Spring Framework Basics

Topics

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

Introduction to Spring Framework

Goal Of Spring Framework

- The Spring Framework Mission Statement
  - J2EE should be easier to use
  - It's best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero
  - JavaBeans offer a great way of configuring applications
  - OO design is more important than any implementation technology, such as J2EE
  - Checked exceptions are overused. A framework should not force to catch

What is Spring Framework? (1)

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

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?

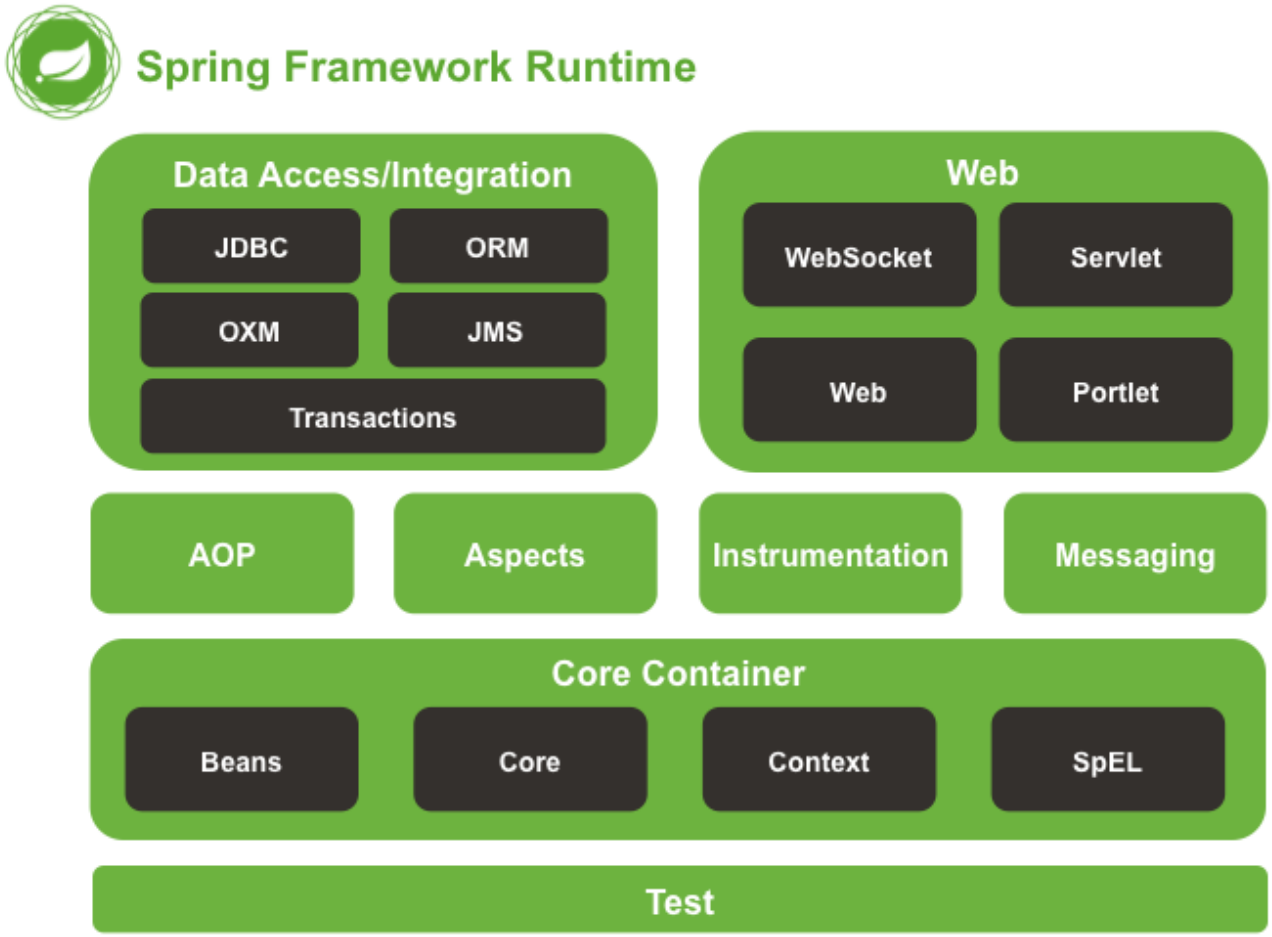
- Conversion of checked exceptions to unchecked
- Extremely modular and flexible
- Well designed
  - Easy to extend
  - Many reusable classes

