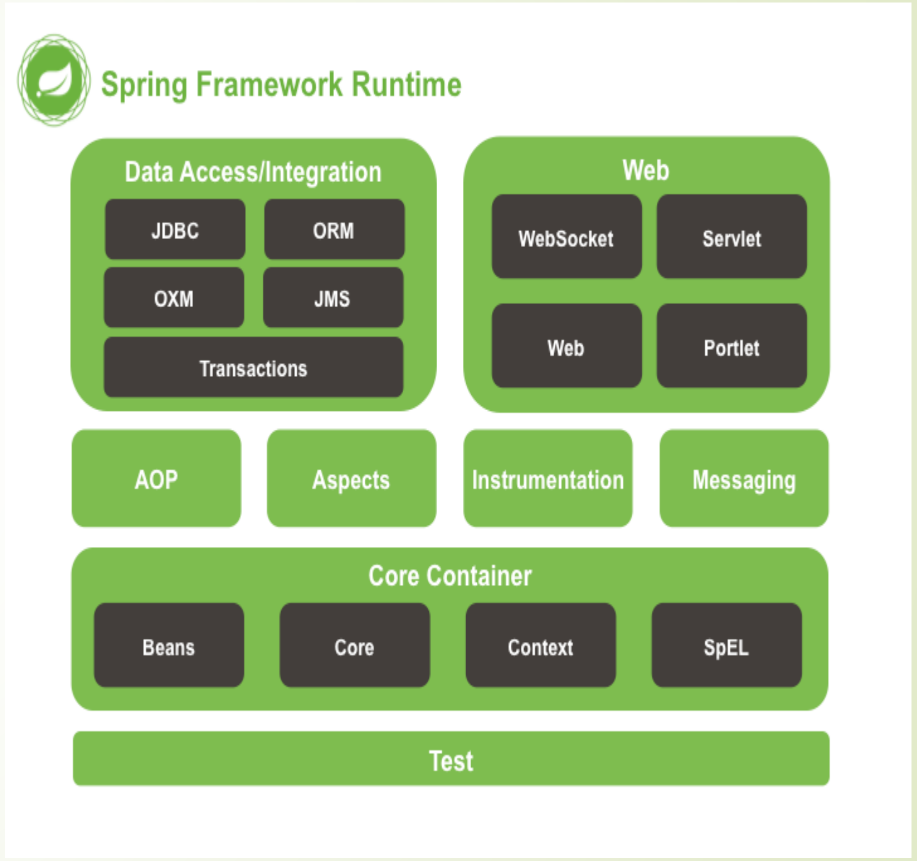
# Spring framework

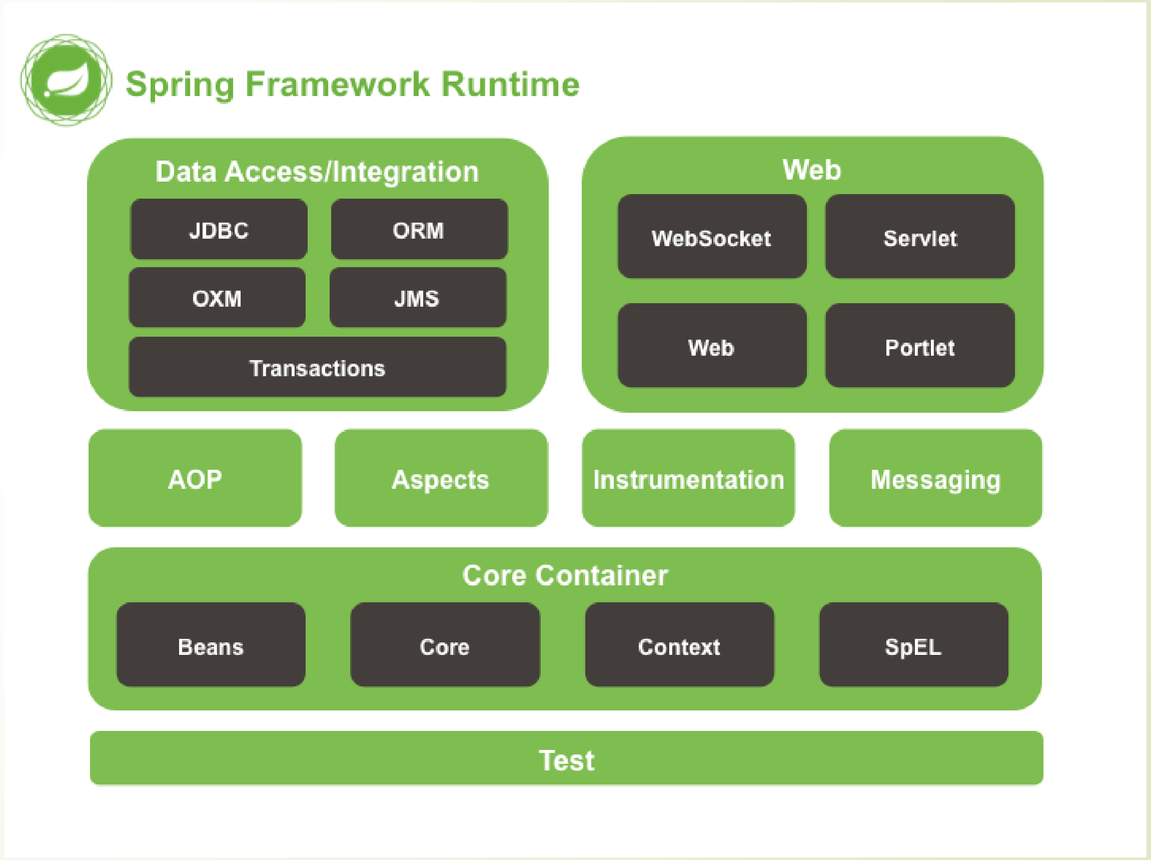
# Spring Introduction

- 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.
- Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.



