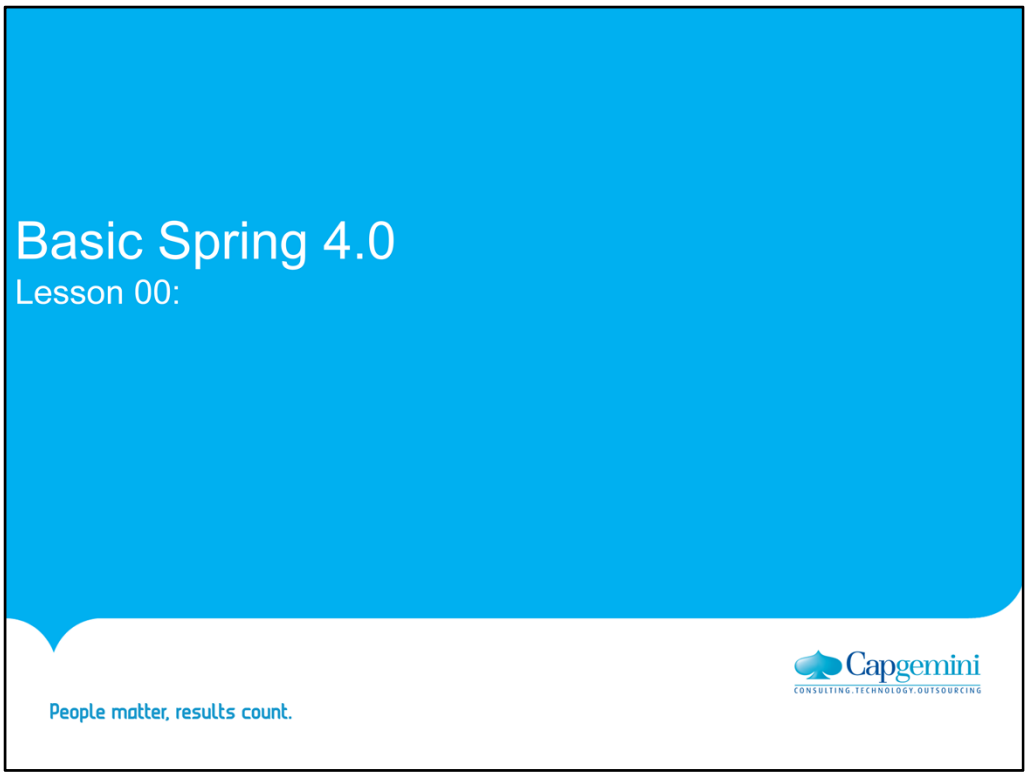


Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
21-Jul-2006	Ver1.0	1.2.6	Shrilata Tavargeri	NA
Jan-2009	Ver2.0	2.5	Shrilata Tavargeri	Changed material with new changes introduced in ver 2.5 and inputs from BU
Aug-2011	Ver 3.0	3.0	Shrilata Tavargeri	Changed material with new changes introduced in ver 3.0 and inputs from BU
June 2013	Ver 4.0	3.0	Mohan Chinnaiah	Revamped materials according to new requirements
May 2015	Ver 5.0	4.0	Rathnajothi Perumalsamy	Changed material with new changes introduced in ver 4.0
June- 2016	Ver 6.0	4.0	Vinod Satpute Yukti Valecha Tanmaya Acharya	Modified as per Toc for ELTP

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2

Keep this as a hidden slide.
Note to co-ordinators: Not to be printed for the class book.

Course Goals and Non Goals

- Course Goals
  - Understand the benefits of using Spring
  - Understand the principles of IoC and AOP
  - Be able to use AOP to handle cross-cutting concerns
  - Connect business objects to persistent stores using Spring's DAO modules
  - Use the Spring MVC web framework to develop flexible web applications
  - Introduction to Spring Testing
- Course Non Goals
  - Design patterns, Spring Integration with different technologies

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 3

Pre-requisites

- Core Java , Java 8 features and JDBC
- XML, DBMS/SQL
- Servlets, JSP
- Concepts of MVC, Design patterns

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 4

Intended Audience

- All Java application developers especially Enterprise Java Programmers
- Software designers

Day Wise Schedule

- Day 1
  - Lesson 1: Introduction to Spring Platform and environment
  - Lesson 2: Introduction to Spring Framework, IoC
- Day 2
  - Lesson 2: Introduction to Spring Framework, IoC (Contd..)
  - Lesson 3 : SpEL (Spring Expression Language)
- Day 3
  - Lesson 4: Spring MVC framework
- Day 4
  - Lesson 5: Spring JPA Integration
- Day 5
  - Lesson 6: AOP (Aspect Oriented Programming)

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 6

Table of Contents

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 7

Table of Contents

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Table of Contents

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 9

References

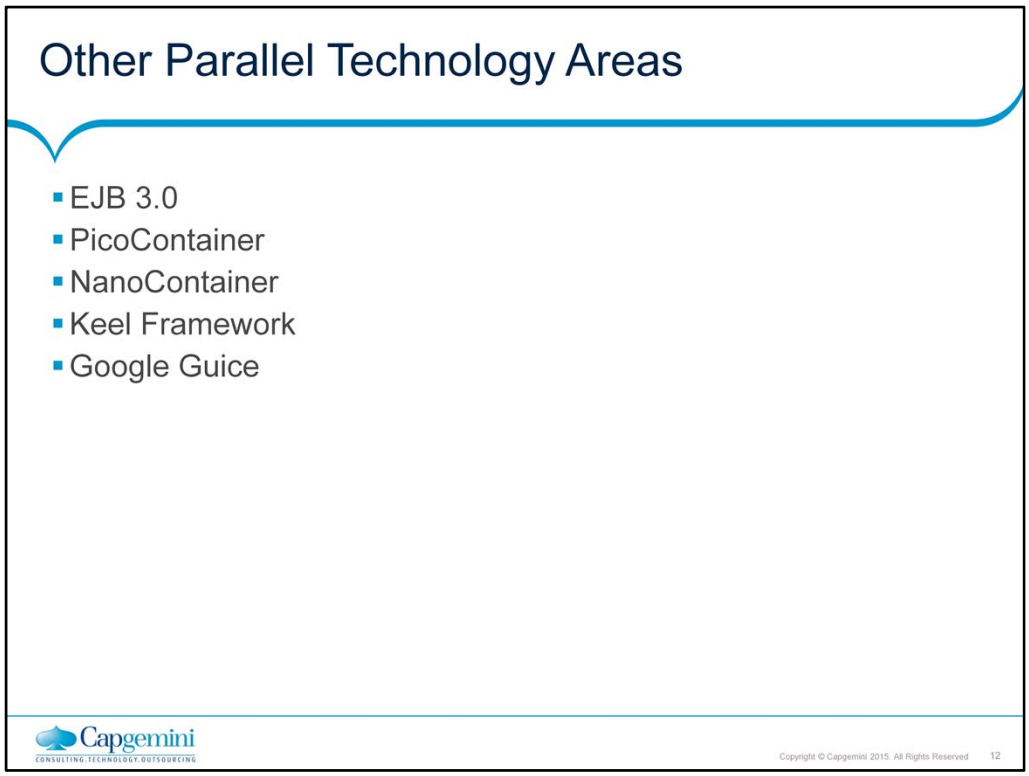
- Spring in Action, Fourth Edition, Manning publications by Craig Walls
- Spring-framework-reference.pdf from SpringSource (this is available in the downloaded Spring software)

Software required

- JDK version 1.8 + with help, Netscape or IE
- MS-Access/Connectivity to Oracle database
- WildFly
- Eclipse Luna
- Spring 4.0 API with docs

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 11



PicoContainer: is an exceptionally small DI (Dependency Injection) container that allows to use DI for your application without introducing any dependencies other than PicoContainer itself
NanoContainer: is an extension to PicoContainer fro managing trees of individual PicoContainer containers.
Keel Framework: is more of a metaframework, in that most of its abilities come from other frameworks that are all brought together under one roof.
Google Guice: focuses purely on DI.

Basic Spring 4.0

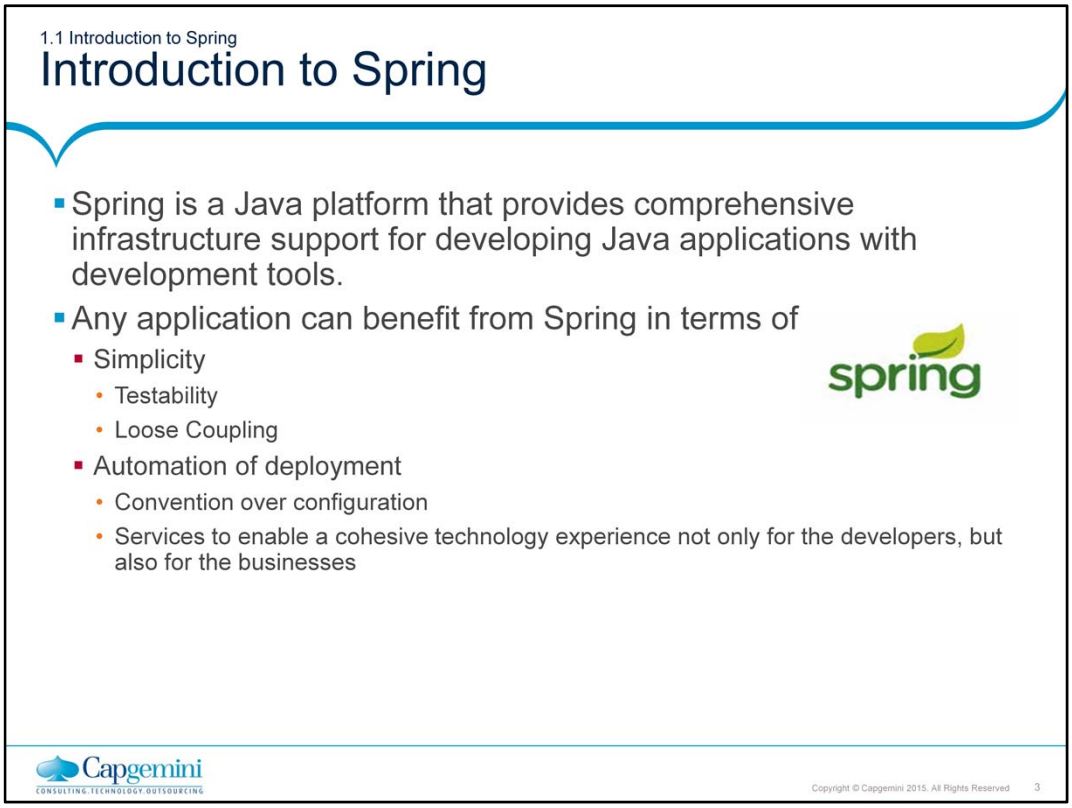
Lesson 1: Introduction to Spring Platform and environment

Lesson Objectives

- In this lesson , you will learn about
  - Introduction to Spring
  - Spring Projects at a glance
  - Spring IO Platform
    - Spring Framework
    - Spring Boot

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2



What is spring?
Spring is a Java platform that provides comprehensive infrastructure support for developing Java applications with development tools.
Any java application can benefit from Spring in terms of

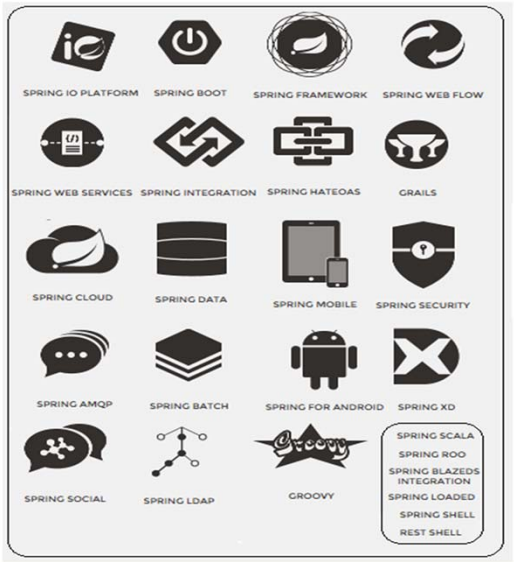
- Automation of deployment
- Convention over configuration
- Testing is simpler
- Services to enable a cohesive technology experience not only for the developers, but also for the businesses

Addresses the complexity of enterprise application development

1.2 Spring projects at a glance

Spring projects at a glance

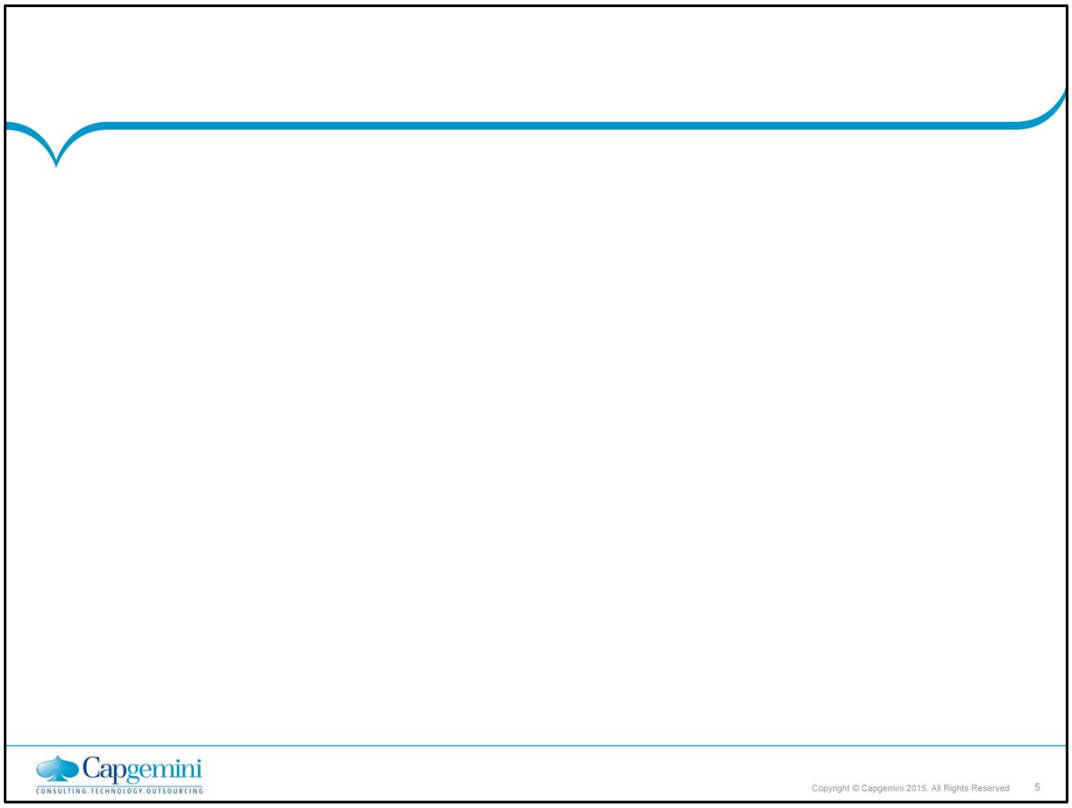
- Spring is modular by design
- Spring has many projects which helps us to build modern applications with any infrastructure needs such as
  - Simple Configuration
  - High Security
  - Connectivity to Big Data
  - Development of Web apps
  - Connectivity to cloud services
  - Integration with any framework.