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 Modules

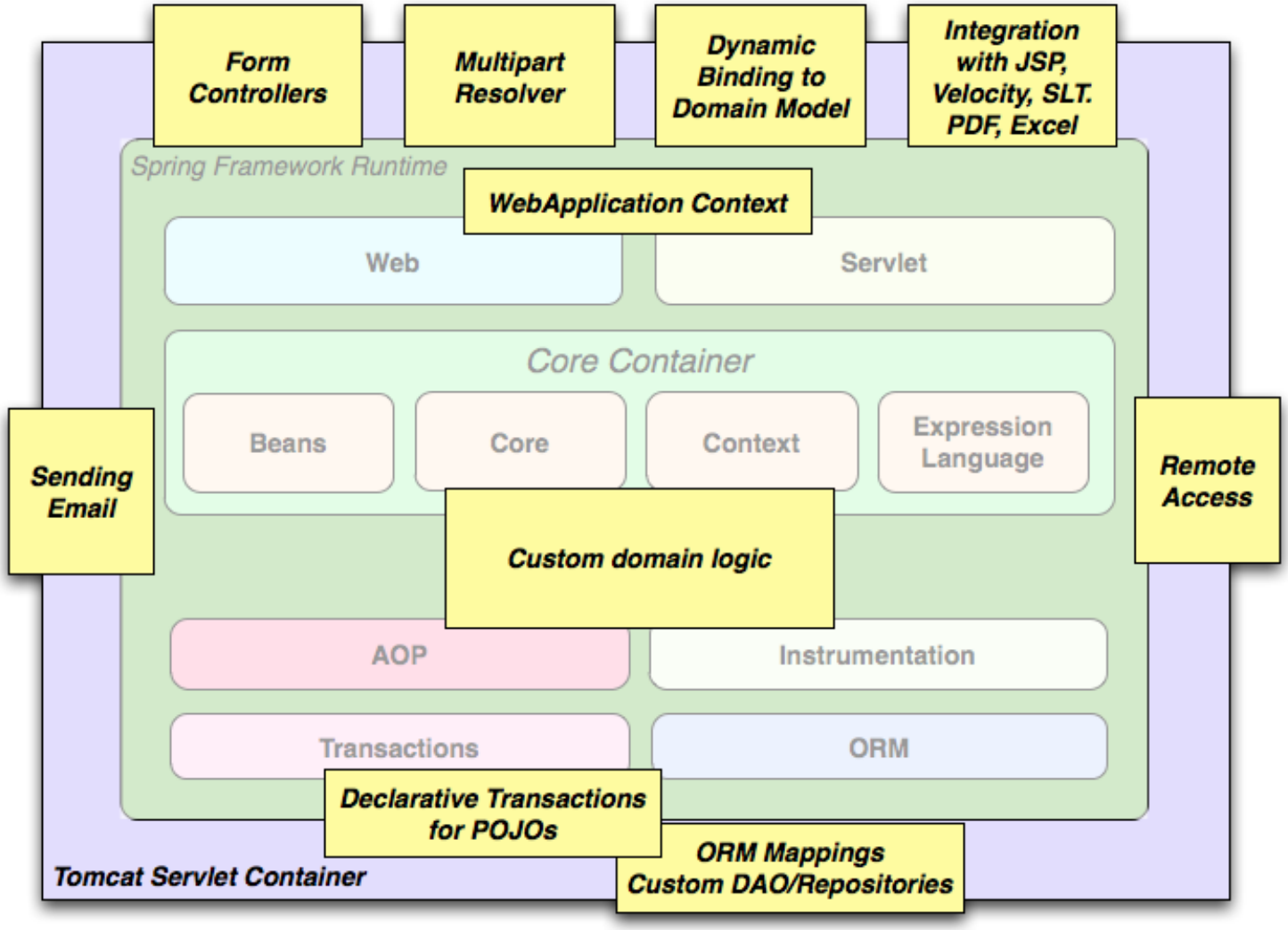


Usage Scenarios

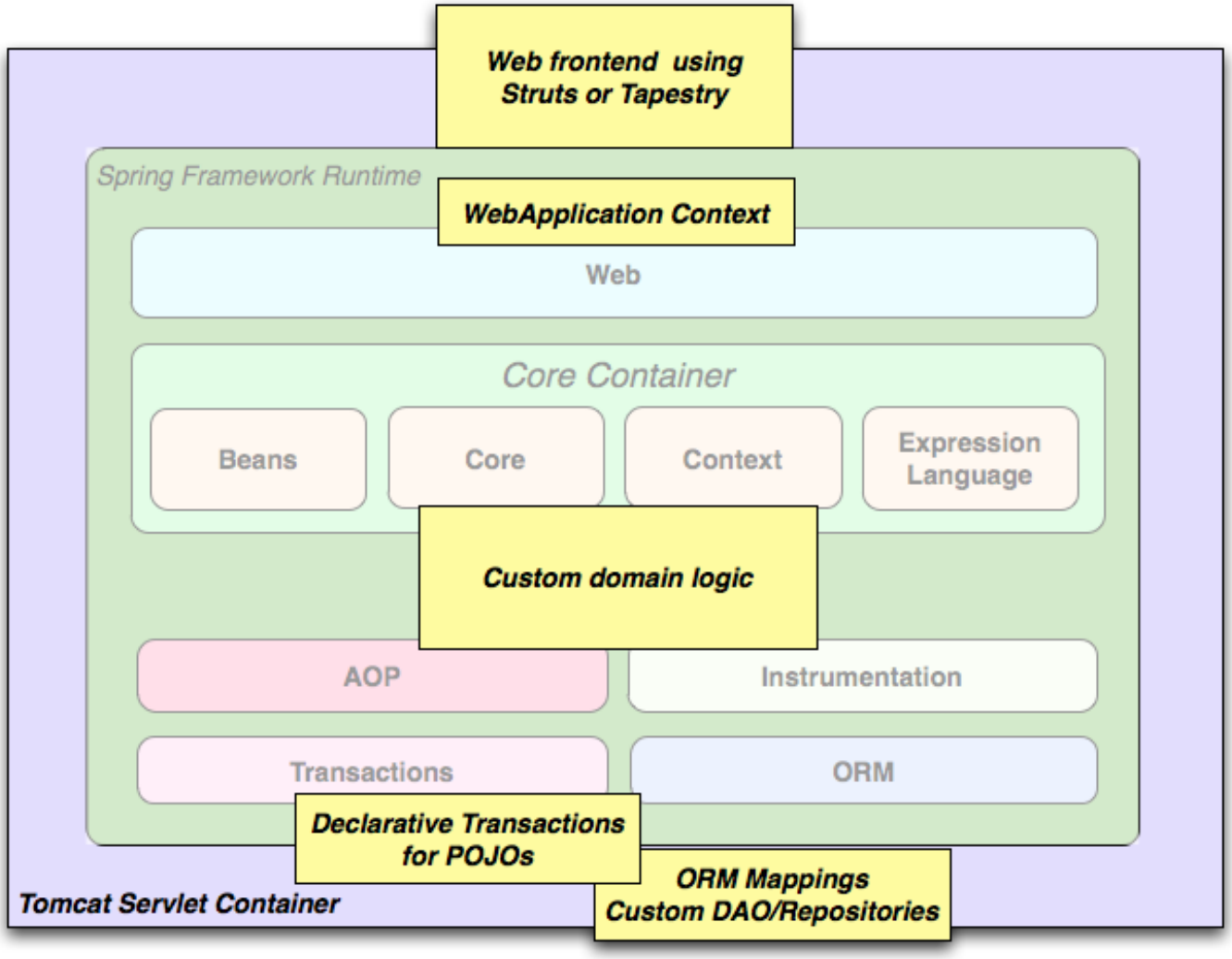
Usage Scenarios

- 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

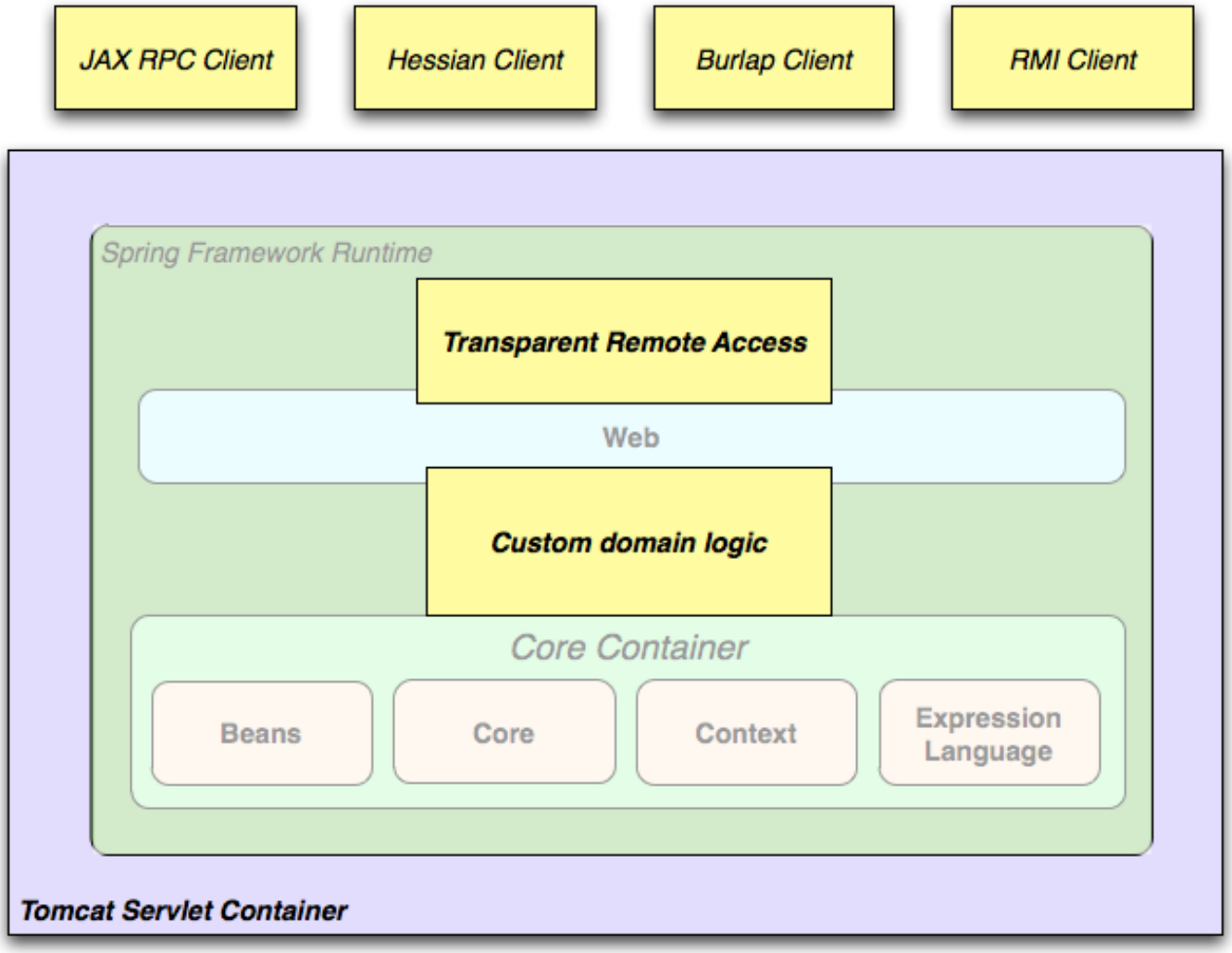
Typical Full-fledged Spring Web Application



Spring Middle-tier Using 3rd party Web Framework



Remoting Usage Scenario



The IOC Container and Dependency Injection

Dependency Injection (DI): Basic concept

Spring Dependency Injection

- A kind of Inversion of Control (IoC)
- “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

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

Beans and Containers

Beans

- In Spring, those objects that form the backbone of your application and that are managed by the Spring IoC *container* are referred to as *beans*.
- A bean is simply an object that is instantiated, assembled and otherwise managed by a Spring IoC container
  - there is nothing special about a bean (it is in all other respects one of probably many objects in your application).
- These beans, and the *dependencies* between them, are reflected in the *configuration metadata* used by a container

The IoC container

- The org.springframework.beans and org.springframework.context packages provide the basis for the Spring Framework's IoC container.
- The BeanFactory interface provides an advanced configuration mechanism capable of managing objects of any nature.
- The ApplicationContext interface builds on top of the BeanFactory (it is a sub-interface) and adds other functionality such as
  - easier integration with Spring's AOP features
  - message resource handling (for use in internationalization),
  - event propagation, and application-layer specific contexts such as the WebApplicationContext for use in web applications.