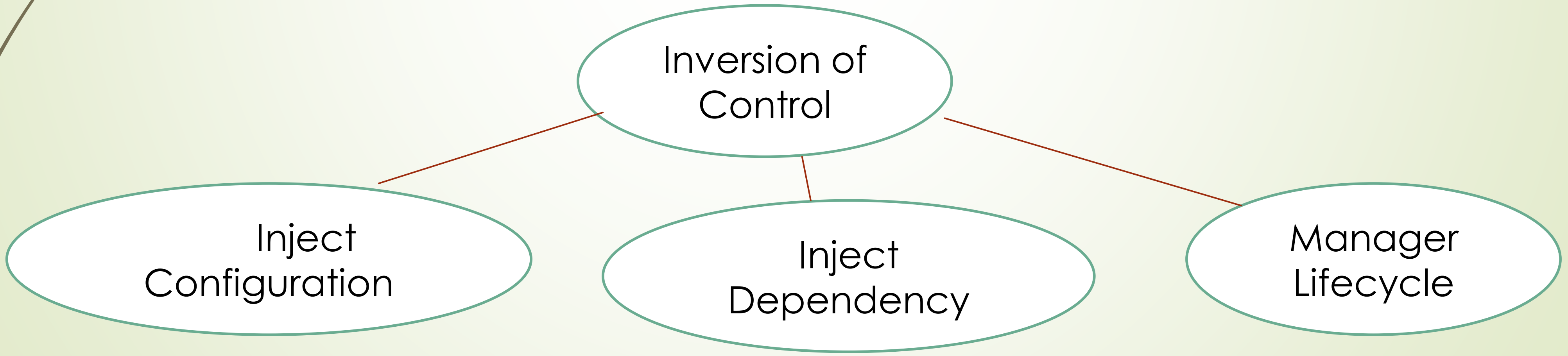
## Spring Modules

- The Spring Framework is divided into modules. Applications can choose which modules they need. At the heart are the modules of the core container, including a configuration model and a dependency injection mechanism