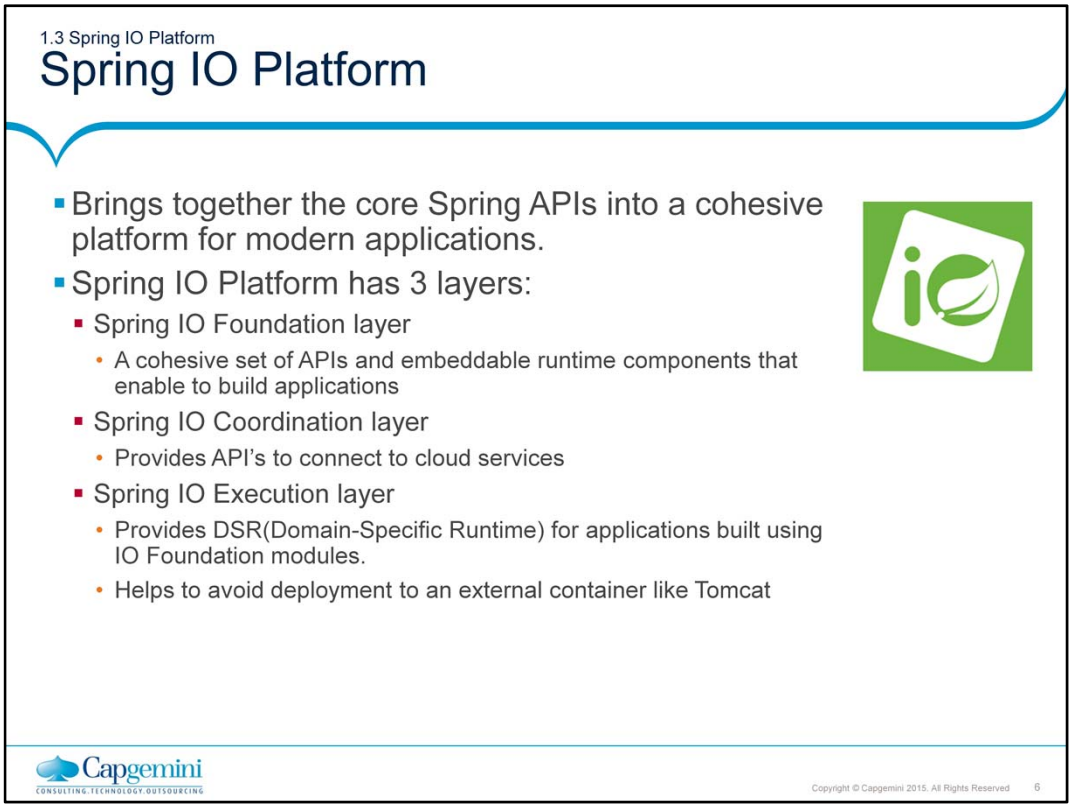
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 4

Spring IO platform includes Foundation Layer modules and Execution Layer domain-specific runtimes (DSRs)
Spring Boot favors convention over configuration and is designed to get you up and running as quickly
Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform
Spring Web Flow builds on Spring MVC and allows implementing the "flows" of a web application
Spring Web Services (Spring-WS) is a product of the Spring community focused on creating document-driven Web services
Spring Integration extends the Spring programming model to support the well-known Enterprise Integration Patterns
Spring eXtreem Data : The project‘s goal is to simplify the development of big data applications. the core Spring APIs into a cohesive and versioned foundational platform for modern applications.



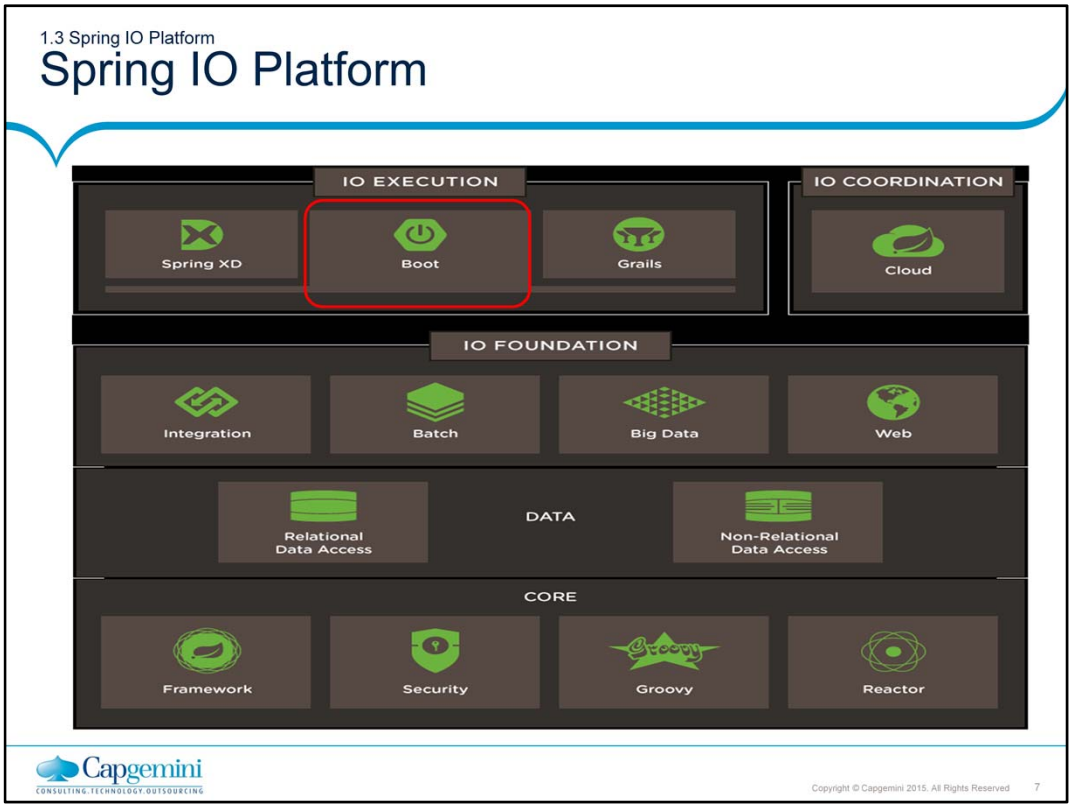
Spring Reactive : Reactive Streams, for handling live data (provide a standard for asynchronous stream processing with non-blocking back pressure on the JVM)
Hypermedia As The Engine Of application State : REST client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia, in contrast with SOA.
Spring HATEOS :link creation and representation assembly
Microservices (wiki) : Software architecture design pattern, in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are small, highly decoupled and focus on doing a small task.
Spring Boot : It's a new framework designed to simplify the bootstrapping and development of a new Spring application with opinionated approach to configuration, freeing developers from the need to define boilerplate configuration.
Spring Cloud : Provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).



Spring IO Platform brings together the core Spring APIs into a cohesive platform for modern applications.
For example, there are many existing applications designed based on the core Spring Framework, and customers may want these applications to be upgraded with features like adding an OAuth secured REST service, connect to cloud services, Moving data into Hadoop, bridging multiple data stores, etc.. In order to upgrade applications including all the services use Spring IO Platform modules.

Spring IO Platform is comprised of 3 layers:

- Spring IO Foundation layer
  - A cohesive set of APIs and embeddable runtime components that enable to build applications
- IO Coordination
  - Provides API's to connect to cloud services with Spring Cloud.
- IO Execution
  - Provides DSR(Domain-Specific Runtime) for applications built using IO Foundation modules.
  - Helps to avoid deployment to an external container like Tomcat
  - Spring IO execution includes three DSRs: Spring XD, Spring Boot, and Grails.



Spring IO Execution Layer:

Spring XD provides a powerful runtime and DSL for describing big data ingestion and analytics, export, and Hadoop workflow management.

Spring Boot reduces the effort needed to create production-ready, DevOps-friendly, XML-free Spring applications. Spring Boot dynamically wires up beans and settings and applies them to your application context

Grails provides a productive and stream-lined full-stack web framework by combining the power of the Spring IO Foundation components with a set of comprehensive Groovy-based DSLs.

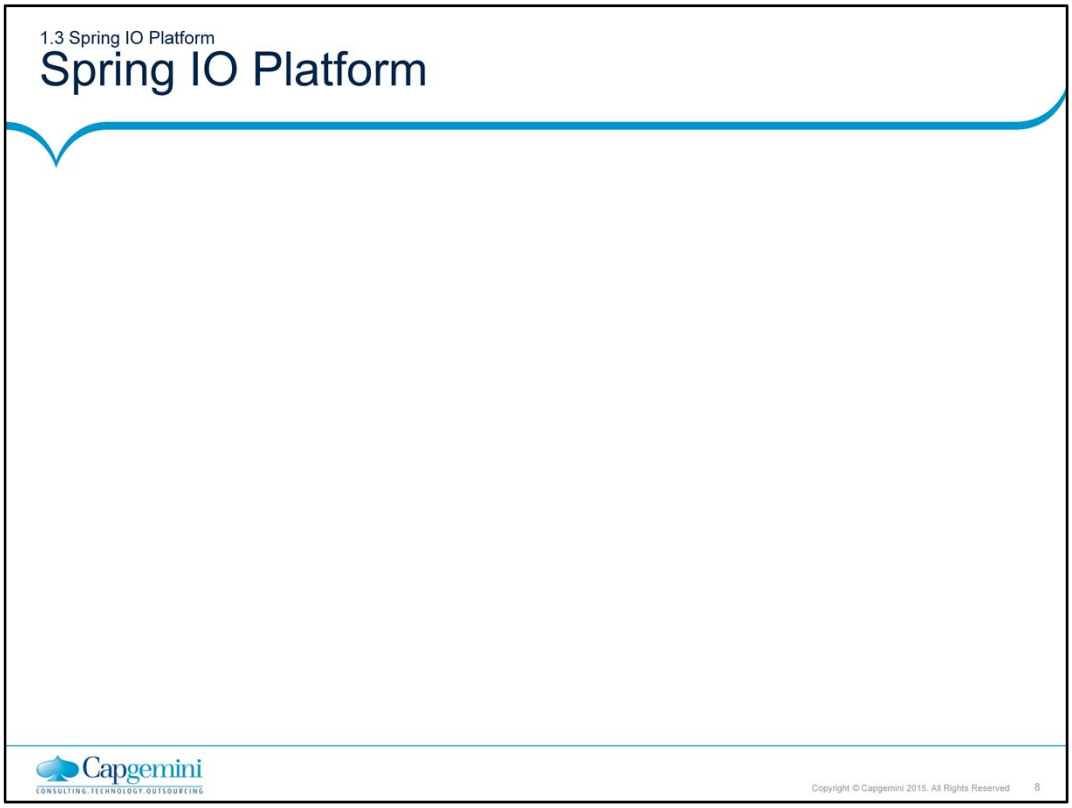
Spring IO Coordination Layer:

Spring Cloud is an open-source library with which an application can be connected with cloud environment. For example, Instead of creating data source object to connect with relational databases use Spring cloud which does all these work(like access and configure service connectors) by using cloud connector.

Spring IO Foundation Layer

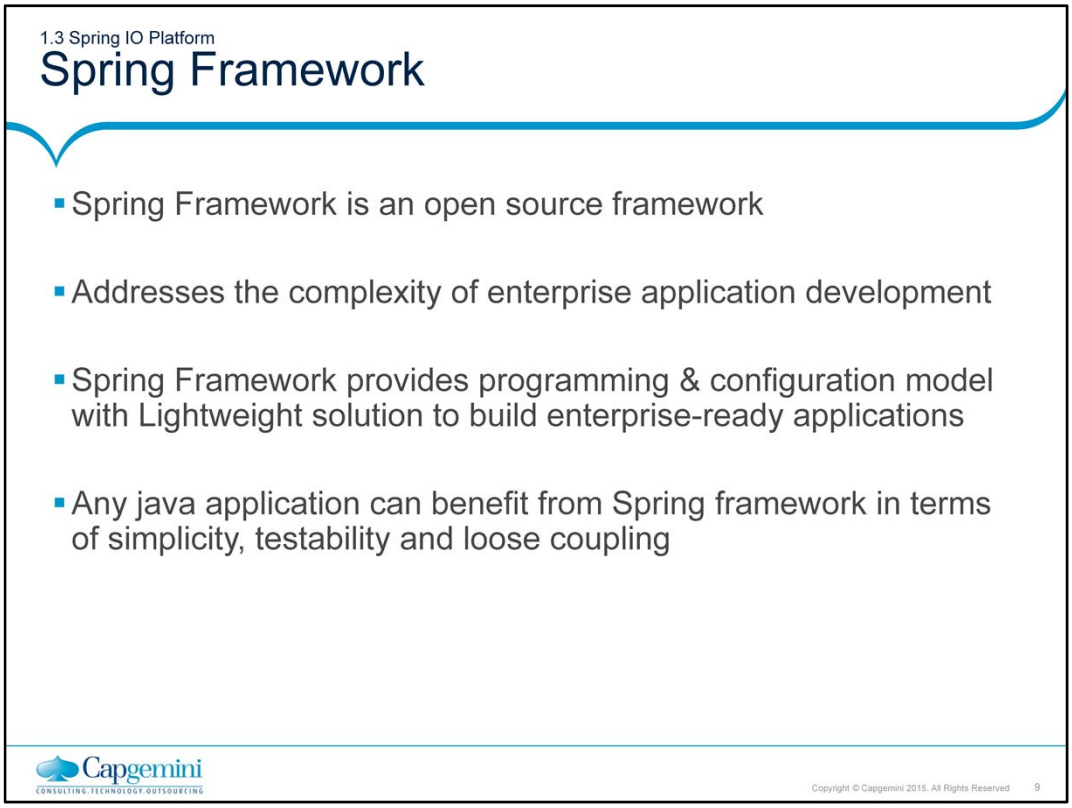
Spring Integration extends the Spring programming model to support the well-known Enterprise Integration Patterns.

Spring Batch is a lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems.



Spring IO Foundation Layer:

Spring Big Data is used to simplify the development of big data applications.
Spring Web builds on Spring MVC and allows implementing the "flows" of a web application.
Spring data makes it easy to use new data access technologies, such as non-relational databases, map-reduce frameworks, and cloud based data services.
Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications on any kind of deployment platform.
Spring security helps us to secure spring based applications since it is a powerful and highly customizable authentication and access-control framework.
Spring groovy can be used to integrate with groovy for building high-productivity dynamic application.
Spring Reactor is a Reactive Streams, for handling live data (provide a standard for asynchronous stream processing with non-blocking back pressure on the JVM)



Spring makes it easy to use POJO (Plain Old Java Objects) to achieve things that were previously only possible with EJBs. However, Spring's usefulness isn't restricted to server-side development. Any java application can benefit from Spring in terms of simplicity, testability and loose coupling.

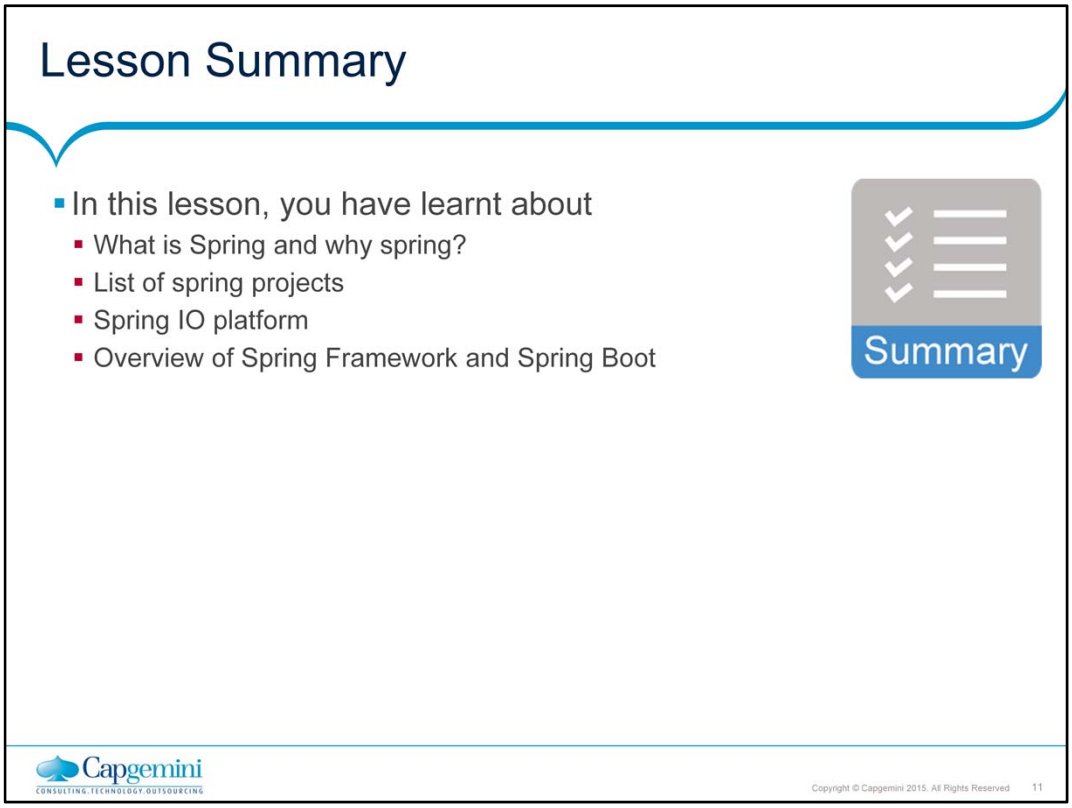
1.3 Spring IO Platform

Spring Boot

- Spring Boot ships with command line tool for executing spring applications
- Spring Boot dynamically wires up beans and settings and applies them to your application context.
- Advantages of using Spring Boot are:
  - No requirement for XML configuration
  - Annotation based configuration
  - Has embedded server
  - Reduces boiler plate code
  - Simplifies testing
  - Simplifies application maintenance
  - Reduces the size of build file

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 10



Thus we have seen that Spring is the most popular and comprehensive of the lightweight J2EE frameworks that have gained popularity since 2003.
We saw how Spring is designed to promote architectural good practice. A typical spring architecture will be based on programming to interfaces rather than classes.
We have seen what is Inversion of control and dependency injection.
We also saw Bean containers and lifecycle of beans in containers. We saw how to hook into the lifecycle of a bean and make it aware of the Spring environment.

