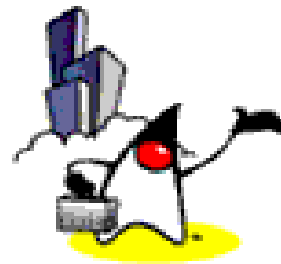




# Spring Framework Basics

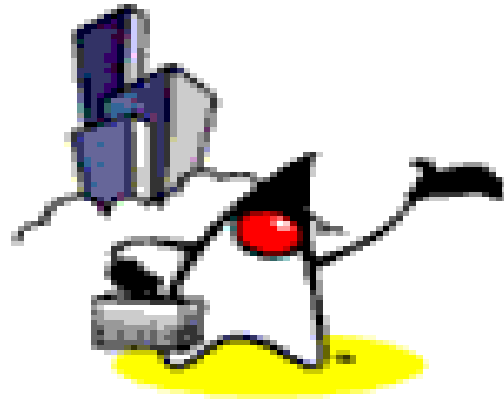


# Topics

- What is Spring framework?
- Why Spring framework?
- Spring framework architecture
- Usage scenario
- Dependency Injection (DI)
  - BeanFactory
  - Autowiring
  - ApplicationContext

# Topics covered in other presentations

- Refactoring HelloWorld application using Spring framework
- Spring framework and data access (persistence)
- Spring AOP (Aspect-Oriented Programming)
- Spring Web



# What is Spring Framework?

# What is Spring Framework?

- Light-weight yet comprehensive framework for building Java SE and Java EE applications

# Key Features

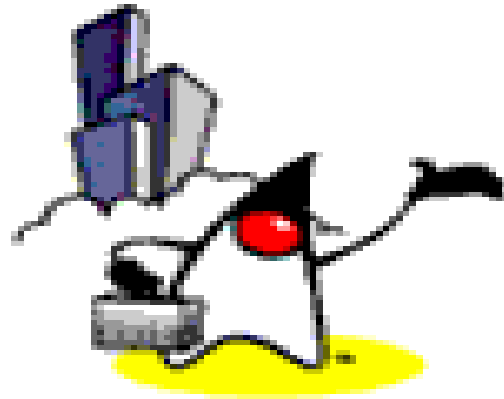
- JavaBeans-based configuration management, applying Inversion-of-Control principles, specifically using the **Dependency Injection** technique
  - This aims to reduce dependencies of components on specific implementations of other components.
- A core bean factory, which is usable globally
- Generic abstraction layer for database transaction management

# Key Features

- Built-in generic strategies for JTA and a single JDBC DataSource
  - This removes the dependency on a Java EE environment for transaction support.
- Integration with persistence frameworks Hibernate, JDO and iBATIS.
- MVC web application framework, built on core Spring functionality, supporting many technologies for generating views, including JSP, FreeMarker, Velocity, Tiles, iText, and POI.

# Key Features

- Extensive aspect-oriented programming (AOP) framework to provide services such as transaction management
  - As with the Inversion-of-Control parts of the system, this aims to improve the modularity of systems created using the framework.



# Why Use Spring Framework?

# Why Use Spring?

- Wiring of components through Dependency Injection
  - Promotes de-coupling among the parts that make the application
- Design to interfaces
  - Insulates a user of a functionality from implementation details
- Test-Driven Development (TDD)
  - POJO classes can be tested without being tied up with the framework

# Why Use Spring?

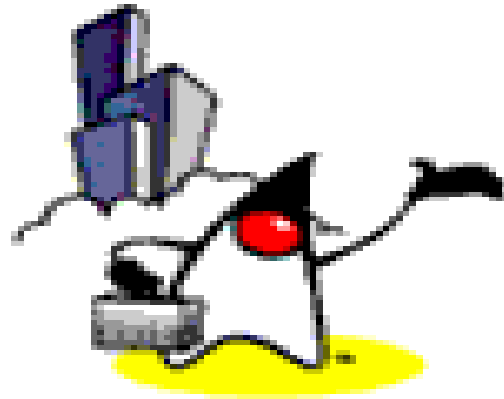
- Declarative programming through AOP
  - Easily configured aspects, esp. transaction support
- Simplify use of popular technologies
  - Abstractions insulate application from specifics, eliminate redundant code
  - Handle common error conditions
  - Underlying technology specifics still accessible

# Why Use Spring?

- Conversion of checked exceptions to unchecked
  - (Or is this a reason not to use it?)
- Not an all-or-nothing solution
  - Extremely modular and flexible
- Well designed
  - Easy to extend
  - Many reusable classes