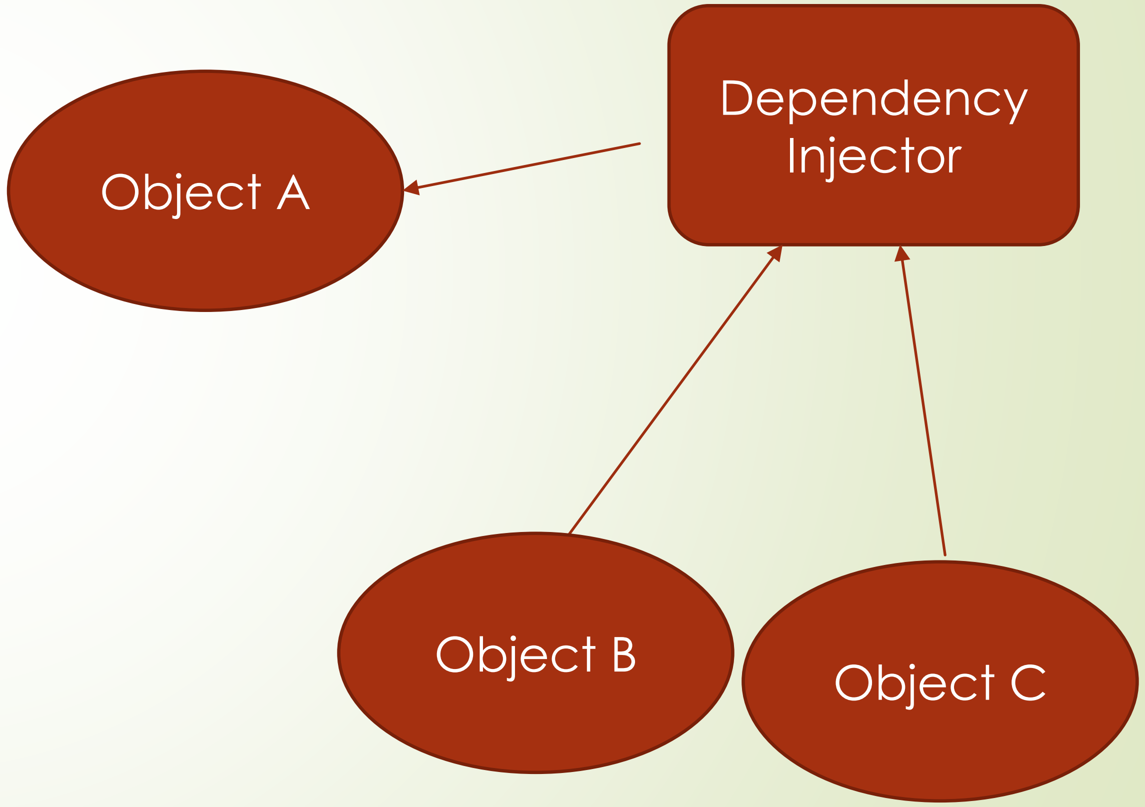
# IoC and Dependency Injection

- Inversion of Control is a design principal that allows classes to be loosely coupled
- In IoC ,the control of objects and container lies with a container or framework instead of main program



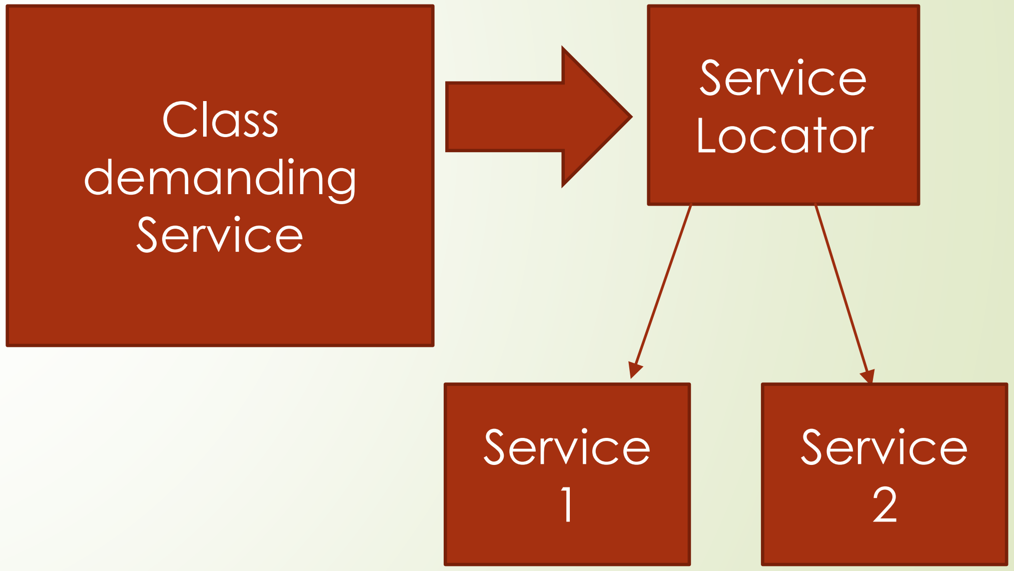
# Different Implementations of IoC

- Dependency Injection
- **Dependency injection** is a technique that allows objects to be separated from the objects they depend upon. For example, suppose object A requires a method of object B to complete its functionality, dependency injection suggests that instead of creating an instance of class B in class A using the new operator, the object of class B should be injected in class A using one of the following methods:
  - Constructor injection
  - Setter injection
  - Interface injection



# Different implementations of IoC

- Service Locator Pattern
- The **service locator pattern** includes service locator object that contains information about all the services that an application provides. Upon instantiation, the services register themselves with the service locator. Once it receives a request from a client, the service locator performs the required task and returns the needed values. This, also, reduces coupling between the dependent classes.