Review Questions

- Question 1: Spring IO _____ layer provides API to connect to cloud services
  - Option 1: Foundation
  - Option 2: Coordination
  - Option 3: Execution
- Question 2: Spring Boot reduces the effort needed to create production-ready, DevOps-friendly, XML-free Spring applications.
  - Option 1: True
  - Option 2: false

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Basic Spring 4.0

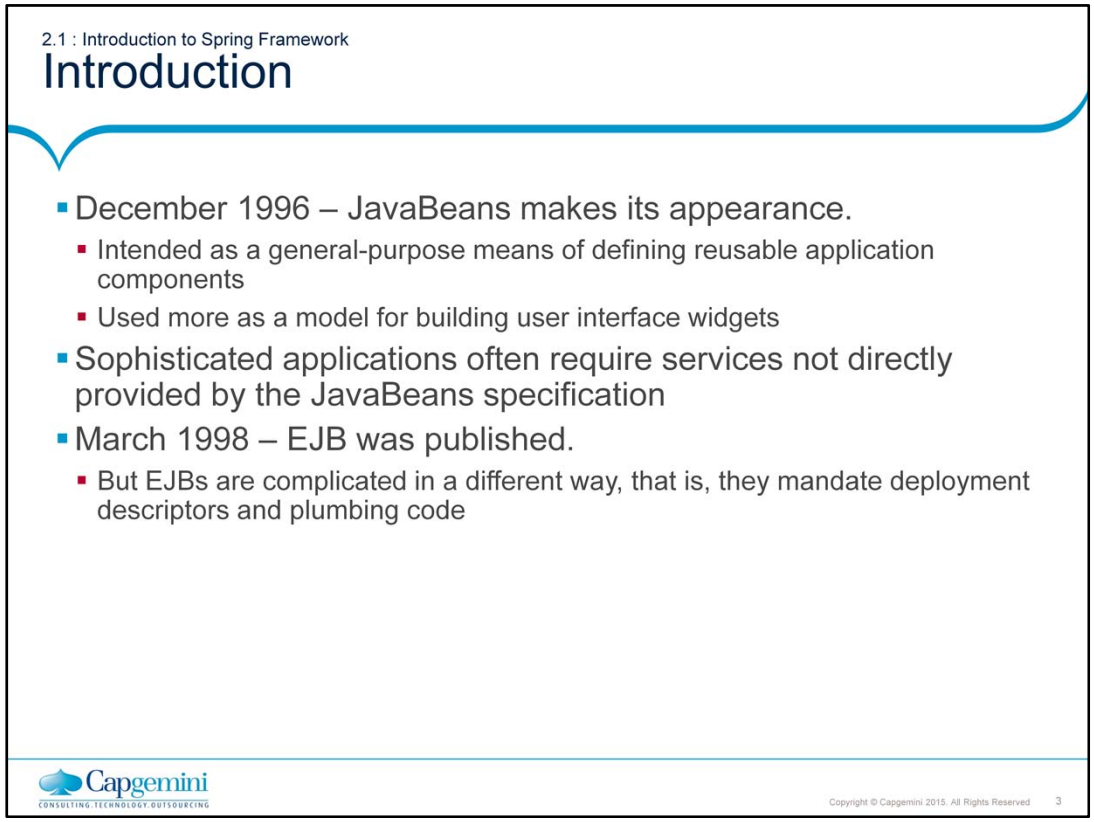
Lesson 2:Introduction to Spring Framework, IoC

Lesson Objectives

- Introduction to Spring Framework
  - Learn about the Spring Framework, its benefits and architecture
  - Learn about the IoC (Inversion of control) and how it allows wiring beans
  - Learns the types of bean factories and life-cycle of beans in these factories
  - Understand how to apply Annotations to Spring applications
- Injecting dependencies through setter and constructor injections
- Wiring Beans
- Bean containers
  - Life cycle of Beans in the factory container
  - BeanPostProcessors and BeanFactoryPostProcessors
- Annotation-based configuration

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

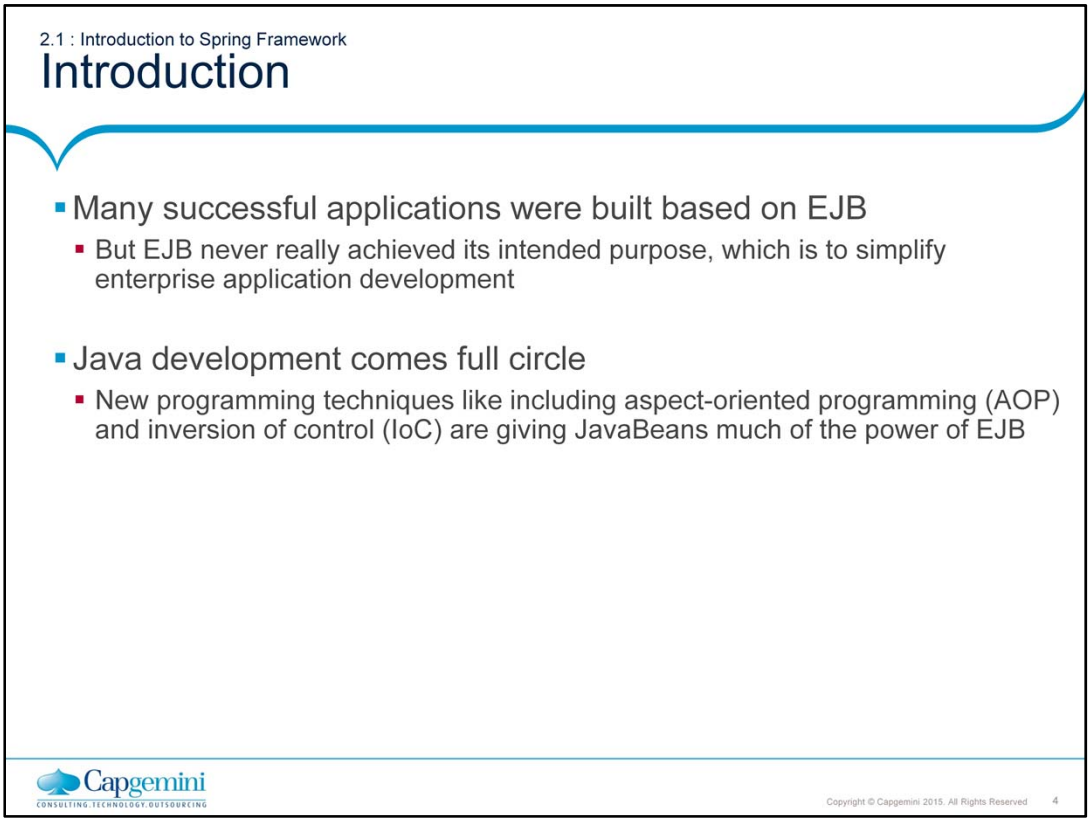
 2



It all started with a bean. In 1996, the Java programming language was still a young, exciting, up-coming platform. Many developers flocked to the language because they had seen how to create rich and dynamic web applications using applets. But they soon learned that there is more to this strange new language than juggling animated cartoon characters. Unlike any language before it, Java made it possible to write complex applications made up of discrete parts. They came for the applets, but stayed for the components.

It was in December of that year that Sun Microsystems published the Java-Beans 1.00-A specification. JavaBeans defined a software component model for Java. This specification defined a set of coding policies that enabled simple Java objects to be reusable and easily composed into more complex applications. Although JavaBeans were intended as a general-purpose means of defining reusable application components, they have been primarily used as a model for building user interface widgets. They seemed too simple to be capable of any “real” work; enterprise developers wanted more.

Sophisticated applications often require services that are not directly provided by the JavaBeans specification such as transaction support, security, and distributed computing. Therefore in March 1998, Sun published the 1.0 version of the Enterprise JavaBeans (EJB) specification. This specification extended the notion of Java components to the server side, providing the much-needed enterprise services, but failed to continue the simplicity of the original JavaBeans specification. In fact, except in name, EJB bears very little resemblance to the original Javabeans specification.

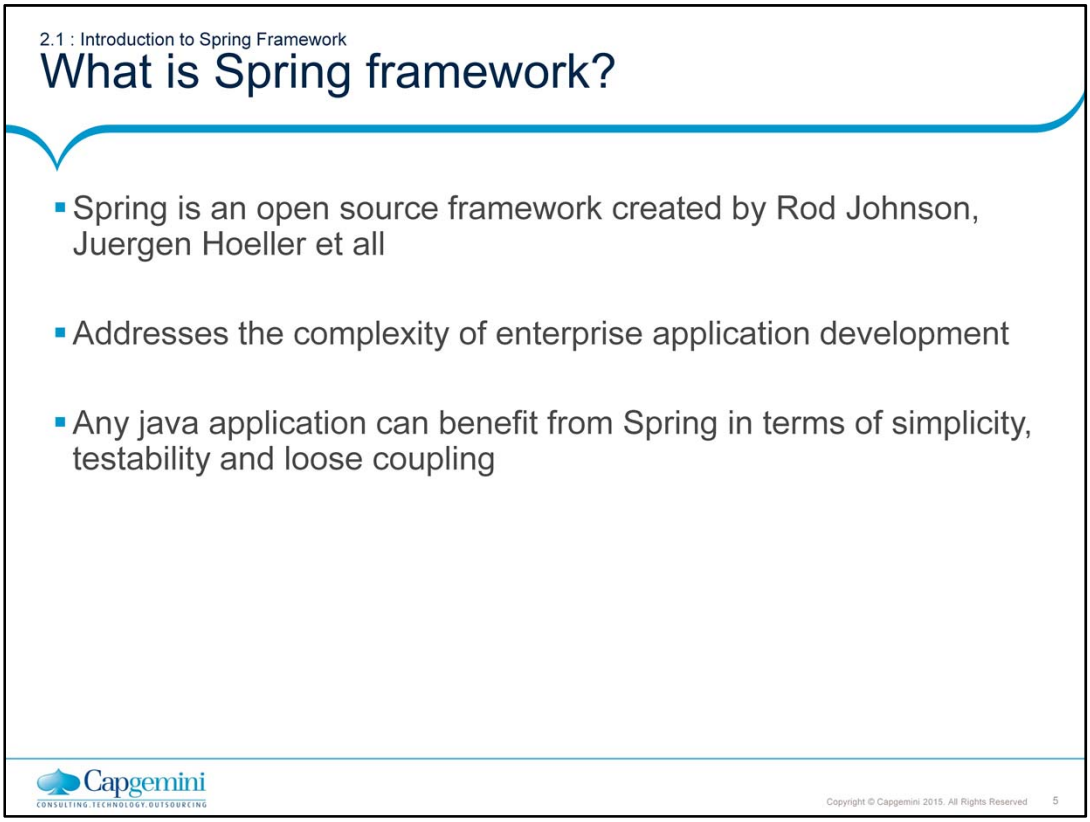


Despite the fact that many successful applications have been built based on EJB, it never really achieved its intended purpose: to simplify enterprise application development. It is true that EJB's declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. But EJBs are complicated in a different way by mandating deployment descriptors and plumbing code (home and remote/local interfaces). As a result, its popularity has started to wane in recent years, leaving many developers looking for an easier way.
Now Java development is coming full circle. New programming techniques, including aspect-oriented programming (AOP) and inversion of control (IoC), are giving JavaBeans much of the power of EJB. These techniques furnish JavaBeans with a declarative programming model reminiscent of EJB, but without all of EJB's complexity. No longer must you resort to writing an unwieldy EJB component when a simple JavaBean will suffice.
And that's where Spring steps into the picture.

> In all fairness, the latest EJB specification (EJB 3) has evolved to promote POJO-based programming model and is simpler than its predecessors.

Quick history :

Spring Framework project founded in Feb 2003
Release 1.0 in Mar 2004
Release 1.2 in May 2005
Release 2.0 in Oct 2006
Release 2.5 in Nov 2007
Release 3.0 in Dec 2009
Release 4.0 in Dec 2013



Spring makes it easy to use POJO (Plain Old Java Objects) to achieve things that were previously only possible with EJBs. However, Spring's usefulness isn't restricted to server-side development. Any java application can benefit from Spring in terms of simplicity, testability and loose coupling.

2.1 : Introduction to Spring Framework

What is Spring framework?

- Spring is a lightweight inversion of control and aspect-oriented container framework
  - Lightweight: in terms of both size and overhead
  - Inversion of control: promotes loose coupling
  - Aspect-oriented: enables cohesive development by separating application business logic from system services
  - Container: contains and manages the life cycle and configuration of application objects
  - Framework: possible to configure and compose complex applications from simpler components

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 6

Put simply, Spring is a lightweight inversion of control and aspect-oriented container framework. Breaking this description down makes it simpler:
Lightweight: Spring is Lightweight in terms of both size and overhead. The entire Spring framework can be distributed in a single jar file of 2.5 MB approximately. Processing overhead required by Spring is negligible. Moreover, Spring is non-intrusive; objects in a Spring-enabled application typically have no dependencies on Spring-specific classes.

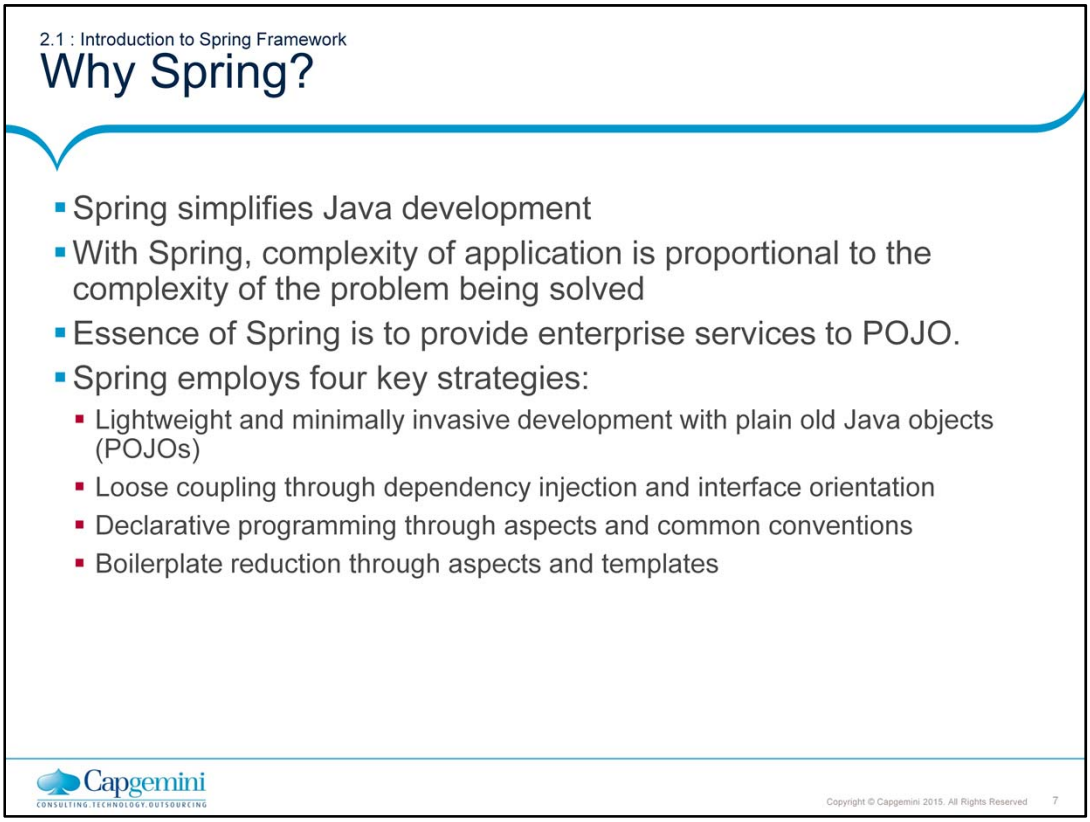
Inversion of control: Spring promotes loose coupling through a technique known as inversion of control (IoC) or also known popularly as Dependency Injection (DI). When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. IoC can be thought of as JNDI in reverse – instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.

Aspect-oriented : Spring comes with rich support for aspect-oriented programming that enables cohesive development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they are supposed to do – perform business logic – and nothing more. They are not responsible for other system concerns such as logging or transactional support.