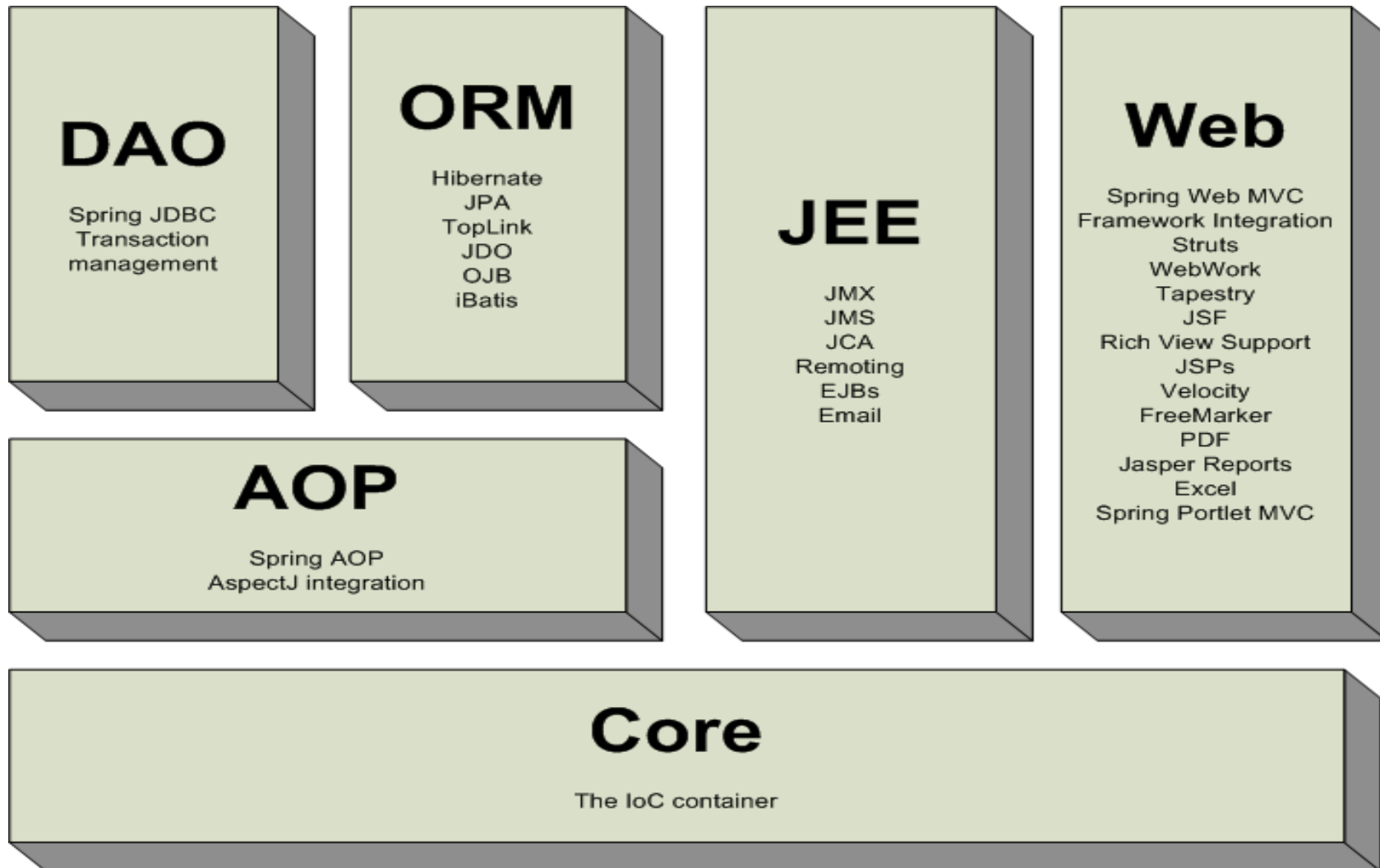
# Why Use Spring?

- Integration with other technologies
  - EJB for J2EE
  - Hibernate, iBates, JDBC (for data access)
  - Velocity (for presentation)
  - Struts and WebWork (For web)



# Spring Framework Architecture

# Spring Framework



# Core Package

- Core package is the most fundamental part of the framework and provides the **IoC and Dependency Injection** features
- The basic concept here is the **BeanFactory**, which provides a sophisticated implementation of the factory pattern which removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic

# DAO Package

- The DAO package provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes
- The JDBC package provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for all your POJOs (plain old Java objects)

# ORM Package

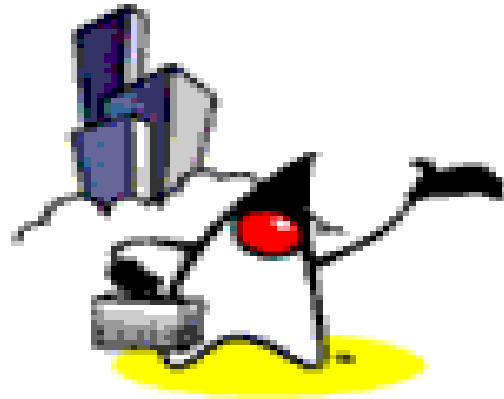
- The ORM package provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- Using the ORM package you can use all those O/R-mappers in combination with all the other features Spring offers, such as the simple declarative transaction management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

# AOP Package

- Spring's AOP package provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated
- Using source-level metadata functionality you can also incorporate all kinds of behavioral information into your code

# MVC Package

- Spring's MVC package provides a Model-View-Controller (MVC) implementation for web-applications
- Spring's MVC framework is not just any old implementation; it provides a clean separation between domain model code and web forms, and allows you to use all the other features of the Spring Framework.