## Dependency Injection in Spring

- Spring provides ApplicationContext interface as representation for IoC container, that instantiates and manages the lifecycle of the objects that are a part of the program.
- ApplicationContext is sub-interface BeanFactory ,which is the root interface for accessing spring container.It provides basic functionality for accessing beans
- All functionalities of BeanFactory are available through ApplicationContext
- The information related to what objects and dependencies are required by the application is provided in configuration files. This file may be one of the following:
- XML
- Java Annotations
- Java code

## Spring Bean

- In Spring, a bean is an object that the Spring container instantiates, assembles, and manages.
- Spring beans form the back bone of application and are managed by Spring IoC container

## Creating Beans

- Beans can be created as POJO classes

```
package com.bean;
public class Pet {
	private String petName;
	private int age;
	private String type;
	public Pet()
	{
}
	public String getPetName() {
		return petName;
	}
```

```
    public void setPetName(String petName) {
        this.petName = petName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    @Override
    public String toString() {
        return "Pet [petName=" + petName + ", age=" + age + ", type=" + type + "]";
    }
}
```

## Spring Context

- Spring contexts are also called Spring IoC containers, which are responsible for instantiating, configuring, and assembling beans by reading configuration metadata from XML, Java annotations, and/or Java code in the configuration files.
- ApplicationContext interface is available through spring context library

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.0.RELEASE</version>
</dependency>
```

## Creating bean configuration through xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd"
 >
 <bean id="pet" class="com.bean.Pet">
   <property name="petName" value="Sam"></property>
   <property name="age" value="5"></property>
   <property name="type" value="cat"></property>
 </bean>
 </beans>
```

## Parts of bean definition

- Id:used to uniquely identify a bean
- class:fully qualified class name for which bean has to be created
- property:Properties of class for which bean has to be created

## Getting bean information from configuration file

```
package SamplePetStore.petstoredemo;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.bean.Pet;
public class App
{
    public static void main( String[] args )
    {
ApplicationContext context=new ClassPathXmlApplicationContext("springbeans.xml");

        Pet pet=context.getBean(Pet.class, "pet");
        System.out.println(pet);
    }
}
```

## Autowiring using xml configuration

- Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive and string values. It works with reference only.
- Types of autowiring
  - no
  - byName
  - byType
  - constructor
  - autodetect

# Autowiring Example

```
<bean id="petstore" class="com.bean.PetStore">
 <property name="storeName" value="Happy Pet Store"/>
 <property name="managerName" value="Peter Joe"/>
 </bean>

 <bean id="pet" class="com.bean.Pet" autowire="byType">
   <property name="petName" value="Mac"/>
   <property name="age" value="5"/>
   <property name="type" value="Cat"/>
 </bean>
```

# Using Annotation for configuration

- The <*context:annotation-config*> annotation is mainly used to activate the dependency injection annotations.
- *@Autowired*, *@Qualifier*, *@PostConstruct*, *@PreDestroy*, and *@Resource* are some of the ones that <*context:annotation-config*> can resolve.
- **<*context:annotation-config*> activates the annotations only for the beans already registered in the application context**.

# Autowiring using annotation

- @Autowired  annotation can be applied upon property ,constructor or setter method
- package com.bean;
- import org.springframework.beans.factory.annotation.Autowired;

```
public class Pet {
  …
…
       @Autowired
       private PetStore petStore;
```

```
@Autowired
    public Pet(PetStore petStore)
    {
        this.petStore=petStore;
    }

OR
@Autowired
public void setPetStore(PetStore petStore)
    {
        this.petStore=petStore;
    }
```