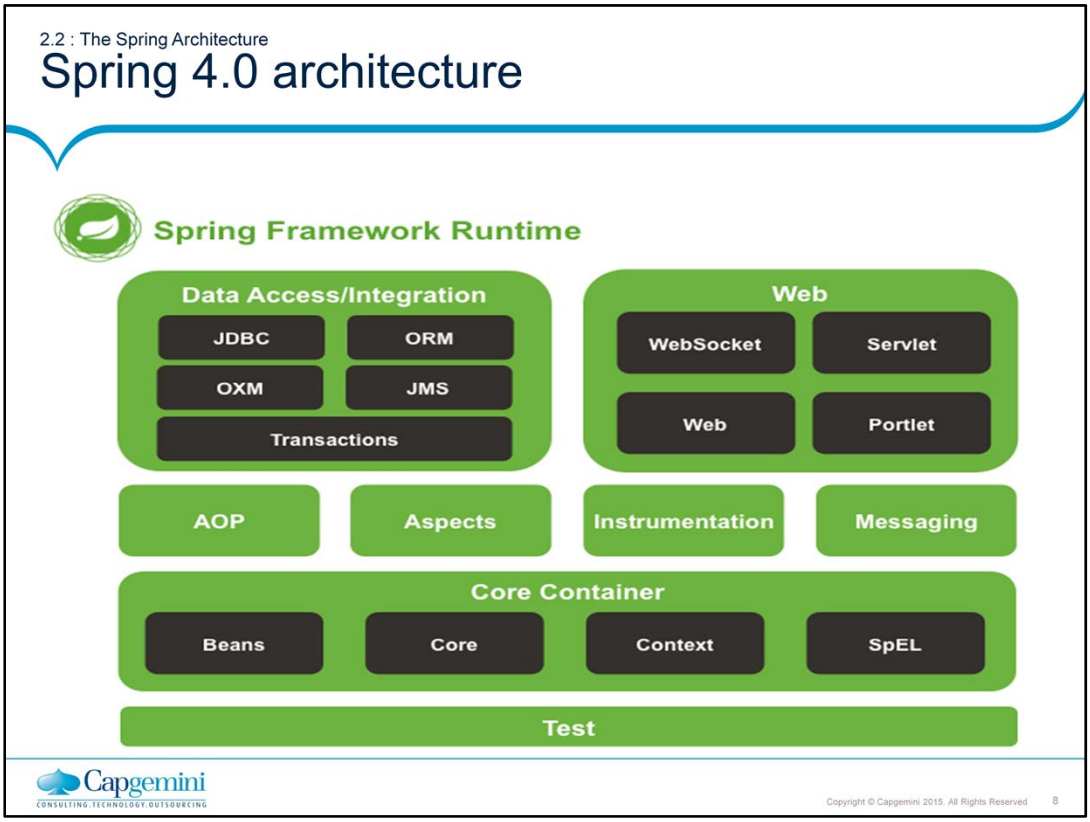
Container: Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects. You can configure how each of your beans should be created – either create one single instance of your bean or produce a new instance every time one is needed based on a configurable prototype – and how they should be associated with each other.

Framework: Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file. Spring also provides much infrastructure functionality (transaction management, persistence framework integration etc) leaving the development of application logic to user.

All these attributes of Spring enable you to write code that is cleaner, more manageable and easier to test.



To put it simply, Spring makes developing enterprise applications easier. The complexity of your application is proportional to the complexity of the problem being solved.
Some key Spring values are:
It's a non-invasive framework : Traditional frameworks force application code to be aware of the framework, implementing framework specific interfaces or extending framework-specific classes. Spring aims to minimize the dependence of application code on the framework.
Promotes pluggability : Spring encourages you to think of application objects as named services. Thus you can swap one service for another without affecting the rest of the application.
Aims to facilitate good programming practices, such as programming to interfaces: Using an IoC container like Spring reduces the complexity of coding to interfaces, rather than classes.
Spring applications are easy to test : Application objects will generally be POJOs and POJOs are easy to test; dependence on Spring APIs will normally be in the form of interfaces that are easy to stub or mock.
Does not reinvent the wheel : Spring does not introduce its own solution in areas such as O/R mapping where there are already good solutions. It also does not implement its own logging abstraction, connection pool, distributed transaction coordinator or other system services that are already well-served in other products or application servers. However Spring does make these existing solutions significantly easier to use.



The Spring Framework is composed of about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming),
Instrumentation, Messaging, and Test. When you download and unzip the Spring framework distribution, you'll find many different JAR files in the dist directory for every modules. When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But this modularity also gives you the freedom to choose the modules that suit your application.

Core Container : The Core Container consists of the spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.
The spring-core and spring-beans modules are the most fundamental part of the framework. It defines how beans are created, configured and managed. The BeanFactory (a sophisticated implementation of the factory pattern and a primary component of this module) applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code
The spring-context module builds on the solid base provided by the Core and Beans modules. The core modules bean factory makes Spring a container, but the context modules makes it a framework. The Context module inherits its features from spring-beans module and adds support for internationalization, event-propagation, resource-loading etc.
The spring-expression module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration : The *Data Access/Integration* layer consists of the spring--JDBC, spring--ORM, spring--OXM, spring--JMS and spring—tx modules.

- The spring-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 *spring-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.
- The spring-oxm module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, XStream etc.
- The spring-jms(Java Messaging Service) module contains features for producing and consuming messages.
- The spring-tx(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 spring-web, spring-webmvc, spring-websocket and spring-webmvc-portlet modules.

- The spring-w*eb* 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.
- The spring-webmvc module (also known as 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 spring-webmvc-portlet module (also known as 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 : AOP (Aspect Oriented Programming) enables behavior that would otherwise be scattered through different methods to be modularized in a single place.

- The spring-aop module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method interceptors and pointcut's to cleanly decouple code that implements functionality that should be separated.
- The separate spring-a*spects* module provides integration with AspectJ.
- The spring-i*nstrumentation* module provides class instrumentation support and classloader implementations to be used in certain application servers.

Messaging: Spring Framework 4 includes a spring-messaging module with key abstractions from the Spring Integration project such as Message, MessageChannel, MessageHandler, and others to serve as a foundation for messaging-based applications. The module also includes a set of annotations for mapping messages to methods, similar to the Spring MVC annotation based programming model.

Test : The *spring-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.

2.2.1 Dependency Injection

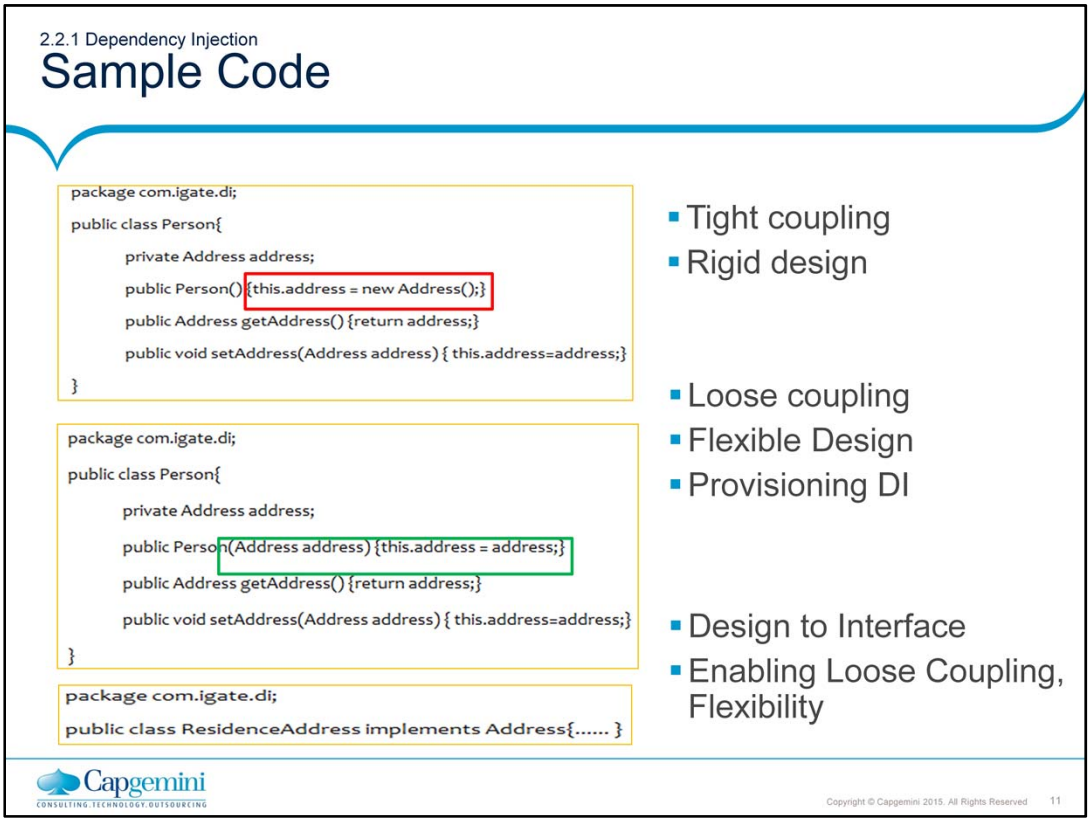
Dependency Injection

- Any enterprise application has objects that depend on each other
- Resolving the dependency is termed as 'Injecting Dependency' which facilitates loose coupling.
- Choosing the low level implementation to be injected into the reference of the interface in higher level layer, is termed as 'Inversion Of Control'

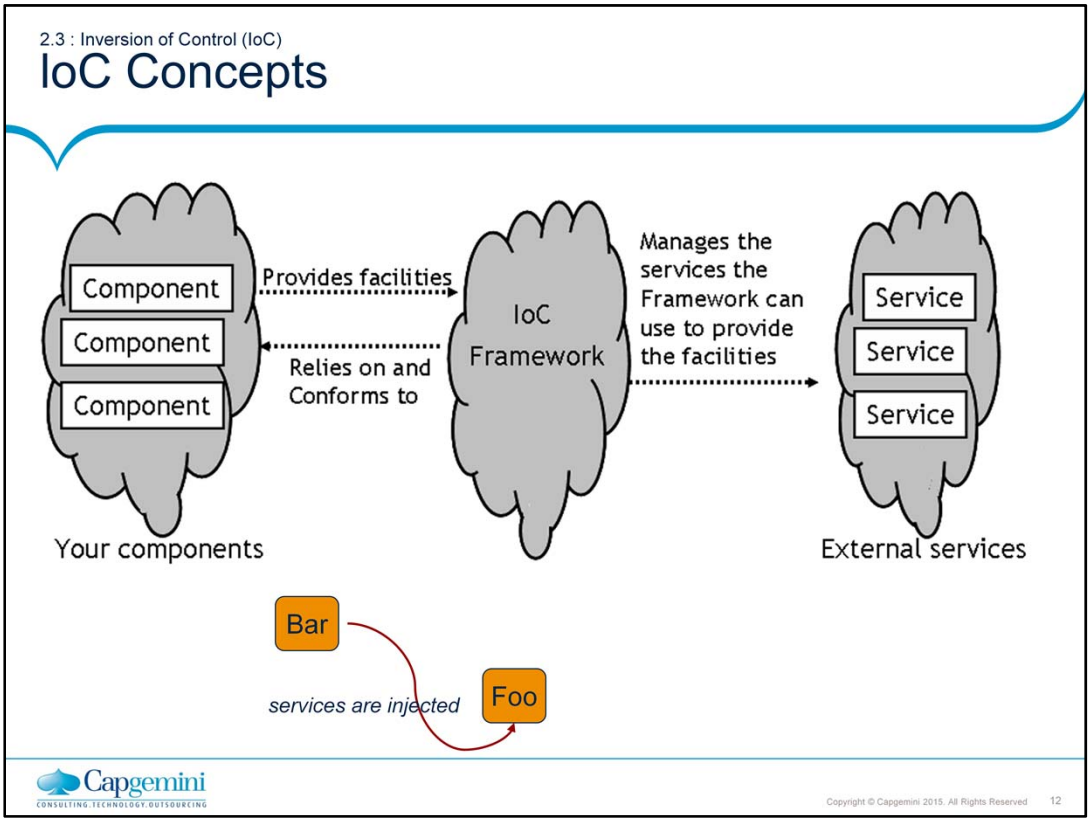
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 10

Dependency Injection
Any enterprise application has objects that depend on each other
Resolving the dependency is termed as 'Injecting Dependency' which facilitates loose coupling.
Choosing the low level implementation to be injected into the reference of the interface in higher level layer, is termed as 'Inversion Of Control'
Thus the choice of low level dependency has control on the quality of the service that will be provided, this is configurable/changeable making the framework highly flexible and modular
For an example, if a person need to have a cup of coffee then either he can prepare by himself using ingredients such as coffeebean, milk, sugar,.. Or he can raise request in vending machine, so that coffee will be prepared and automatically delivered to him.
Similarly, instead of creating object manually with the use of new operator, object creation will be taken care by container using DI.



In DI or IOC process objects define their dependencies, only through constructor arguments, arguments
to a factory method, or properties that are set on the object instance after it is constructed or returned
from a factory method. The container (the environment provided to the Spring Beans, for wiring) then injects those dependencies when it creates the spring bean, this process is an inverse of traditional approach, hence the name Inversion of Control (IoC), of the bean itself
controlling the instantiation or location of its dependencies by using direct construction of classes.



Understanding inversion of control: Inversion of control is at the heart of the Spring framework. As seen earlier, any non-trivial application is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to a highly coupled and hard-to-test code. Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system ie dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

Dependency injection is kind of an Inversion of Control pattern. The term dependency injection describes the process of providing (or injecting) a component with the dependencies it needs, in an IoC fashion. Dependency Injection proposes separating the implementation of an object and the construction of objects that depend on them. Dependency injection is a form of PUSH configuration; the container pushes dependencies into application objects at runtime. This is opposite to the traditional PULL configuration in which the app object pulls dependencies from its environment. In the figure, we have application objects offering external services. The application components depend on these external services. The job of coordinating the implementation and construction is left to the assembler code, which In this case would be the spring IoC framework.

2.3 : Inversion of Control (IoC)

IoC, Beans and BeanFactories

- Used to achieve loose coupling between several interacting components in an application.
- The IoC framework separates facilities that your components are dependent upon and provides the “glue” for connecting the components.
- DI it is specific type of IoC
- BeanFactory is the core of Spring’s DI container.
- In Spring, the term “bean” is used to refer to any component managed by the container.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 13

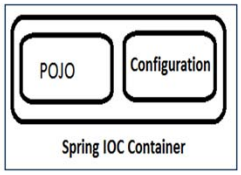
Loose coupling is one of the critical elements in object-oriented software development. It allows you to change the implementations of two related objects without affecting the other object. Strong coupling directly affects scalability of an application. Tightly coupled code is difficult to test, reuse and understand. On the other hand, completely uncoupled code doesn’t do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but it must be managed very carefully. A common technique used to reduce coupling is to hide implementation details behind interfaces so that actual implementation class can be swapped out without impacting the client class.
That is what IoC is all about: the responsibility of coordinating collaboration between dependent objects is transferred away from the objects themselves. This is where lightweight framework containers like Spring come into play.
IoC introduces the concept of a framework of components that in turn has many similarites to a J2EE container. The IoC framework seperates facilities that your components are dependent upon and provides the “glue” for connecting the components.
With Inversion of Control (IoC), you can achieve loose coupling between several interacting components in an application.