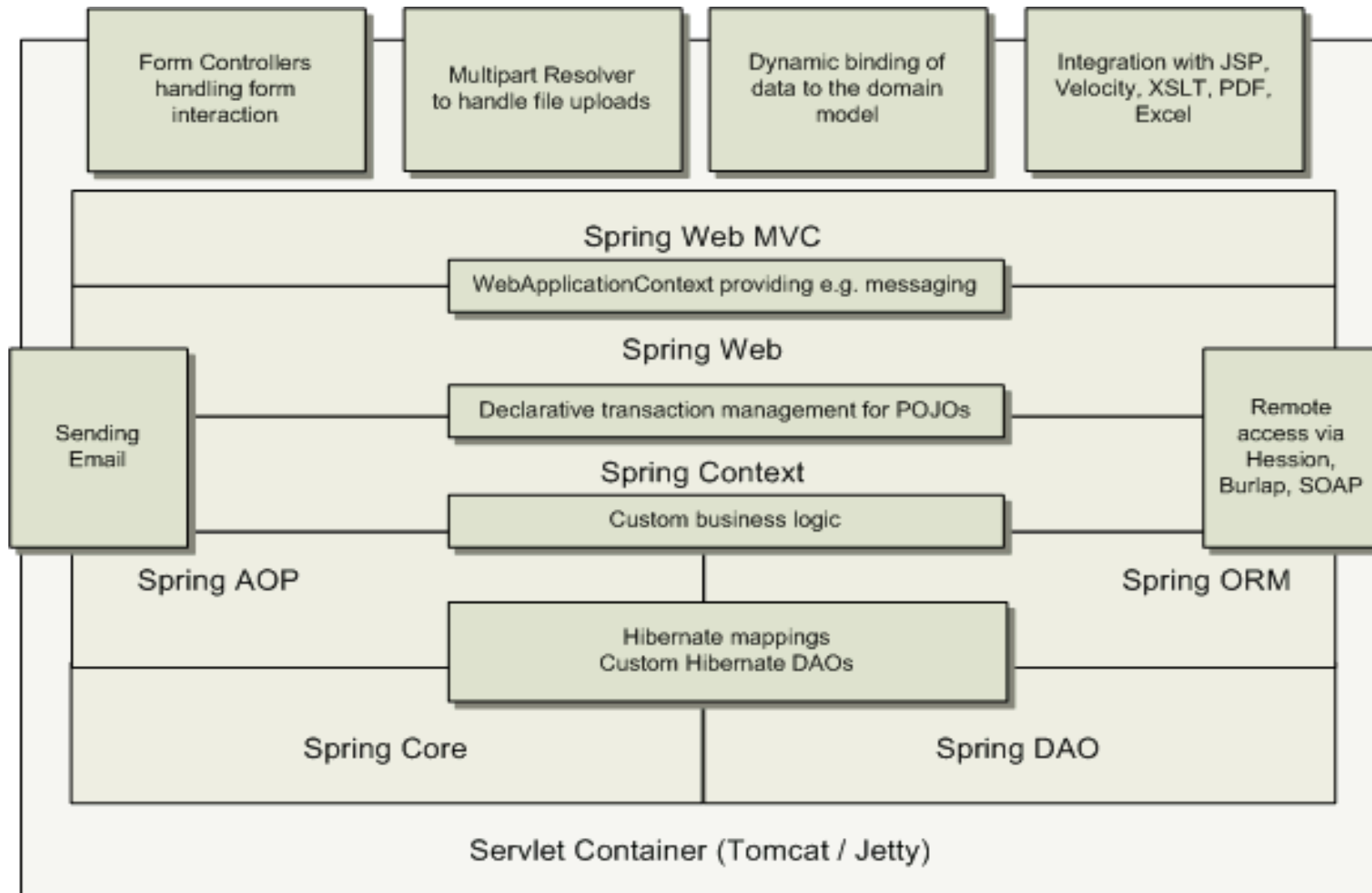


# Usage Scenarios

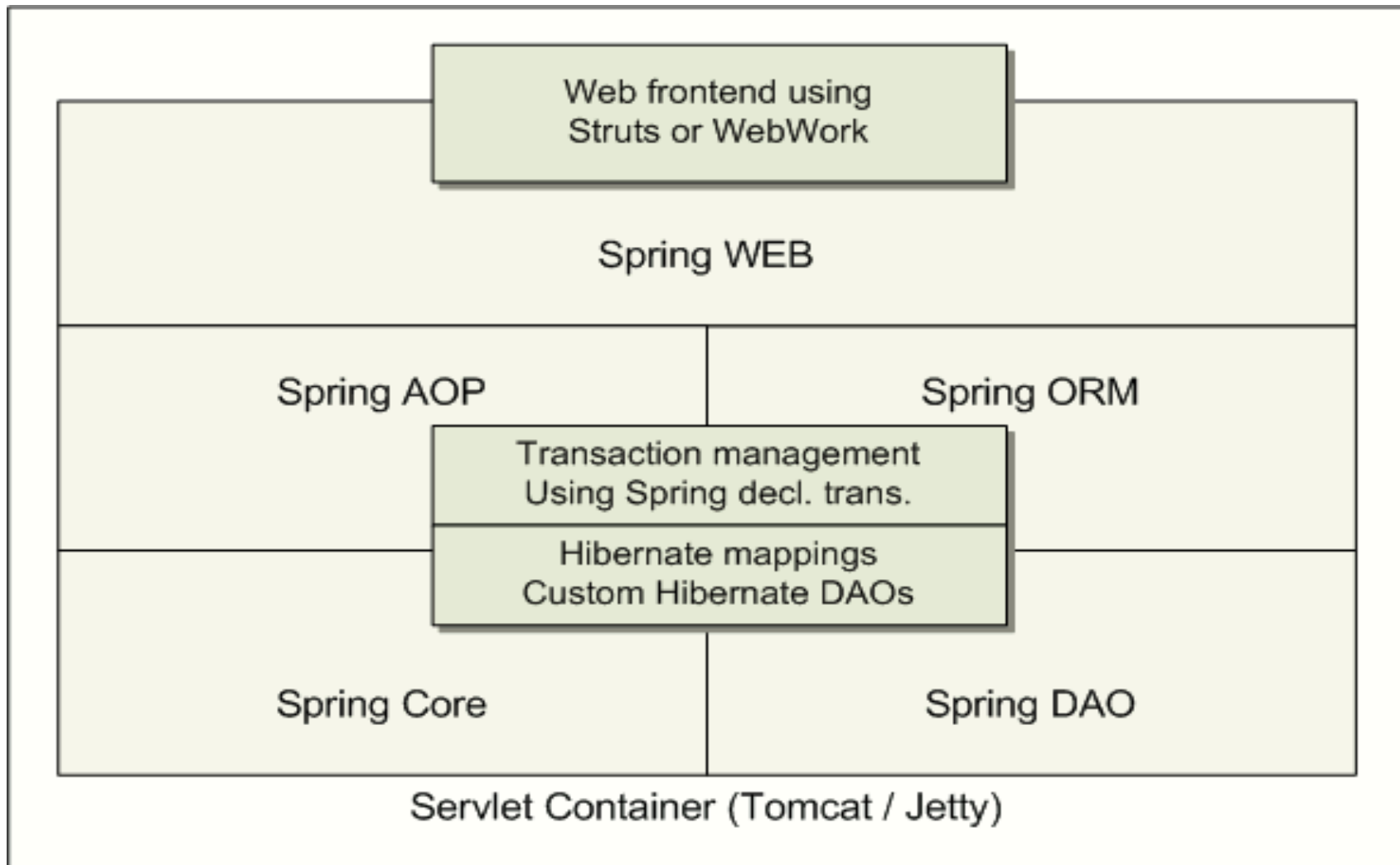
# Usage Scenarios

- You can use Spring in all sorts of scenarios, from applets up to fully-fledged