- Sample Case Study

# Java Based Configuration

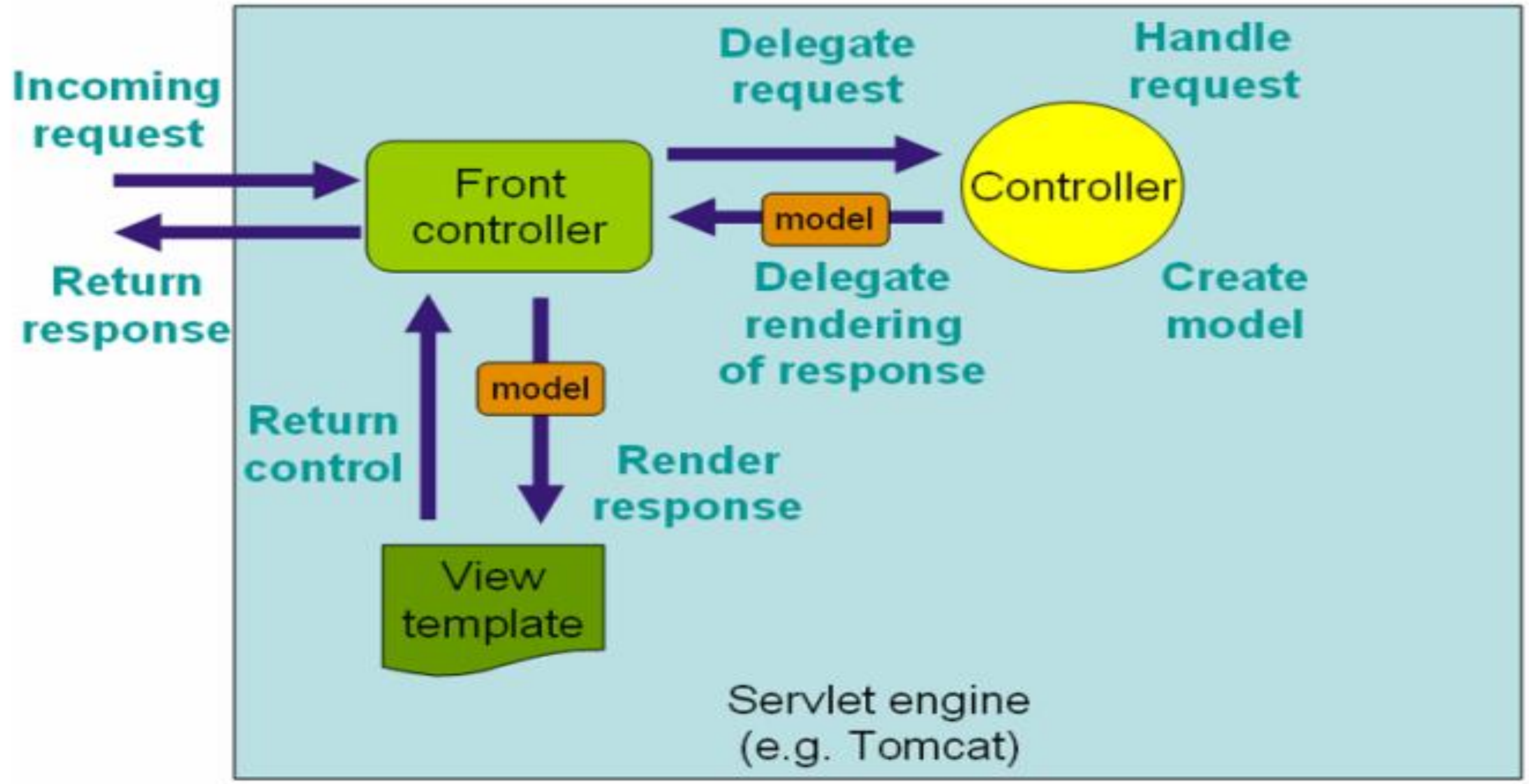
- Java-based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations
- @Configuration :Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- @Bean: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.
- @Component:allows Spring to detect our custom beans automatically.

In other words, without having to write any explicit code, Spring will:

- Scan our application for classes annotated with *@Component*
- Instantiate them and inject any specified dependencies into them
- Inject them wherever needed