The core of Spring’s DI container is the BeanFactory. A bean factory is responsible for managing components and their dependencies. We shall see bean factories in detail later in this session. In Spring, the term “bean” is used to refer to any component managed by the container. Typically, beans adhere to Javabeans specification

2.3 : Inversion of Control (IoC)

Spring Beans and Configuration Metadata

- Spring Managed POJOs with business logic are termed as Spring Beans, Spring IOC container manages one or more beans
- Spring meta-data configuration can be …
- XML based
- Java based : Annotations
- Java based configurations are recommended practice since inclusion of Spring JavaConfig project
- Spring Beans and Configuration Metadata is combined and the Spring Framework's IoC container is created & initialized
- Spring IoC container is decoupled from configuration metadata format



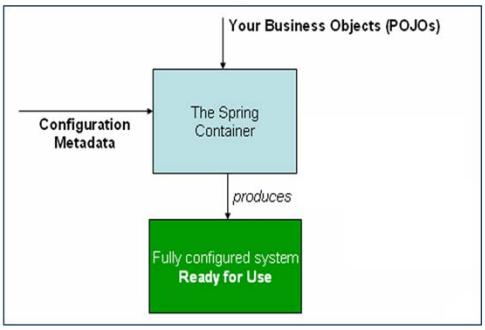
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 14

2.3 : Inversion of Control (IoC)

Spring IOC Container

- The Spring IOC container instantiates, configures, and assembles the beans by reading configuration metadata
- It composes the application and interdependencies between the objects
- At this stage, the application is fully configured and ready to use
- The Spring IOC Container manages the entire lifecycle of the Spring Beans

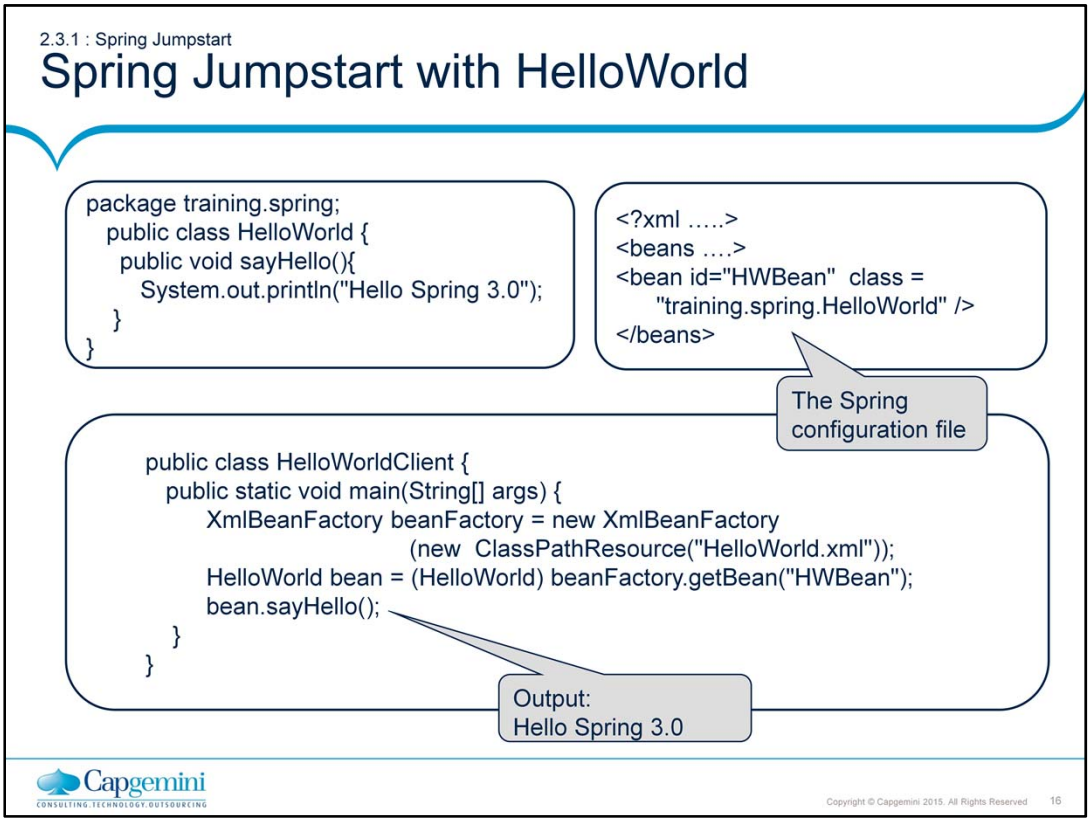


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 15

The Spring IOC container can manage any class you want it to manage; it is not limited to managing true JavaBeans
Spring container can also handle non-bean-style classes
Spring IoC container manages one or more beans
In the Spring IOC container, bean definitions are represented as BeanDefinition objects, with metadata:

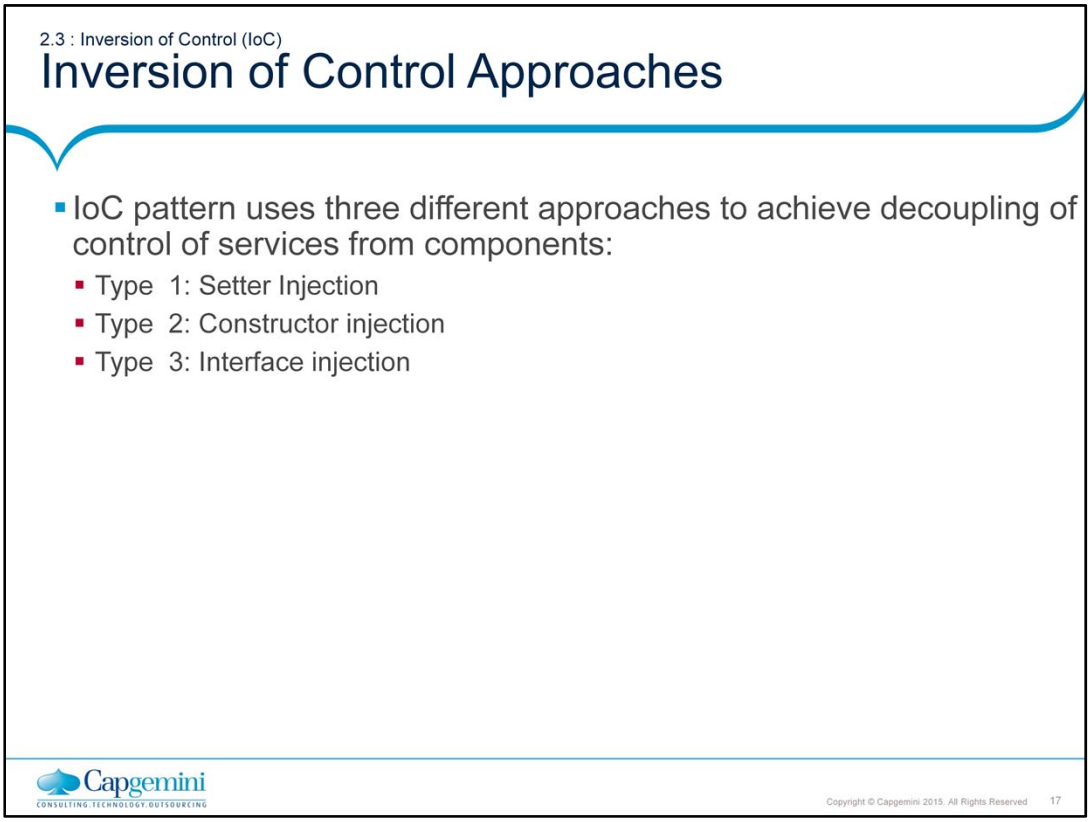
- A package-qualified class name
- Bean behavioural configuration elements
- References to other beans, collaborators or dependencies
- Configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool



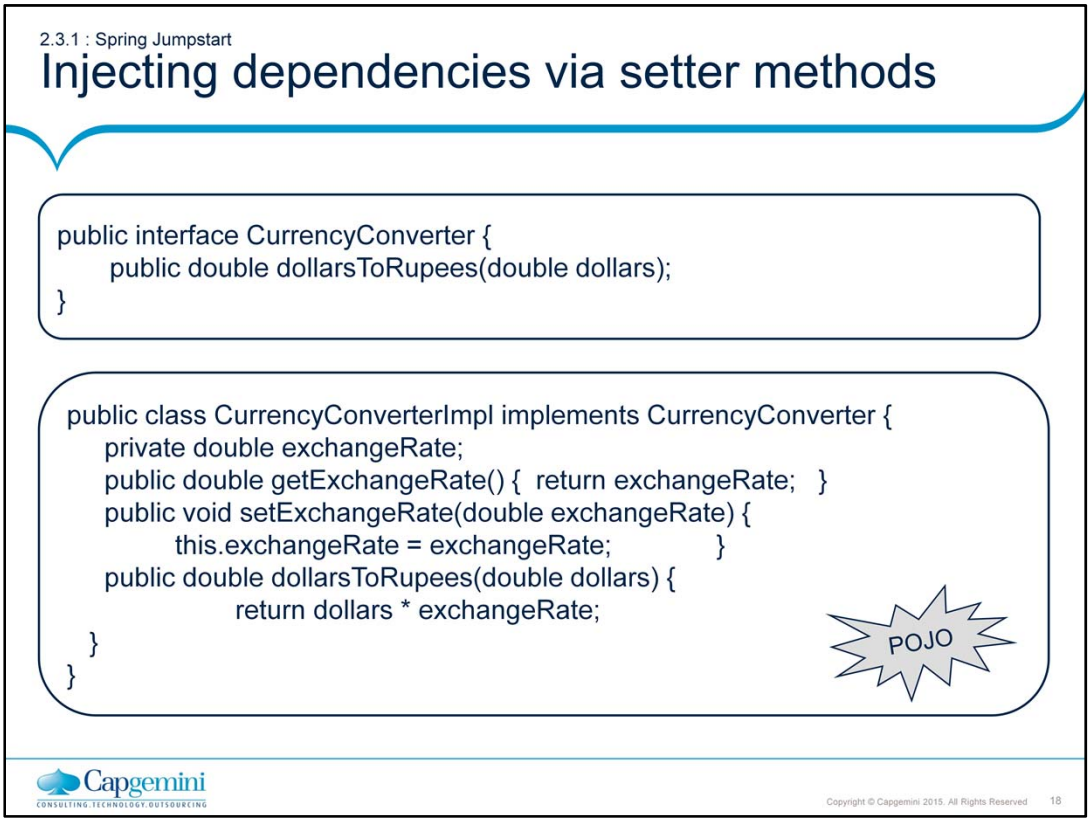
Dependency injection is the most basic thing that Spring does. We shall be covering this in detail later in the session. For now, let us see how Spring works with an example.
Spring-enabled applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application. Difference lies in how these classes are configured and introduced to each other. Typically a Spring application has an XML file that describes how to configure the classes, known as Spring configuration file.

Pls. refer to Appendix-A for a detailed explanation of converting a traditional Java application into a Spring-based application.

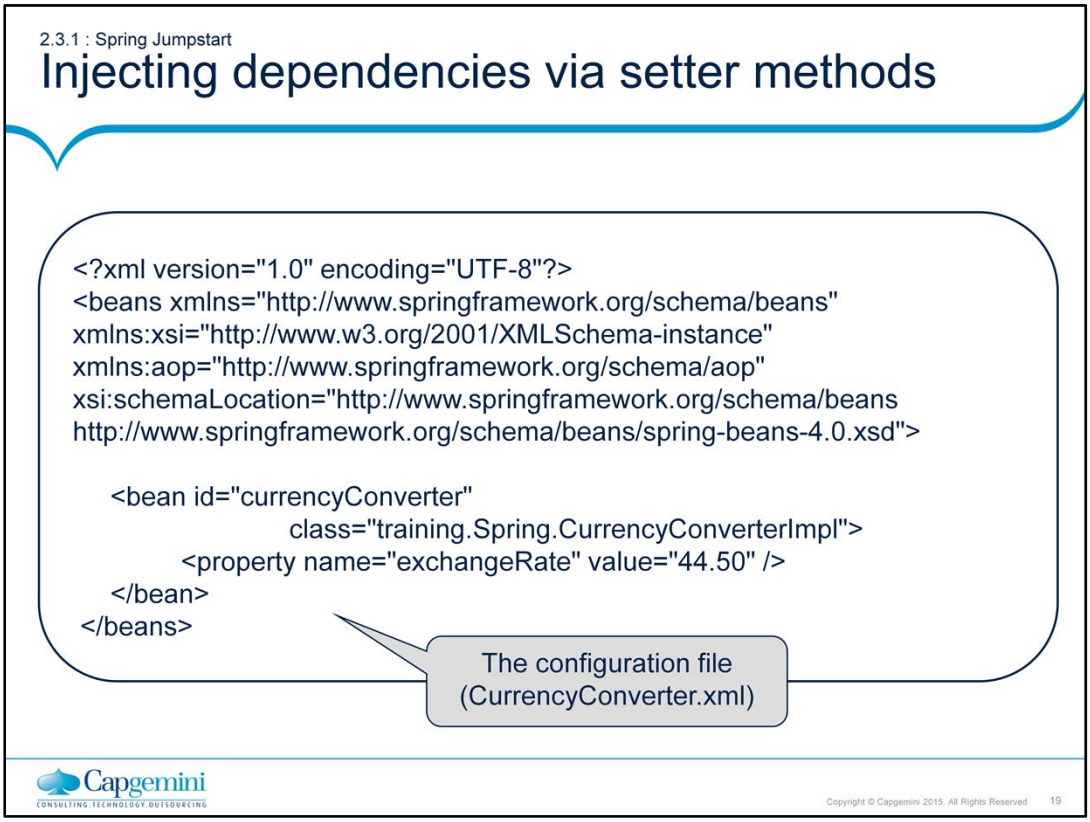
Look in slide above for a typical Hello World application using Spring Framework. Detailed explanation for this code is given in subsequent demos.



The IoC pattern uses three different approaches to achieve decoupling of control of services from your components:
Type 1 : Interface injection: This is how most J2EE worked. Components are explicitly conformed to a set of interfaces with associated configuration metadata, in order to allow framework to manage them correctly.
Type 2 : Setter Injection: External metadata is used to configure how components can interact. Our first example used this approach, by using a Springconfig.xml file.
Type 3 : Constructor injection: Components are registered with the framework, including the parameters to be used when the components are constructed, and the framework provides instances of the component with all the specified facilities applied. Our last example used this approach.
There is a fourth approach called “Lookup-method injection”. This has been covered in detail in Appendix-B.



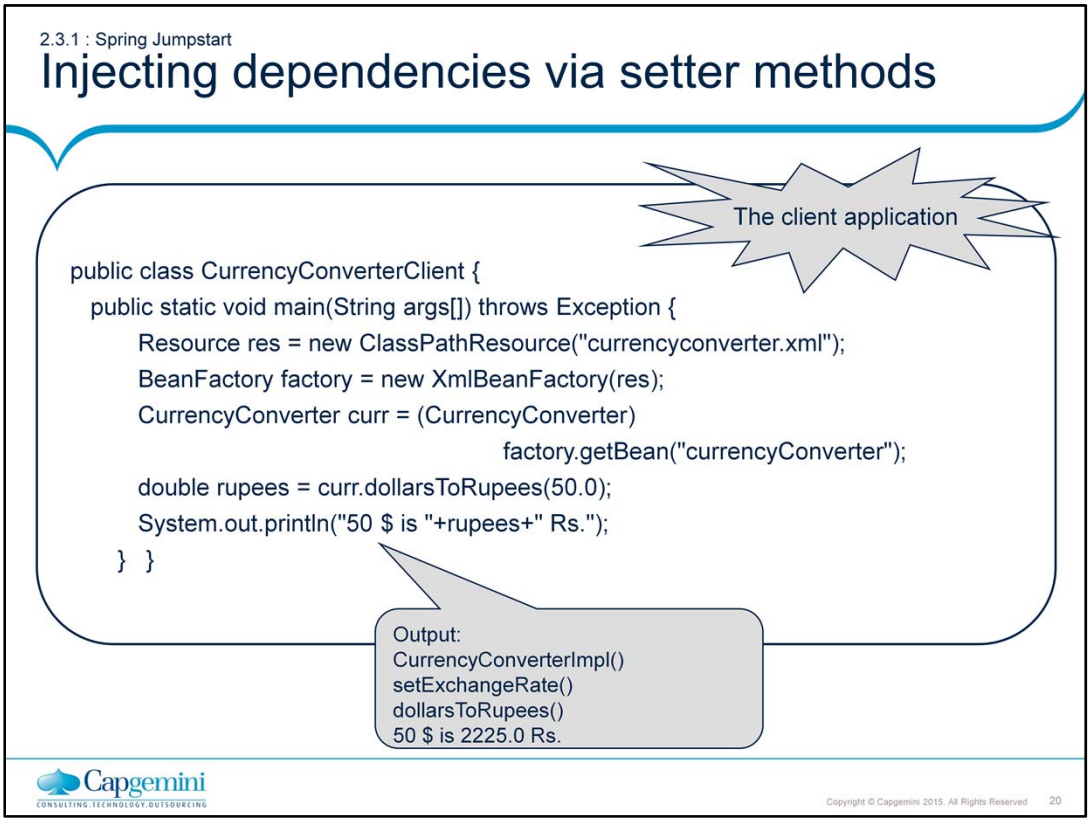
The example shows a service class whose purpose is to print the value of dollars converted to rupees. The listing above shows CurrencyConverter.java, an interface that defines the contract for the service class.
CurrencyConverterImpl.java implements the CurrencyConverter interface. Although it is not necessary to hide the implementation behind an interface, its highly recommended as a way to separate the implementation from its contract.
The CurrencyConverterImpl class has a single property exchangeRate. This property is simply a double variable that will hold the exchange rate passed by its setter method. We can also pass value through the constructor (We shall see this in the next example)



Question now is who will make a call to either the constructor or the setExchangeRate() method to set the exchangeRate property? The Spring configuration file in the above listing tells how to configure the CurrencyConverter service. This XML file declares an instance of a CurrencyConverterImpl in the Spring container and configures its exchangeRate property with a value of 44.50.
Notice the <beans> element at the root of the XML file. This is the root element of any Spring configuration file. The <bean> element is used to tell the Spring container about a class and how it should be configured. The id attribute is used to name the bean currencyConverter and the class attribute specifies the bean's fully qualified class name.
Within the <bean> element, the <property> element is used to set a property, in this case exchangeRate property. By using <property>, we are telling the Spring container to call setExchangeRate() when setting the property. This is called setter injection and is a straightforward way to configure and wire bean properties. The value of the exchange rate is defined using the value attribute. The following snippet of code illustrates roughly what the container does when instantiating the currencyConverter service based on the XML definition seen above.

```
CurrencyConverterImpl currencyConverter = new CurrencyConverterImpl();
currencyConverter.setExchangeRate(44.50);
```

Notice the configuration metadata is represented in XML. But it can also be done using Java annotations, or Java code.



Finally look at the class above. This class loads the Spring container and uses it to retrieve the currencyConverter service. The BeanFactory class used here is the Spring container. After loading the currencyconverter.xml file into the container, the main() method calls the getBean() method on the BeanFactory to retrieve a reference to the CurrencyConverter service. With this reference in hand, it finally calls the dollarsToRupees() method. When we run the above application (CurrencyConverterClient.java), output is as seen above.