enterprise applications using Spring's transaction management functionality and web framework integration

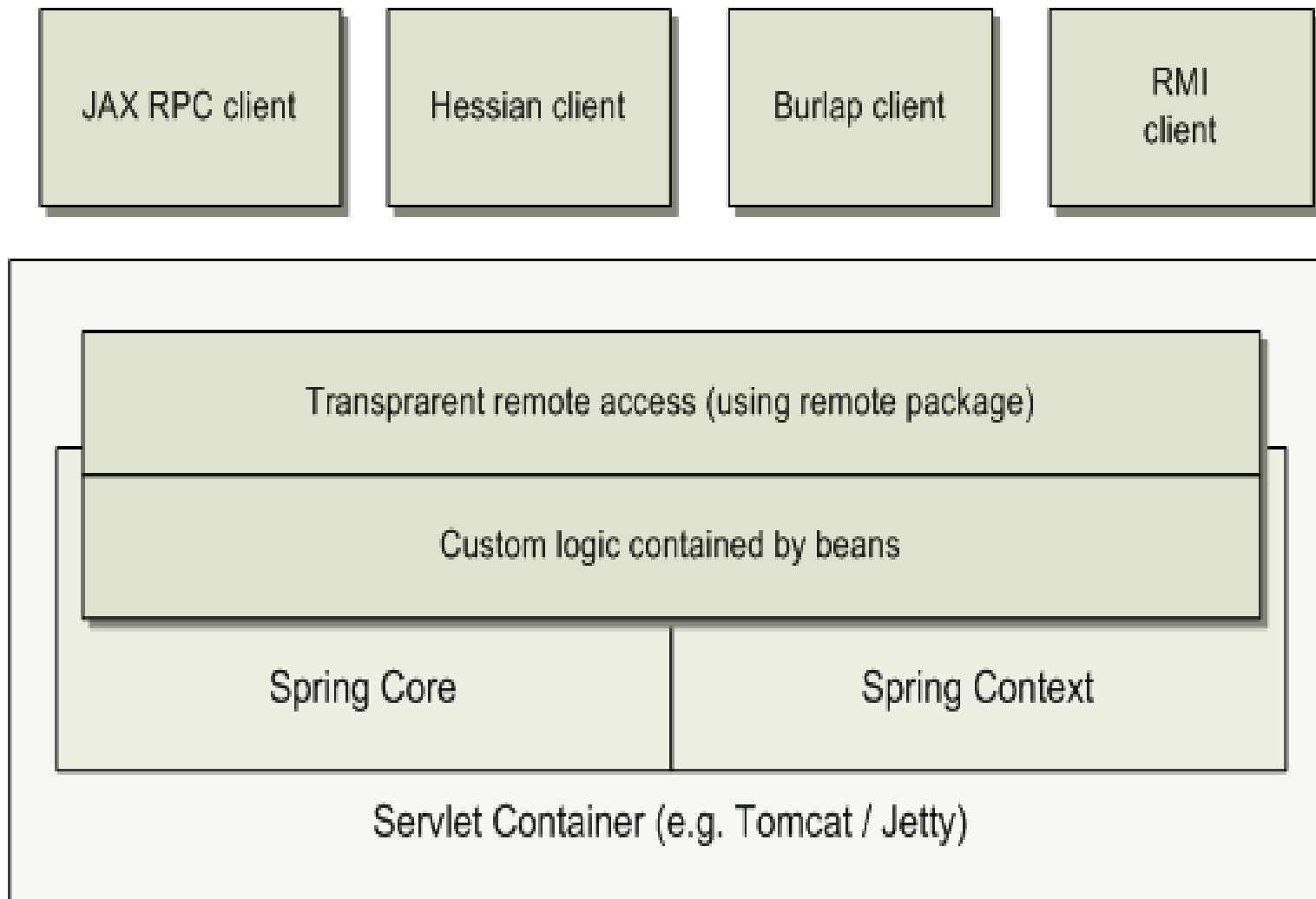
# Typical Full-fledged Spring Web Application



