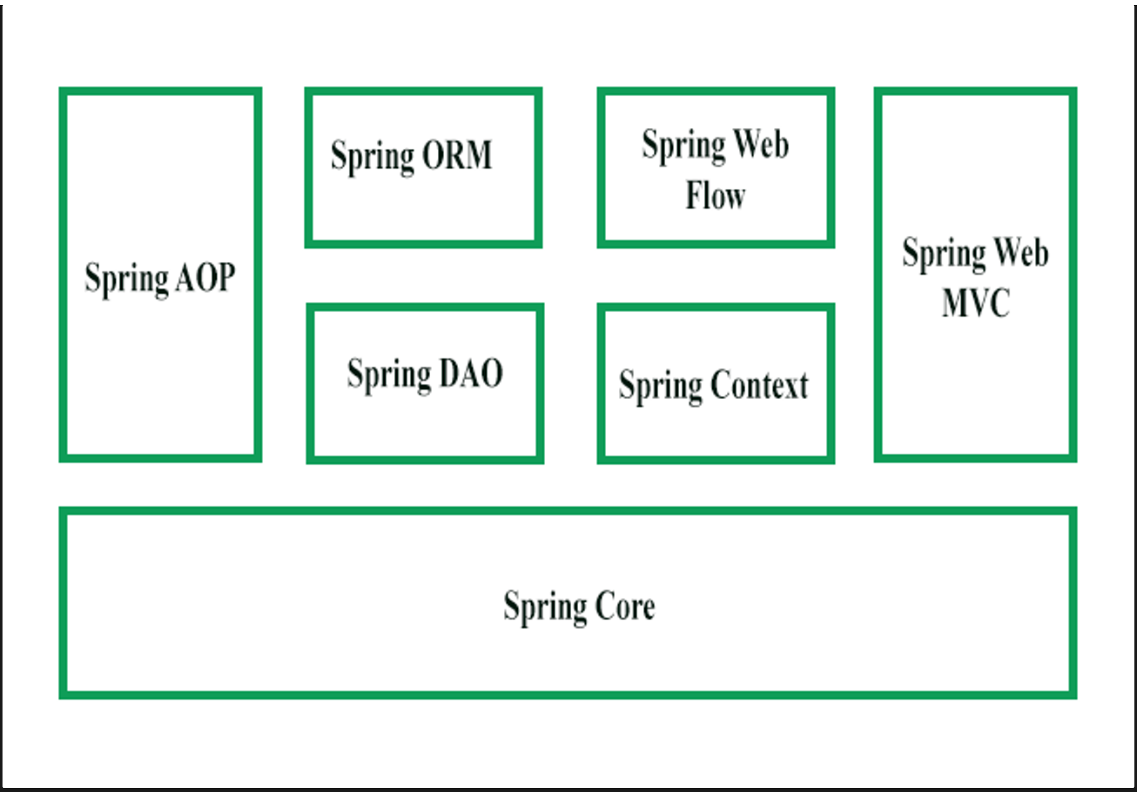
With the help of the Spring framework, we can develop all types of Java applications like:

Desktop(standalone) application, Remoting application, Web application, Enterprise application, etc.

Spring follows the modular architecture, which means spring has several modules to develop a Java Application, based on our requirement we can choose either all the modules or we can choose some specific modules to develop an application.

Some of the spring modules are:

- Spring core module (IOC container): this is the base module of the remaining other modules.
- AOP(Aspect-oriented programming) module: to apply the middleware services in a cross-cutting concern fashion.
- JDBC and DAO module: to develop a data access layer using the JDBC (abstraction on the plain JDBC)
- ORM module: to develop the data access layer using the ORM approach (abstraction over the ORM software).
- Web-MVC module: to develop the presentation layer in an easy manner using MVC architecture (Spring MVC).
- Test module: to conduct the unit test by using the mock objects.



Spring Core module:

Tight coupling and loose coupling between objects:

If one class calls another class functionality, then we can say that both classes are coupled with the other.

Example: If class A calls the functionality of class B then class A will be called a dependent class and class B will be called a dependency class.

```
class B{ public void funB(){ System.out.println("inside funB of B"); } } class A { // Dependent B b1 = new B(); //dependency public void funA(){ System.out.println("inside funA of A"); b1.funB(); } }
```

Here A class needs the service of B class then A will be dependent on B

If the dependent class wants to call the methods of the dependency class then, it has to create an object of its dependency class, and then the dependent class can call the functionality of its dependency class.

Now suppose, if any changes are made in the dependency class and if it is forced to do the changes inside the dependent class also then we can say that both classes are tightly coupled with each other.

Example:

```
//dependency class public class Car { public void start() { System.out.println("Car started..."); } } //dependent on the car class public class Travel { Car c=new Car(); public void journey() { c.start(); System.out.println("Jounrney started..."); } }
```

Here both the dependent and the dependency classes are tightly coupled with each other, because if change the method name start() to go() inside the dependency class then we need to change the same inside the dependent class also.

Tight coupling generates the problem in another way also, when the dependent class wants to change from the current dependency to another similar dependency ex:

```
//another dependency public class Bike { public void ride() { System.out.println("ride started...."); } }
```

Here if we change the dependency from Car to Bike we need to modify the dependent class also.

Example:

```
public class Travel { //Car c=new Car(); Bike b=new Bike(); public void journey() { //c.start(); b.ride(); System.out.println("Jounrney started..."); } }
```

In order to get loose coupling from dependent to its dependencies we need to follow the following rules:

1. Design the dependencies classes by the following POJO by the POJI model.
2. Apply the Dependency Injection mechanism.

Example:

```
//Vehicle.java interface Vehicle { public void go(); }
```

```
//dependency class class Car implements Vehicle{ public void start() {
System.out.println("Car started..."); } @Override public void go() { start(); } } class
Bike implements Vehicle{ public void ride() { System.out.println("ride started...."); }
@Override public void go() { ride(); } } //dependent on the car class class Travel {
//it is the dependency Vehicle v; //here we can store one of its implemented class obj.
//constructor injection point /*public Travel(Vehicle v) { this.v=v; } */ //setter
injection point public void setV(Vehicle v) { this.v=v; } public void journey() {
v.go(); System.out.println("Jounrney started..."); } } class Demo { public static void
main(String[] args) { //Travel tr=new Travel(new Car()); //injecting the dependency obj
to the dependent, by calling constrcutor injection. Travel tr=new Travel(); tr.setV(new
```