This example illustrates the basics of configuring and using a class in Spring. It is simple because it only illustrates how to configure a bean by injecting a double value into a property. The real power of Spring lies in how beans can be injected into other beans using IoC (Inversion of Control – which shall be discussed in the next topic).

2.3.1 : Spring Jumpstart

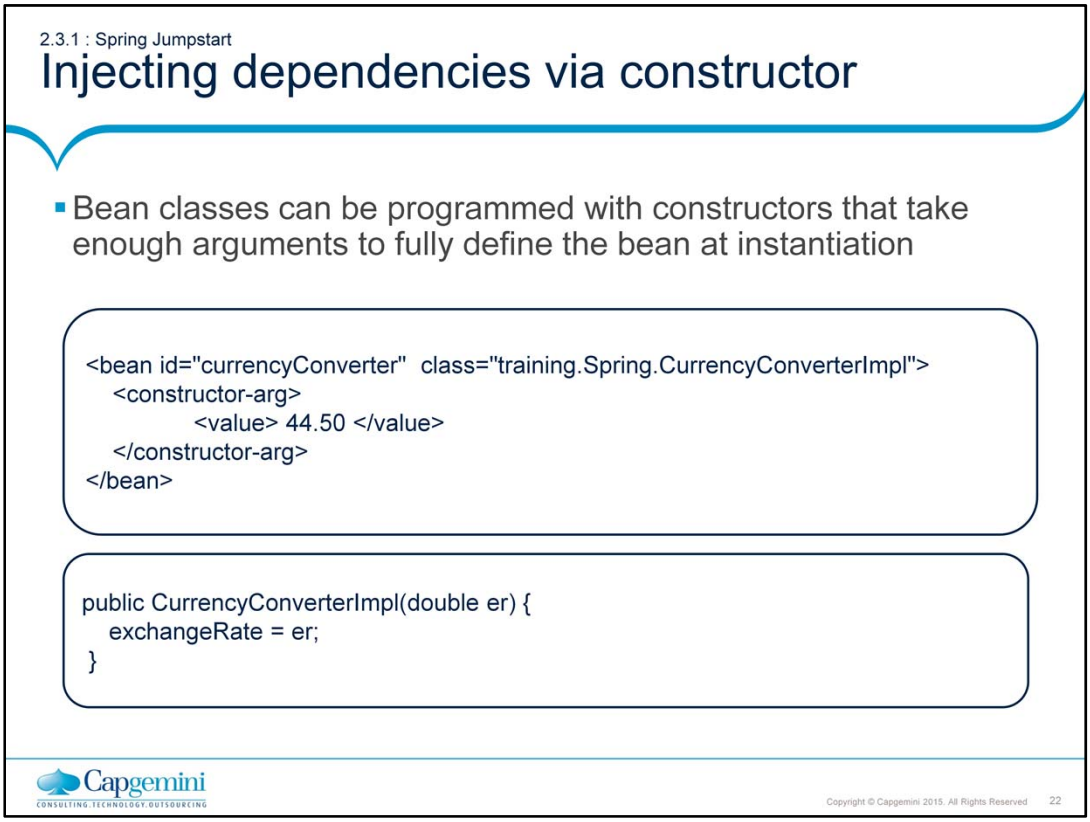
DemoSpring_1

- This demo illustrates how the container will instantiate the CurrencyConverter service using setter injection.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

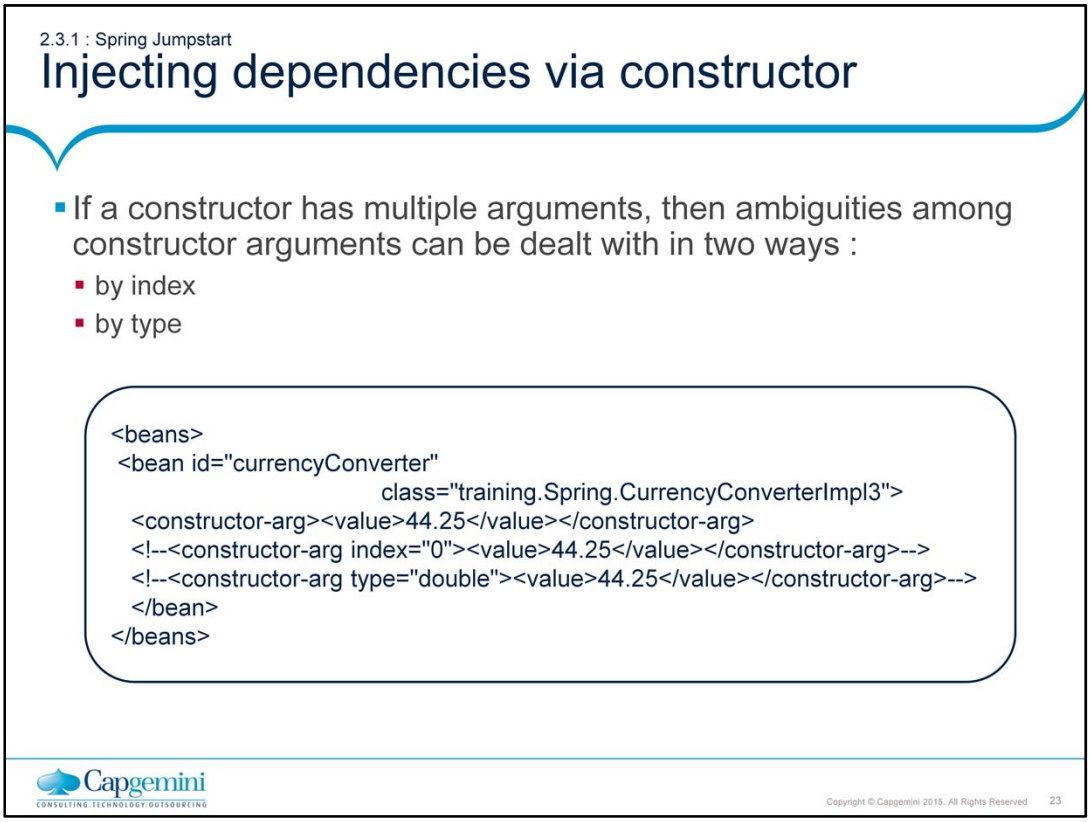
 21

Please refer to demo, DemoSpring_1.



Injecting dependencies via constructor:
Setter injection assumes that all mutable properties are available via a setter method. For one thing, when this type of bean is instantiated, none of its properties have been set and it could possibly be in a invalid state. Second, you may want all properties to be set just once, when the bean is created and become immutable after that point. This is impossible when all properties are exposed via setter methods.

In Java, a class can have multiple constructors and thus you can program your bean classes with constructors that take enough arguments to fully define the bean at instantiation. This is called constructor injection. In the above example, this is done by having Spring set the exchangeRate property through currencyConverterImpl's single argument constructor.



The <constructor-arg> element has an optional index attribute that specifies the ordering of the constructor arguments.

```
<constructor-arg index=”1”>
   <value> some-value </value>
<constructor-arg>
```

The type attribute lets you specify exactly what type each argument is supposed to be.

```
<constructor-arg type=”java.lang.String”>
   <value> some-value </value>
<constructor-arg>
```

The code above illustrates how the container will instantiate the CurrencyConverter service when using the <constructor-arg> element using the same classes seen earlier.

2.3.1 : Spring Jumpstart

DemoSpring_2

- This demo illustrates how the container will instantiate the CurrencyConverter service when using the <constructor-arg> element.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 24

Please refer to demo, DemoSpring_2.

Sometimes the only way to instantiate an object is through a static factory method. Spring is ready-made to wire factory-created beans through the <bean> element's factory-method attribute. DemoSpring_2 demonstrates this. Pls. refer to demos.

2.3 : Inversion of Control (IoC)

Using collections for injection

- Often, beans need access to collections of objects, rather than just individual beans or values.
- Spring allows you to inject a collection of objects into your beans.
- You can choose either <list>, <map>, <set> or <props> to represent a List, Map, Set or Properties instance.
- You will pass in the individual items just as you would with any other injection.

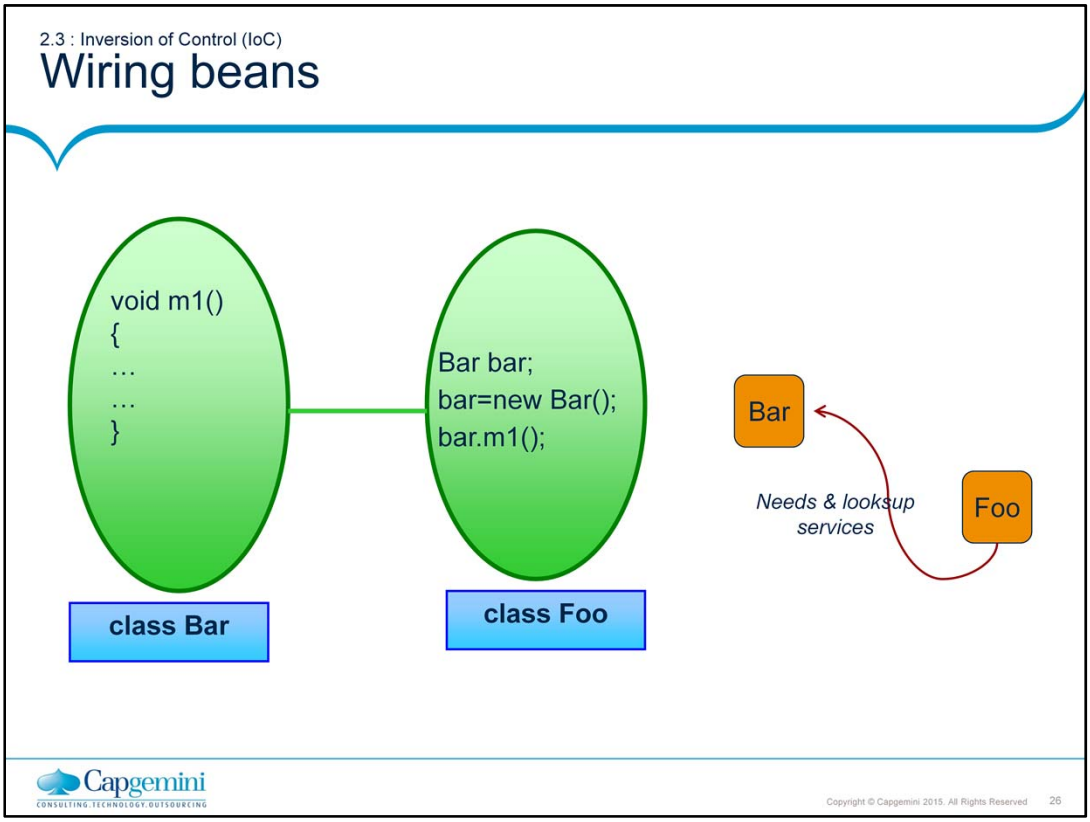
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 25

Example:

```
<bean id="complexObject" class="example.ComplexObject">
  <property name="people">
     <props>
          <prop key="HarryPotter">The magic property</prop>
           <prop key="JerrySeinfeld">The funny property</prop>
      </props>
  </property>
  <property name="someList">
      <list>
            <value>red</value>
             <value>blue</value>
      </list>
    </property>
<property name="someMap">
    <map>
          <entry key="an entry" value="just some string"/>
          <entry key ="a ref" value-ref="myDataSource"/>
    </map>
</property>
</bean>
```

One example is also available. Refer to demo, DemoSpring_5



Software code is normally broken down into logical components or services that interact with one another. In java, these components are usually instances of Java classes or objects. Each object must use or work with other objects in order to do its job.

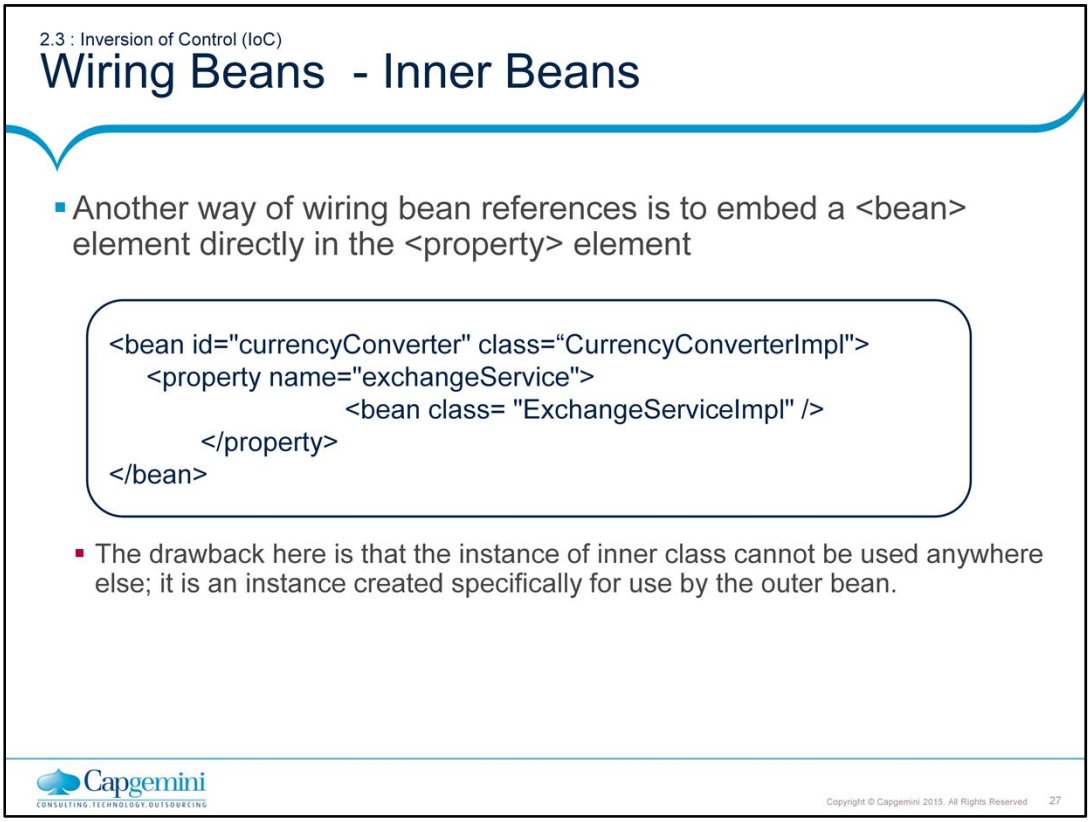
To understand this better, let us see a situation in which two java classes need to communicate. In the above figure, class Foo depends on an instance of class bar for some functionality. Traditionally, Foo creates instance of Bar using new operator or obtains one from a factory class.

In IoC however, an instance of Bar is provided to Foo at runtime by some external processes ie the container will handle the “injection” of an appropriate implementation.

Inversion of Control is best understood through the term the “Hollywood Principle,” which basically means “Don’t call me, I’ll call you.”

We don't directly connect our components and services together in code but describe which services are needed by which components in a configuration file. A container is responsible for hooking it up. This concept is similar to 'Declarative Management'.

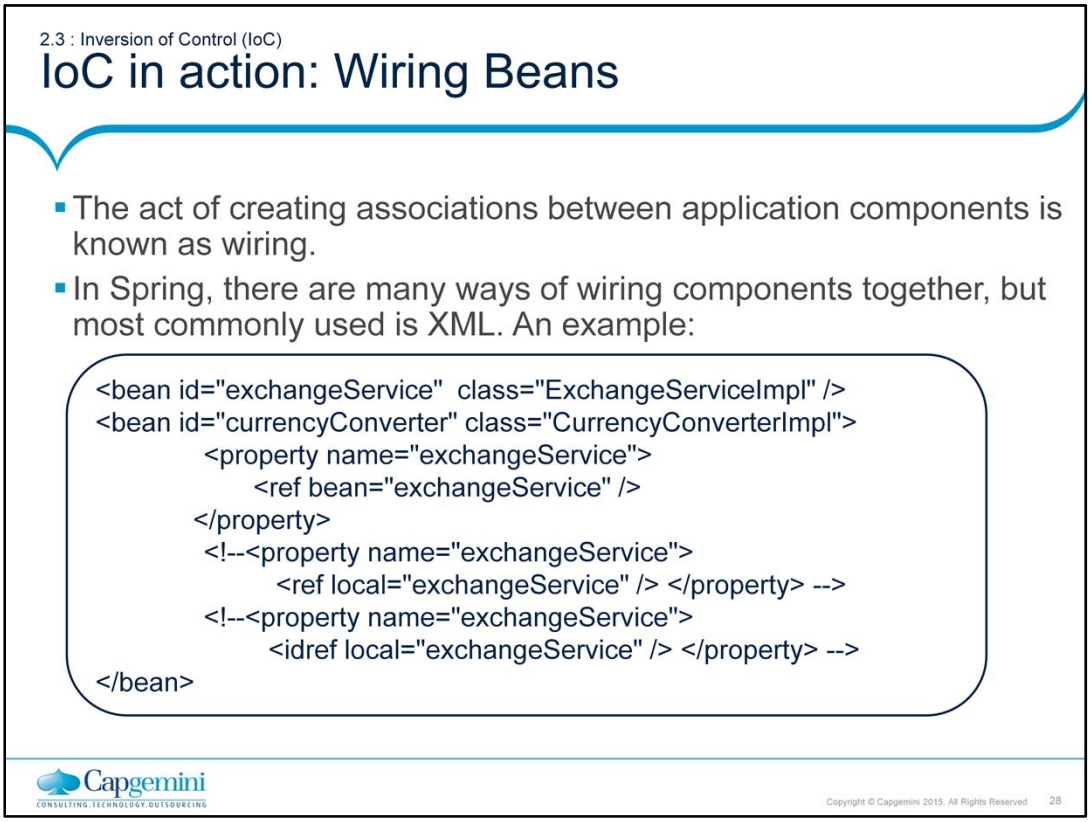
Spring-enabled applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application. Difference lies in how these classes are configured and introduced to each other. Typically a Spring application has an XML (the Spring configuration) file that describes how to configure these classes.