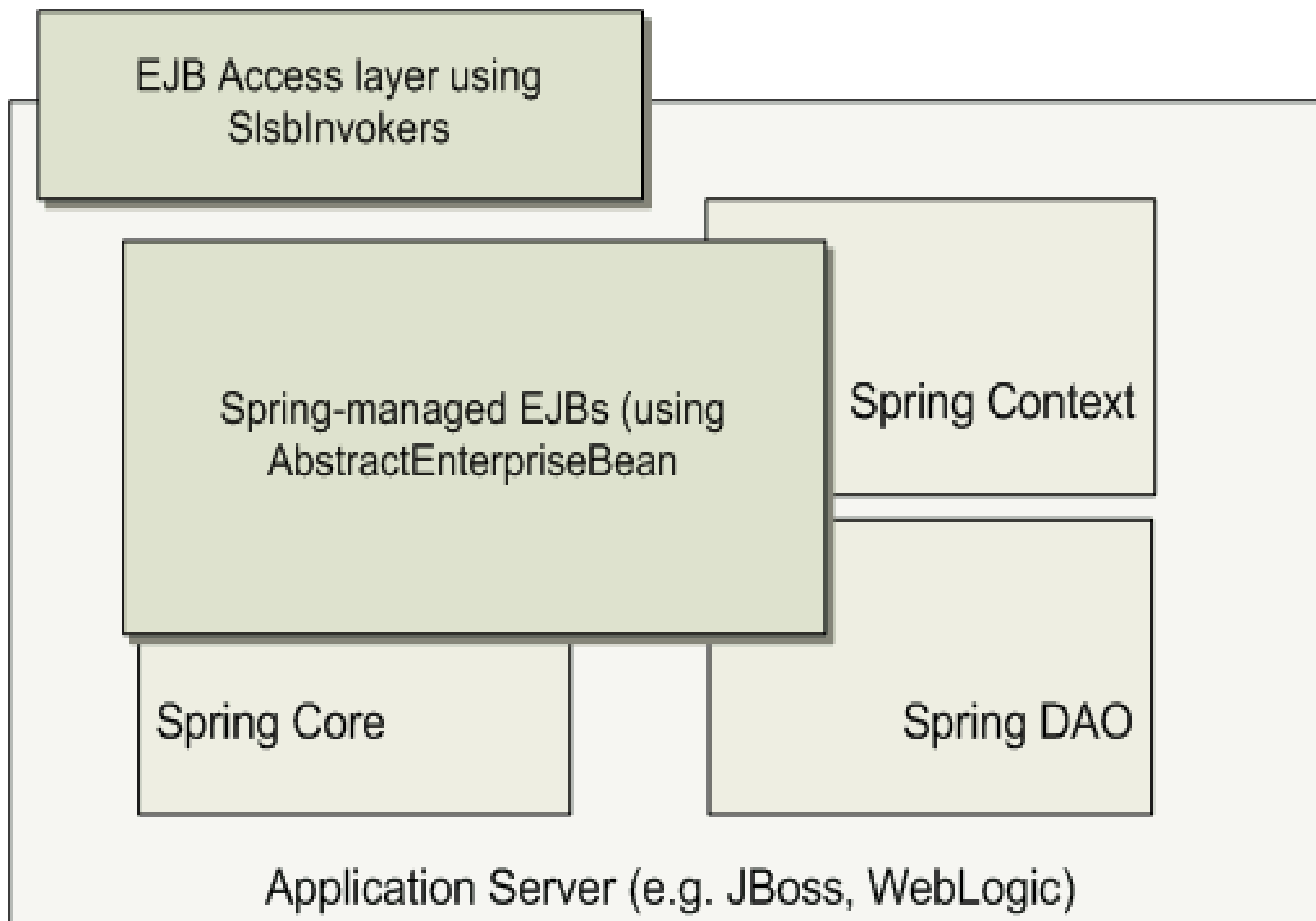
# Spring Middle-tier Using 3<sup>rd</sup> party Web Framework