## Spring MVC

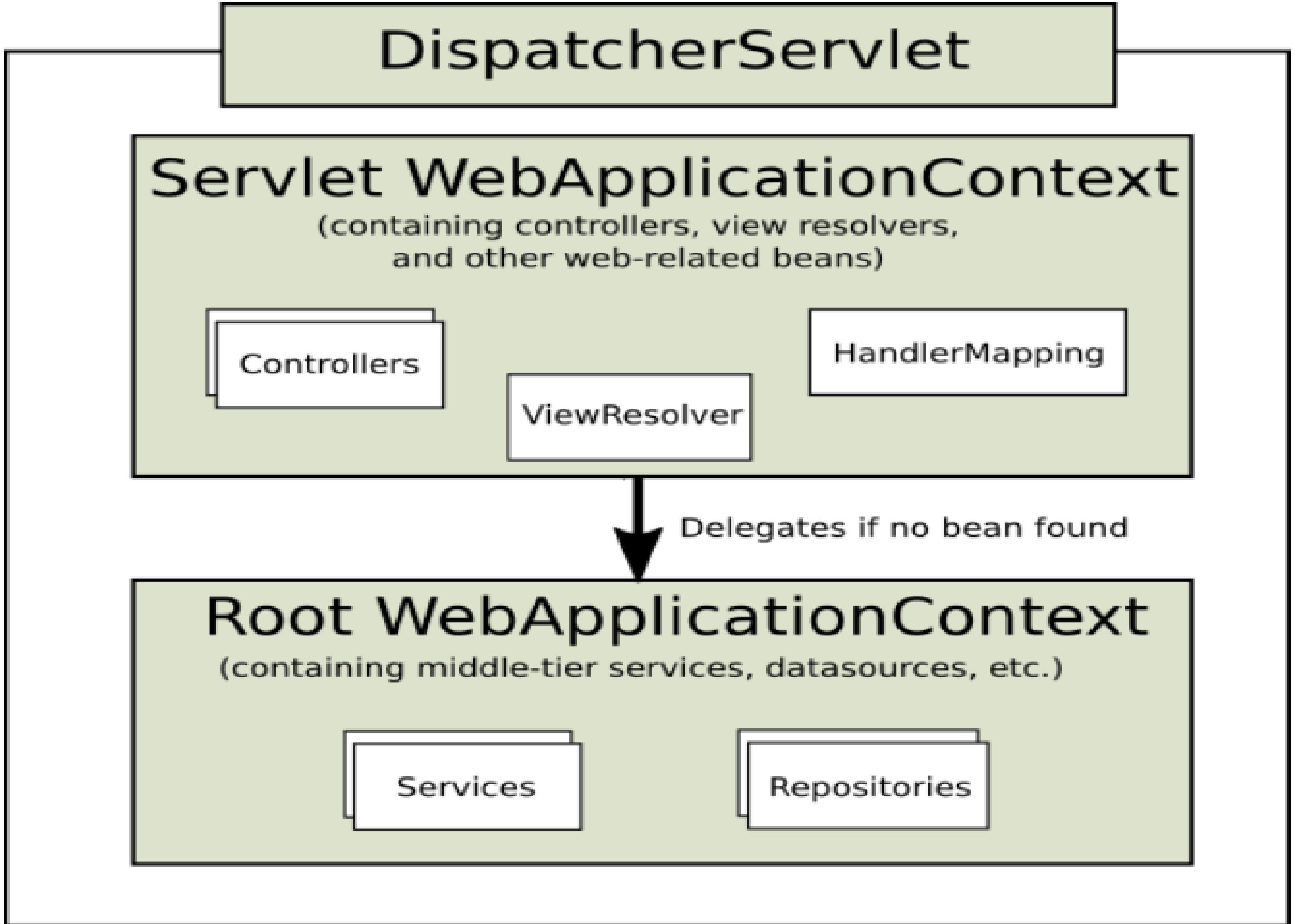
- The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods. With the introduction of Spring 3.0, the @Controller mechanism also allows you to create RESTful Web sites and applications, through the @PathVariable annotation and other features


Incoming request
Return response
Front controller
Delegate request
Handle request
Controller
model
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)

## Dispatcher Servlet

- The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class), and as such is declared in the web.xml of your web application. You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file. This is standard Java EE Servlet configuration; the following example shows such a DispatcherServlet declaration and mapping:

```
<web-app>
 <servlet>
 <servlet-name>example</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
 <servlet-name>example</servlet-name>
 <url-pattern>/example/*</url-pattern>
 </servlet-mapping>
</web-app>
```


DispatcherServlet
Servlet WebApplicationContext
(containing controllers, view resolvers,
and other web-related beans)
Controllers
ViewResolver
HandlerMapping
Delegates if no bean found
Root WebApplicationContext
(containing middle-tier services, datasources, etc.)
Services
Repositories

Upon initialization of a DispatcherServlet, Spring MVC looks for a file named [servlet-name]- servlet.xml in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.