The drawback of the above method is that you cannot reuse the instance of ExchangeServiceImpl anywhere else – it is an instance created by specifically for use by the currencyConverter bean.

Inner beans aren't limited to setter injection. You may also wire inner beans into constructor arguments. E.g.:

```
<bean id="currencyConverter"
                class="com.Spring.CurrencyConverterImpl4">
    <constructor-arg>
        <bean class="com.Spring.ExchangeServiceImpl" />
    </constructor-arg>
</bean>
```



The act of creating associations between application components is known as wiring. In Spring there are many ways of wiring components together, but most commonly used is XML. A BeanFactory will load the bean definition and wire the beans together. In the above example, the <ref> subelement of the <property> tag is used to set the value or constructor argument to be a reference to another bean from the factory. The bean attribute is the ID of the other bean. Specifying the target bean by using the bean attribute of the ref tag is the most general form, and will allow creating a reference to any bean in the same BeanFactory or parent BeanFactory.
Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the local attribute must be the same as the id attribute of the target bean. The XML parser will issue an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors are early as possible) if the target bean is in the same XML file.
Specifying the idref tag will allow Spring to validate at deployment time whether the other bean actually exists.

2.3 : Inversion of Control (IoC)

DemoSpring_3

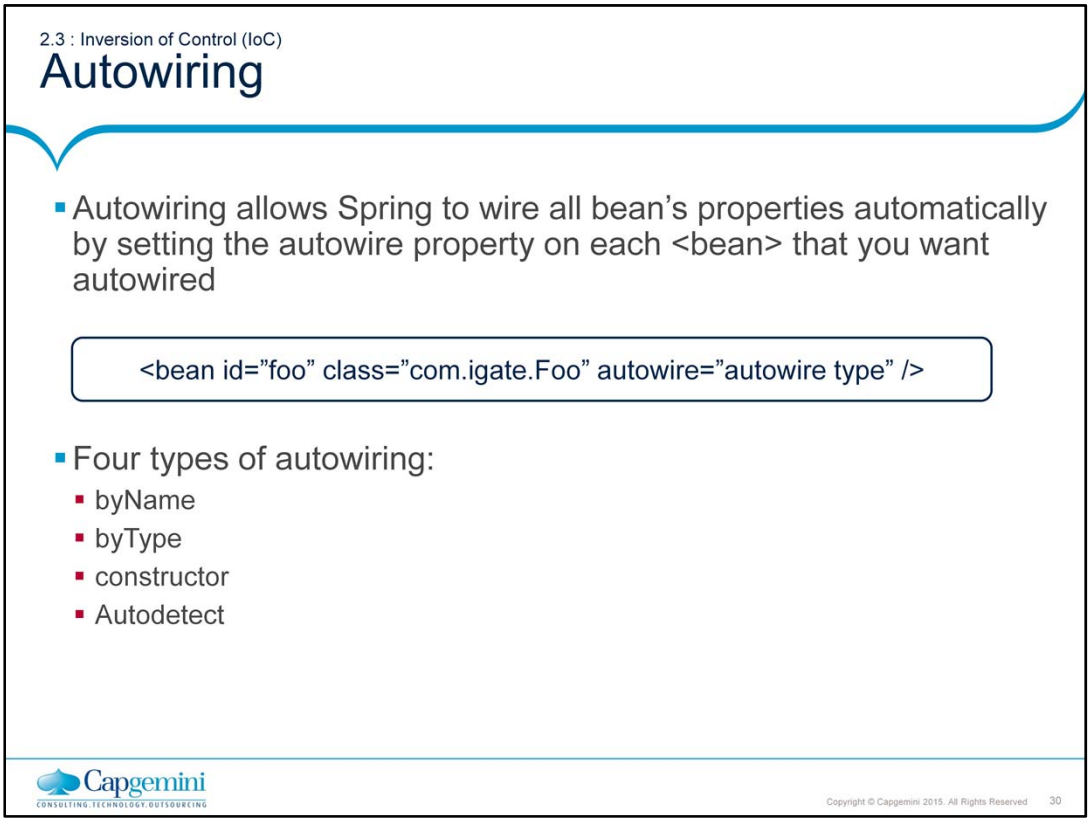
- This demo illustrates how the BeanFactory loads the bean definition and wires the beans together

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 29

Please refer to demo, DemoSpring_3. CurrencyConverterClient.java uses an XmlBeanFactory (a BeanFactory implementation) to load currencyconverter.xml and to get a reference to the CurrencyConverter object. It then invokes the dollarsToRupees() method.
There are two application objects. The instance of ExchangeRateImpl is responsible for retriving the exchange rate, using which, the currency converter instance would convert currency. When the CurrencyConverterImpl is instantiated, it in turn first instantiates the ExchangeServive bean. The ExchangeService instance in the CurencyConverterImpl class then exposes its getExchangeRate() method to return the exchange rate.

Summarizing how the container initializes and resolves bean dependencies:
The container first initializes the bean definition, without initializing the bean itself, typically at the time of the container startup. The bean dependencies may be explicitly expressed in the form of constructor arguments or arguments to a factory method and/or bean properties.
Each property or constructor argument in a bean definition is either an actual value to set, or a reference to another bean in the bean factory.
Constructor arguments or bean properties that refer to another bean will force the container to create or obtain that other bean first. Effectively, the referred bean is a dependent of the calling bean. This can trigger a chain of bean creation.
Every Constructor argument or bean property must be able to be converted from String format to the actual format expected. Spring is able to convert from String to built-in scalar types like int, float etc. Spring uses JavaBeans PropertyEditors to convert all other types.



So far, you have seen how to wire all your bean's properties explicitly. You can also have Spring wire them automatically by setting the autowire property on each bean that you want autowired.
There are four types of autowiring:
byName: Attempts to find a bean in the container whose name (or id) is same as the name of the property being wired. If matching bean is not found, then property will remain unwired.
byType: Attempts to match all properties of the autowired bean with beans whose types are assignable to the properties. Properties for which there's no matching bean will remain unwired.
constructor : Tries to match a constructor of the autowired bean with beans whose types are assignable to the constructor arguments. In the event of ambigous beans or ambigous constructors, an org.Springframework.beans.factory.UnsatisfiedDependencyException will be thrown.
autodetect : Attempts constructor autowiring first. If that fails, attempts autowiring byType.

2.3 : Inversion of Control (IoC)

DemoSpring_4

- This demo illustrates automatically wiring your beans

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 31

Refer to demo, DemoSpring_4 .This uses the same exchange service and currency converter service seen in earlier examples. However, by using the autowire attribute in the currencyconverter.xml we can execute the client program even without specifying the exchange service.

```
<bean id="exchangeService" class="com.igate.intro.ExchangeService">
  <constructor-arg><value>44.25</value></constructor-arg>
</bean>
```

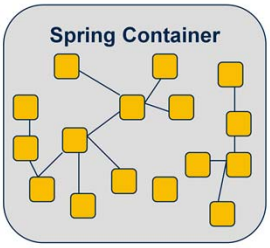
```
<bean id="currencyConverter"
class="com.igate.intro.CurrencyConverter" autowire="byName"/>
```

```
public class CurrencyConverter
{
private ExchangeService exchangeService;
..... }
```

2.4 : Bean containers

Bean containers: concept

- The container or bean factory is at the core of the Spring framework and uses IoC to manage components.
- Bean factory is responsible to create and dispense beans.
- It takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.
- Spring has two types of containers:
  - Bean factories that are the simplest, providing basic support for dependency injection
  - Application contexts that build on bean factory by providing application framework services

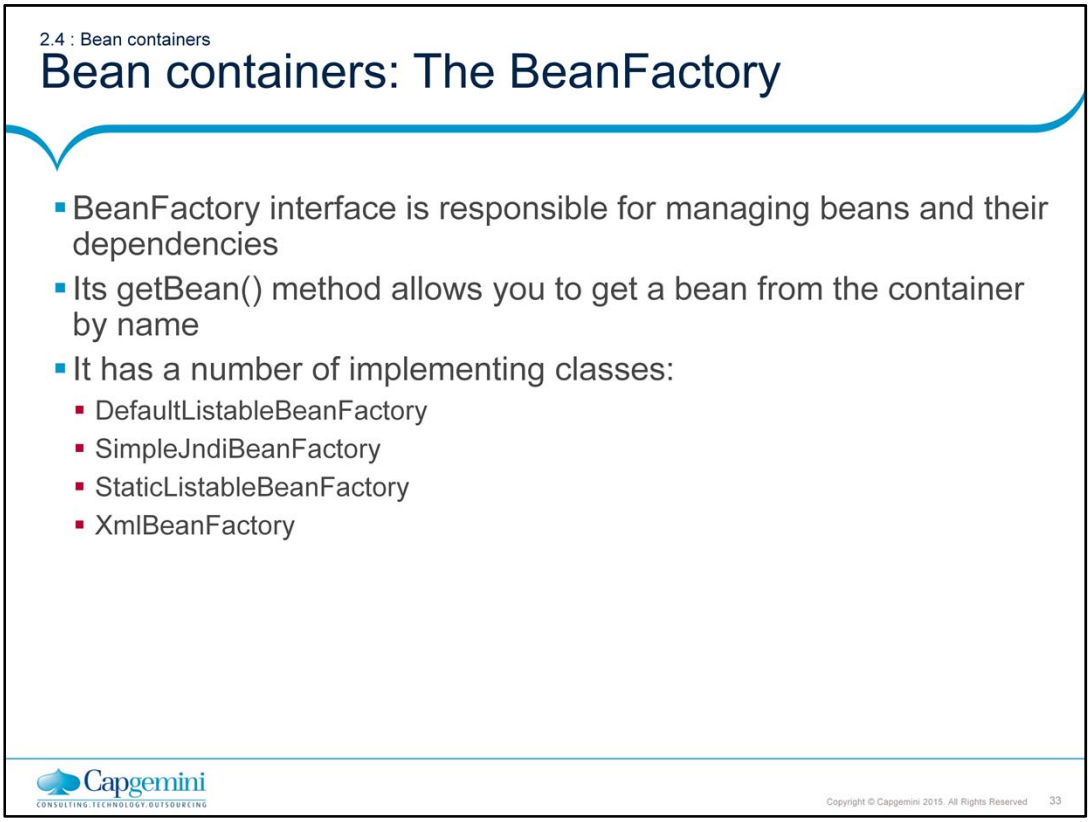


Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

 32

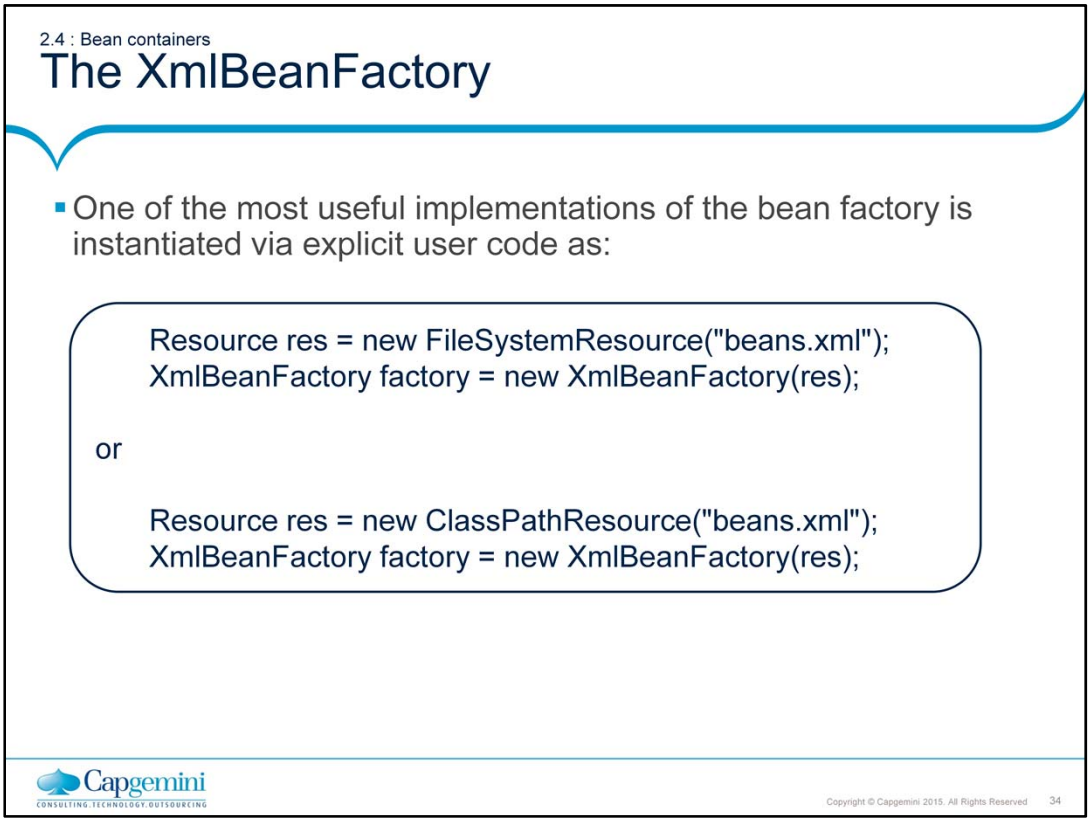
The container is at the core of the Spring framework and uses IoC to manage components. The basic IoC container in Spring is called the bean factory. Any bean factory allows the configuration and wiring of objects using dependency injection, in a consistent and workable fashion. A bean factory also provides some management of these objects, with respect to their lifecycles. Thus,
Bean factory is a class whose responsibility is to create and dispense beans.
A bean factory knows about many objects within an application.
Able to create associations between collaborating objects as they are instantiated.
This removes the burden of configuration from the bean itself and the bean's client. As a result, when a bean factory hands out objects, those objects are fully configured, aware of their collaborating objects and ready to use.
A bean factory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.

Spring actually comes with two different types of containers:
Beanfactory interface: provides an advanced configuration mechanism capable of managing any type of object.
ApplicationContext interface : is a sub-interface of BeanFactory. It allows easier integration with Spring's AOP features, message resource handling, event publication, and application-layer specific contexts such as the WebApplicationContext for use in web applications.
We shall look at the Application context in detail later.