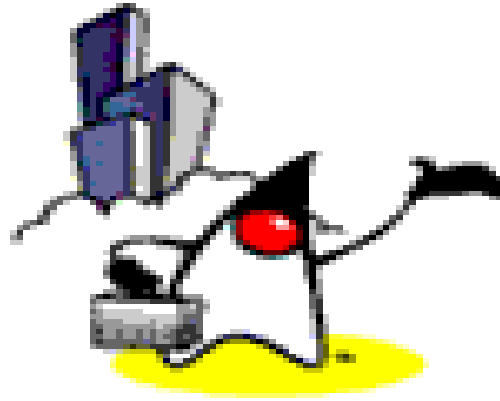


# Remoting Usage Scenario



# EJBs – Wrapping Existing POJOs





# Dependency Injection (DI): Basic concept

# Spring Dependency Injection

- A kind of Inversion of Control (IoC)
- “Hollywood Principle”
  - Don't call me, I'll call you
- “Container” resolves (injects) dependencies of components by setting implementation object (push)
- As opposed to component instantiating or Service Locator pattern where component locates implementation (pull)
- Martin Fowler calls Dependency Injection

# Benefits of Dependency Injection

- Flexible
  - Avoid adding lookup code in business logic
- Testable
  - No need to depend on external resources or containers for testing
- Maintainable
  - Allows reuse in different application environments by changing configuration files instead of code
  - Promotes a consistent approach across all applications and teams

# Two Dependency Injection Variants

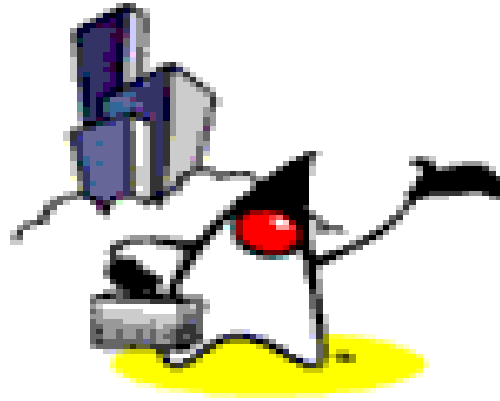
- Constructor dependency Injection
  - Dependencies are provided through the constructors of the component
- Setter dependency injection
  - Dependencies are provided through the JavaBean-style setter methods of the component
  - More popular than Constructor dependency injection

# Constructor Dependency Injection

```
public class ConstructorInjection {  
  
    private Dependency dep;  
  
    public ConstructorInjection(Dependency dep) {  
        this.dep = dep;  
    }  
}
```

# Setter Dependency Injection

```
public class SetterInjection {  
  
    private Dependency dep;  
  
    public void setMyDependency(Dependency dep) {  
        this.dep = dep;  
    }  
}
```



# Dependency Injection (DI): DI Support in Spring

# Sub-topics

- *BeanFactory* interface
- *XmlBeanFactory* implementation
- Bean configuration file
  - Setter dependency injection
  - Constructor dependency injection
- Beans
- Injection parameters

# BeanFactory

- BeanFactory object is responsible for managing beans and their dependencies
- Your application interacts with Spring's DI container through BeanFactory interface
  - BeanFactory object has to be created by the application typically XmlBeanFactory
  - BeanFactory object, when it gets created, read bean configuration file and performs the wiring
  - Once created, the application can access the beans via BeanFactory interface

# BeanFactory Implementations

- XmlBeanFactory
  - Convenience extension of DefaultListableBeanFactory that reads bean definitions from an XML document

# Reading XML Configuration File via XmlBeanFactory class

```
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class XmlConfigWithBeanFactory {

    public static void main(String[] args) {
        XmlBeanFactory factory =
            new XmlBeanFactory(new FileSystemResource("beans.xml"));
        SomeBeanInterface b =
            (SomeBeanInterface) factory.getBean("nameOftheBean");
    }
}
```

# Bean Configuration File

- Each bean is defined using `<bean>` tag under the root of the `<beans>` tag
- The `id` attribute is used to give the bean its default name
- The `class` attribute specifies the type of the bean

# Bean Configuration File Example: Setter DI

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
  "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
```