Important Application Contexts

- ClassPathXmlApplicationContext
- WebApplicationContext
- FileSystemApplicationContext
- AnnotationConfigApplicationContext

Configuration metadata

The Spring IoC container consumes some form of configuration metadata This configuration metadata is nothing more than how you inform the Spring container as to how to "instantiate, configure, and assemble [the objects in your application]". **This configuration metadata is typically supplied in a simple and intuitive XML format.**

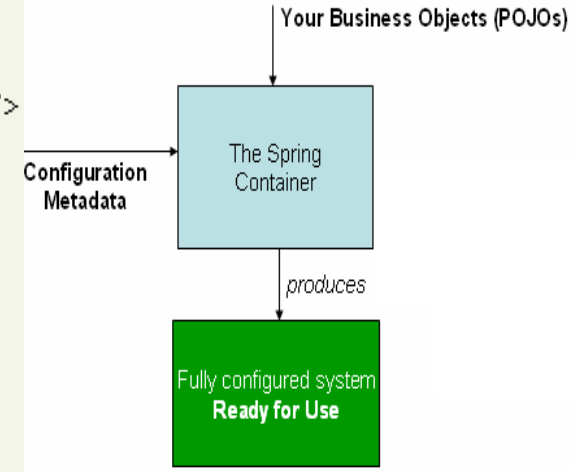
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```



Instantiating a container

- Instantiating a Spring IoC container is straightforward.

```
ApplicationContext ctx=

new ClassPathXMLApplicationCOntext("context.xml");
```

Composing XML-based configuration metadata

- It can often be useful to split up container definitions into multiple XML files.

```
<beans>

    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>

</beans>
```

Wiring a Bean

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

What is Wiring?

- The act of creating associations between application components is referred to as wiring
- There are many ways to wire a bean but common approach is via XML

Wiring example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="greetBean“       class="GreetingServiceImpl">
      <property name="greeting">
      <value>Hello friends of Spring</value>
      </property>
    </bean>
 </beans>
```

Wiring the beans

- Prototype and Singleton beans
  - all spring beans are singleton

```
<bean id =“myBean” class=“com.jp.TestBean”
                                        scope=“singletone”/>
```

Scope attribute has values:

1. Singleton
2. Prototype
3. Request
4. Session
5. Application
6. WebSocket

Dependency Injection: Autowiring

Auto Wiring

- So far we wired beans explicitly using <property> tag
- Spring can also do Wiring automatically

```
<bean id="foo" class="com.jp.spring.Foo“
        autowire= "autowire type"/>
```

Autowiring Properties

- Beans may be auto-wired (rather than using <ref>)
  - Per-bean attribute *autowire*
  - *Explicit settings override*
- *autowire="byName"*
  - Bean identifier matches property name
- *autowire="byType"*
  - Type matches other defined bean
- *autowire="constructor"*
  - Match constructor argument types
- *autowire="autodetect"*
  - Attempt by constructor, otherwise "type"
- Autowire="no"
  - no autowire is allowed

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”/>
```

Spring Framework Annotations

Limited to Spring Core Configuration

Annotations To Define a Bean

- Type Level Annotations (Used before Class definition)
  - @Component
  - @Service
  - @Repository
  - @Controller
  - @Configuration
- Method Level Annotations (used before method)
  - @Bean

Examples (Type level)

```
@Repository
public class JdbcDaoImpl{


}
```

```
@Service
public class EmpService{


}
```

```
@Component
public class PrintAspect{


}
```

```
@Controller
public class LoginController{


}
```

Examples (method level)

```
@Configuration
public class JdbcConfig {

@Bean
public DriverManagerDataSource dataSource() {

        //your code goes here
        return ds;

}
```

How Dependency is Injected

- Use @Autowired at the injection point to inject Dependency
- @Autowired can be used at following levels
  - Type
  - Field
  - Method

Example @Autowired

```
@Service
public class HrService {

@Autowired
private EmpDao dao;

public String registerEmp(int id, String name, String city, double salary) {

String resp=dao.save(new Emp(id, name, city, salary));
return resp;
        }
}
```

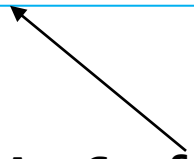
Loading Annotated Beans

- Every Spring Application has at least one Configuration class
  - annotated with @Configuration
- Spring Framework uses @ComponentScan to locate annotated beans in the project
- You are generally advised to keep the main configuration class in the base package.

An Example

```
@Configuration
@ComponentScan(basePackages="com.demo.spring")
public class AppConfig {

@Bean
public EmpDaoJPAImpl jpaBean() {
return new EmpDaoJPAImpl();
}
}
```



```
ApplicationContext ctx
=new AnnotationConfigApplicationContext(AppConfig.class);
```

The Spring Container will be created from the **AppConfig** Class and spring will scan all the sub packages of "com.demo.spring" including the base package

Demo

We will See some Demonstration on Spring Annotations