```
  <bean id="renderer" class="StandardOutMessageRenderer">
```

```
    <property name="messageProvider">
```

```
      <ref local="provider"/>
```

```
    </property>
```

```
  </bean>
```

```
  <bean id="provider" class="HelloWorldMessageProvider"/>
```

```
</beans>
```

# Bean Configuration File Example: Constructor DI

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="renderer" class="StandardOutMessageRenderer">
    <property name="messageProvider">
      <ref local="provider"/>
    </property>
  </bean>
  <bean id="provider" class="ConfigurableMessageProvider">
    <constructor-arg>
      <value>This is a configurable message</value>
    </constructor-arg>
  </bean>
</beans>
```

# Bean Example: Constructor DI

```
public class ConfigurableMessageProvider implements
    MessageProvider {

    private String message;

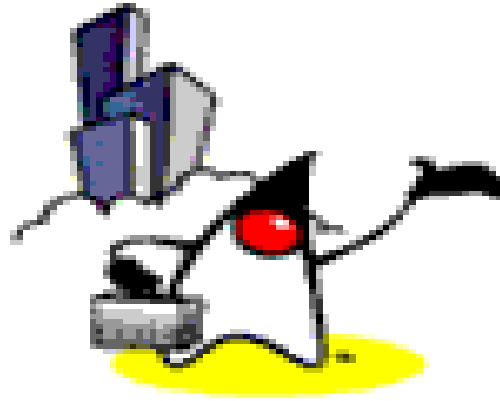
    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

}
```

# Beans

- The term “bean” is used to refer any component managed by the BeanFactory
- The “beans” are in the form of JavaBeans (in most cases)
  - no arg constructor
  - getter and setter methods for the properties
- Beans are singletons by default
- Properties the beans may be simple values or references to other beans
- Beans can have multiple names



# Dependency Injection (DI): Injection Parameter Types

# Injection Parameter Types

- Spring supports various kinds of injection parameters
  1. Simple values
  2. Beans in the same factory
  3. Beans in another factory
  4. Collections
  5. Externally defined properties
- You can use these types for both setter or constructor injections

# 1. Injecting Simple Values

```
<beans>
```

```
  <!-- injecting built-in vals sample -->
  <bean id="injectSimple" class="InjectSimple">
    <property name="name">
      <value>John Smith</value>
    </property>
    <property name="age">
      <value>35</value>
    </property>
    <property name="height">
      <value>1.78</value>
    </property>
    <property name="isProgrammer">
      <value>true</value>
    </property>
    <property name="ageInSeconds">
      <value>1103760000</value>
    </property>
  </bean>
```

```
</beans>
```

## 2. Injecting Beans in the same Factory

- Used when you need to inject one bean into another (target bean)
- Configure both beans first
- Configure an injection using `<ref>` tag in the target bean's `<property>` or `<constructor-arg>`
- The type being injected does not have to be the exact type defined on the target
  - if the type defined on the target is an interface, the type being injected must be an implementation of it
  - if the type defined on the target is a class, the type being injected can be the same type or sub-type

## 2. Injecting Beans in the same Factory

```
<beans>
```

```
  <!-- oracle bean used for a few examples -->
```

```
  <bean id="oracle" name="wiseworm" class="BookwormOracle"/>
```

```
  <!-- injecting reference sample (using id) -->
```

```
  <bean id="injectRef" class="InjectRef">
```

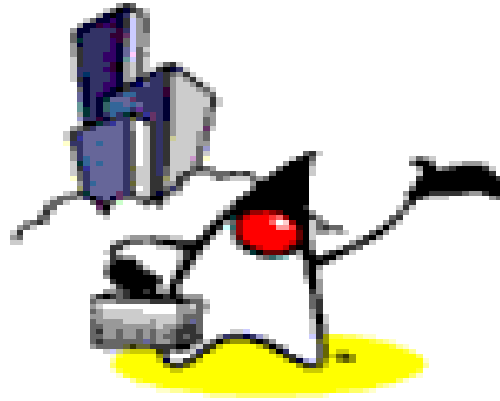
```
    <property name="oracle">
```

```
      <ref local="oracle"/>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```



# Dependency Injection (DI): Bean Naming

# Bean Naming

- Each bean must have at least one name that is unique within the containing BeanFactory
- Name resolution procedure
  - If a <bean> tag has an id attribute, the value of the id attribute is used as the name
  - If there is no id attribute, Spring looks for name attribute
  - If neither id nor name attribute are defined, Spring use the class name as the name
- A bean can have multiple names
  - Specify comma or semicolon-separated list of names in the name attribute

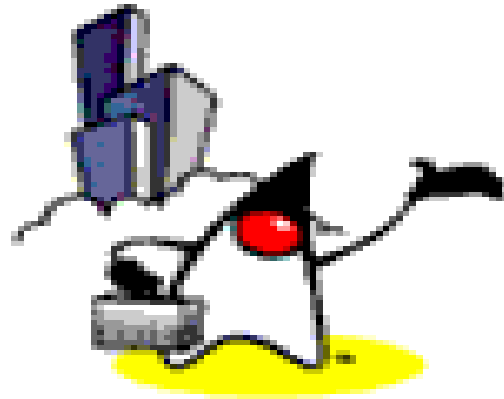
# Bean Naming Example

```
<bean id="mybeanid" class="mypackage.MyClass"/>
```

```
<bean name="mybeanname" class="mypackage.MyClass"/>
```

```
<bean class="mypackage.MyClass"/>
```

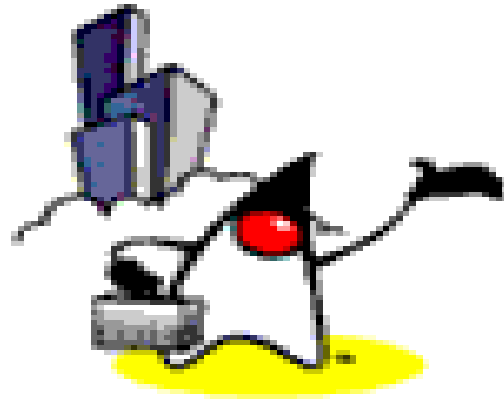
```
<bean id="mybeanid" name="name1,name2,name3"  
    class="mypackage.MyClass"/>
```



# Dependency Injection: Autowiring

# Autowiring Properties

- Beans may be auto-wired (rather than using <ref>)
  - Per-bean attribute *autowire*
  - *Explicit settings override*
- *autowire="name"*
  - Bean identifier matches property name
- *autowire="type"*
  - Type matches other defined bean
- *autowire="constructor"*
  - Match constructor argument types
- *autowire="autodetect"*
  - Attempt by constructor, otherwise "type"



# ApplicationContext

# What is ApplicationContext?

- Extension of BeanFactory
  - It provides all the same functionality and more
  - Reduces the amount of code you need
  - In a more framework-oriented style
- Add new features over BeanFactory
  - Resource management and access
  - Additional life-cycle interfaces
  - Improved automatic configuration of infrastructure components
  - Event publication
  - Internationalization

# When to Use ApplicationContext?

- Use ApplicationContext over BeanFactory to take advantage of its extended functionality
  - Except for a few limited situations such as perhaps in an Applet, where memory consumption might be critical, and a few extra kilobytes might make a difference

# Using MessageSource

- The ApplicationContext interface extends an interface called MessageSource, and therefore provides messaging (i18n or internationalization) functionality.

```
<beans>
  <bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSou
    rce">
    <property name="basenames">
      <list>
        <value>format</value>
        <value>exceptions</value>
        <value>windows</value>
      </list>
    </property>
  </bean>
</beans>
```

# Propagating Events

- Event handling in the `ApplicationContext` is provided through the `ApplicationEvent` class and `ApplicationListener` interface
  - If a bean which implements the `ApplicationListener` interface is deployed into the context, every time an `ApplicationEvent` gets published to the `ApplicationContext`, that bean will be notified
  - Essentially, this is the standard Observer design pattern.

# Three Built-in Events

- `ContextRefreshEvent`
  - `ApplicationContext` is initialized or refreshed
- `ContextClosedEvent`
  - `ApplicationContext` is closed
- `RequestHandleEvent`
  - A web-specific event telling all beans that a HTTP request has been serviced

# Example: Event Handling

- Configuration

```
<bean id="emailer" class="example.EmailBean">  
  <property name="blackList">  
    <list>  
      <value>black@list.org</value>  
      <value>white@list.org</value>  
      <value>john@doe.org</value>  
    </list>  
  </property>  
</bean>
```

```
<bean id="blackListListener" class="example.BlackListNotifier">  
  <property name="notificationAddress" value="spam@list.org"/>  
</bean>
```

# Example: Event Handling

- Bean class

```
public class EmailBean implements ApplicationContextAware {  
  
    /** the blacklist */  
    private List blacklist;  
  
    public void setBlackList(List blackList) {  
        this.blackList = blackList;  
    }  
  
    public void setApplicationContext(ApplicationContext ctx) {  
        this.ctx = ctx;  
    }  
  
    public void sendEmail(String address, String text) {  
        if (blackList.contains(address)) {  
            BlackListEvent evt = new BlackListEvent(address, text);  
            ctx.publishEvent(evt);  
            return;  
        }  
        // send email  
    }  
}
```

# Example: Event Handling

- Notifier class

```
public class BlackListNotifier implement ApplicationListener {  
  
    /** notification address */  
    private String notificationAddress;  
  
    public void setNotificationAddress(String notificationAddress) {  
        this.notificationAddress = notificationAddress;  
    }  
  
    public void onApplicationEvent(ApplicationEvent evt) {  
        if (evt instanceof BlackListEvent) {  
            // notify appropriate person  
        }  
    }  
}
```

# Example

- U  
- E

# Example

- U  
- E

# How to use `ApplicationContext`?

- Many users will use `ApplicationContext` in a completely declarative fashion, not even having to create it manually, but instead relying on support classes such as `ContextLoader` to automatically start an `ApplicationContext` as part of the normal startup process of a J2EE web-app
  - it is still possible to programmatically create an `ApplicationContext`



# Spring Framework Basics