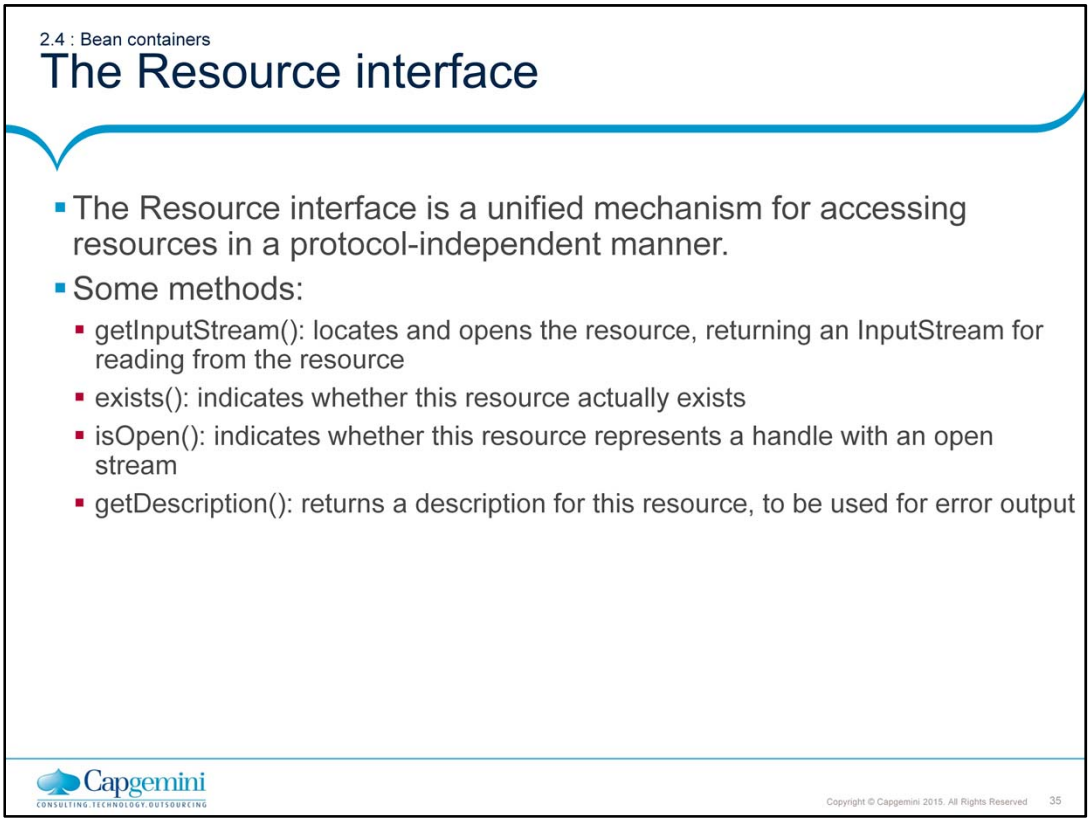
Lets start our exploration of Spring containers with the most basic of Spring containers : the BeanFactory.
Bean factory is responsible for managing beans and their dependencies. Your application interacts with Spring DI container via the BeanFactory interface. It has a getBean() method that allows you to get a bean from the container by name.
Additional methods allow you to query the bean factory to see if bean exists, to find the type of bean and to find if a bean is configured as a singleton.
Different bean factory implementations exist to support varying levels of functionality with XmlBeanFactory being the most common representation. A partial listing of the impelmenting classes follows:
DefaultListableBeanFactory
SimpleJndiBeanFactory
StaticListableBeanFactory
XmlBeanFactory

Please refer to the Spring documentation for more information on these classes

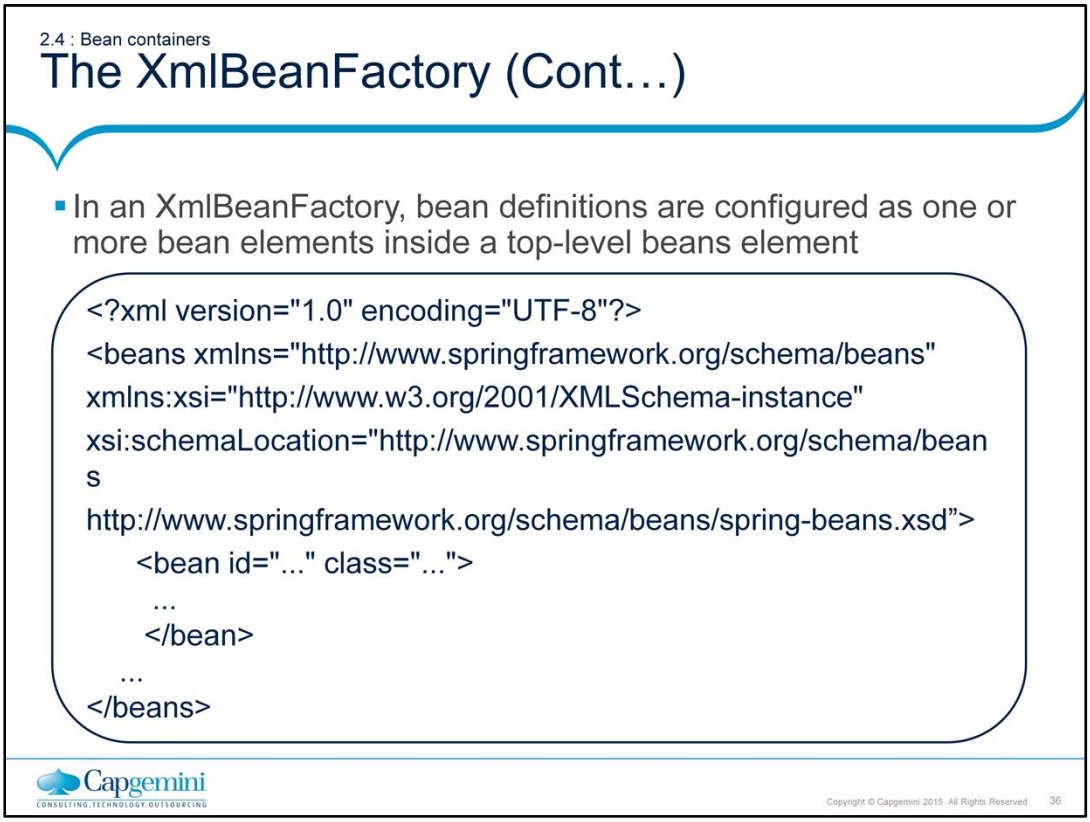


One of the most useful implementations of the bean factory is the org.Springframework.beans.factory.xml. XmlBeanFactory. The BeanFactory is instantiated via explicit user code such as shown above.
This simple line of code tells the bean factory to read the bean definitions from the XML file (beans.xml in this case). But the bean factory doesn't instantiate the beans just yet. Beans are "lazily" instantiated into bean factories, meaning that while the bean factory will immediately load the bean definitions, beans themselves will not be instantiated until they are needed.

The Spring IoC container consumes some form of configuration metadata; which 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. When using XML-based configuration metadata, you write bean definitions for those beans that you want the Spring IoC container to manage, and then let the container do its stuff.
Note
XML-based metadata is by far the most commonly used form of configuration metadata. It is not however the only form of configuration metadata that is allowed. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. The XML-based configuration metadata format really is simple though, and so the majority of this material will use the XML format to convey key concepts and features of the Spring IoC container.



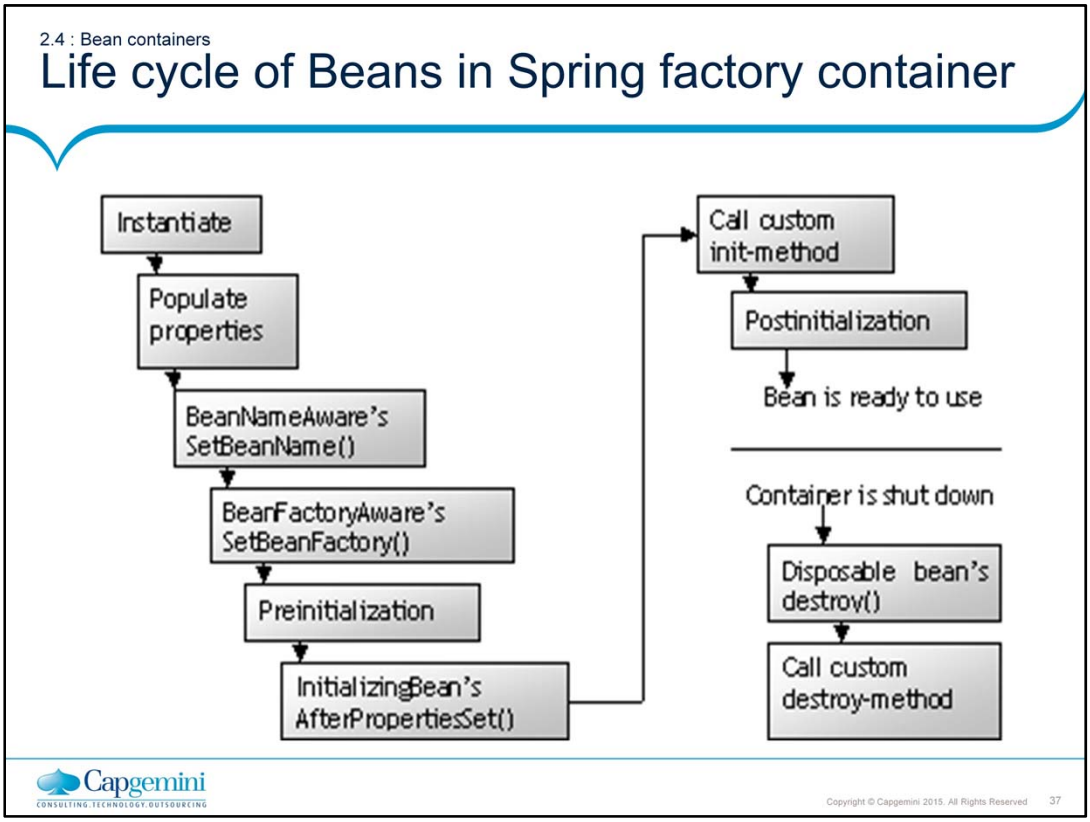
The Resource interface: Often, an application needs to access a variety of resources in different forms. You may need to access some configuration data stored in a file in the filesystem, some image data stored in a JAR file on the classpath, or maybe some data on a server elsewhere. Spring provides a unified mechanism for accessing resources in a protocol-independent manner. This means that your application can access a file resource in the same way, whether it is stored in the file system, the classpath or on a remote server.
At the core of the Spring's support is the Resource interface. This defines self-explanatory methods mentioned above. There are a number of implementations that come supplied straight out of the box in Spring:
UrlResource : The UrlResource wraps a java.net.URL, and may be used to access any object that is normally accessible via a URL, such as files, an HTTP target, an FTP target, etc.
ClassPathResource : This class represents a resource which should be obtained from the classpath. This uses either the thread context class loader, a given class loader, or a given class for loading resources.
FileSystemResource : This is a Resource implementation for java.io.File handles. It obviously supports resolution as a File, and as a URL.
ServletContextResource: This is a Resource implementation for ServletContext resources, interpreting relative paths within the relevant web application's root directory.
Two of the implemented classes examples for this interface ie. ClassPathResource and FileSystemResource, are shown in previous slide. Please refer to the Spring documentation for more details.



A BeanFactory configuration consists of, at its most basic level, definitions of one or more beans that the BeanFactory must manage. In an XmlBeanFactory, these are configured as one or more bean elements inside a top-level beans element.
The first versions of Spring used a DTD. But Spring 2.0 onwards uses schema for the xml configuration file.
To retrieve a bean from a bean factory, simply call the getBean() method, passing it the name of the bean you want to retrieve.

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

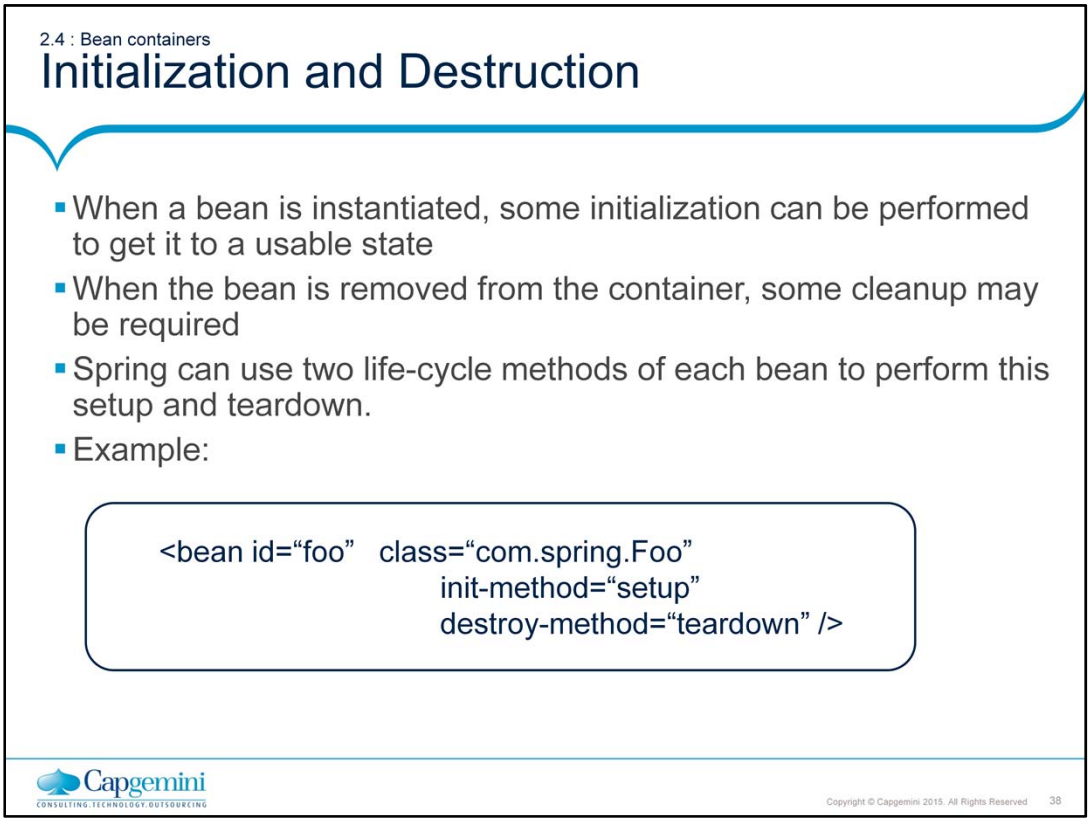
When getBean() is called, the factory will instantiate the bean and begin setting the bean's properties using dependency injection. Thus begins the bean's life cycle within the container (explained further on).



In a traditional Java application, the life cycle of a bean is fairly simple. Java's new keyword is used to instantiate the bean and it is ready to use. In contrast, the life cycle of a bean within a Spring container is a bit more elaborate.
A bean factory performs several setup steps before a bean is ready to use.
The container finds the bean's definition and instantiates the bean.
Using dependency injection, Spring populates all the properties as specified in the bean definition.
If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean's ID.
If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory() passing an instance of itself.
If there are any BeanPostProcessers associated with the bean, their PostProcessBeforeInitialization() methods will be called.
If an init-method is specified for the bean, it will be called.
Finally, if there are any BeanPostProcessers associated with the bean, their PostProcessAfterInitialization() methods will be called.
The bean is now ready to be used and will remain in the bean factory until it is no longer needed. It is removed from the factory in two ways:

If the bean implements the DisposableBean interface, the destroy() method is called.

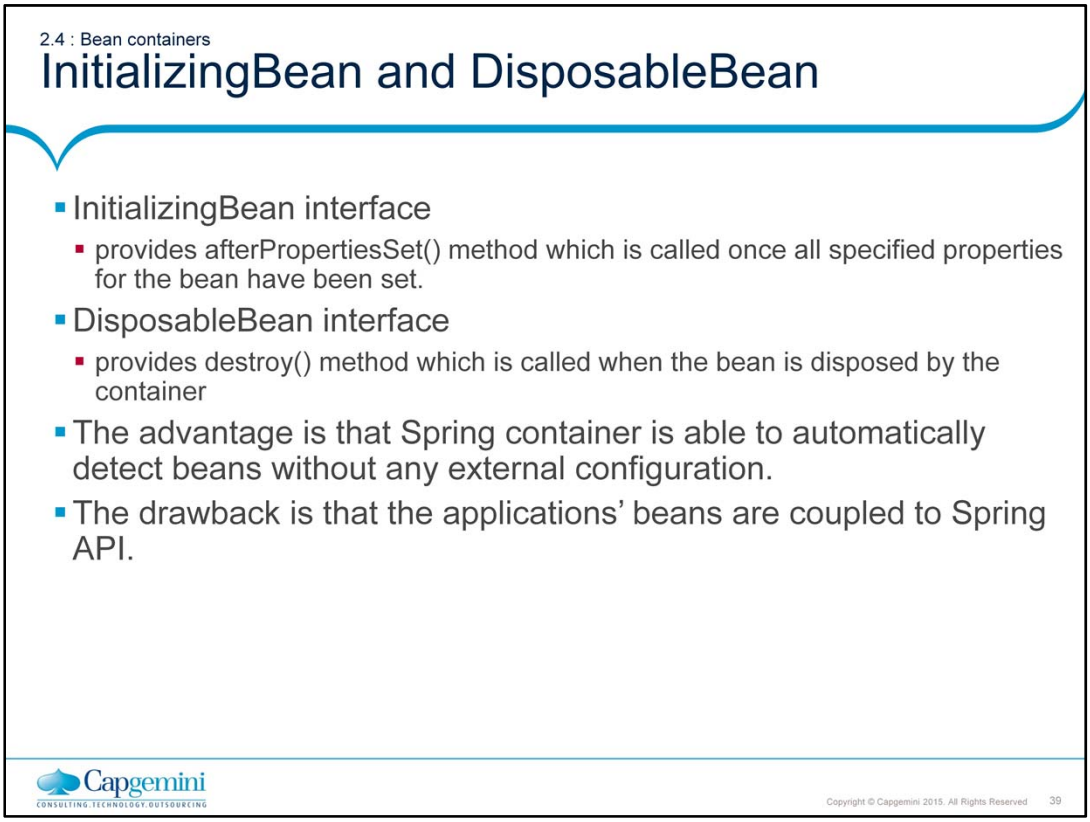
If a custom destroy-method is specified, it will be called.



Declaring a custom init-method in your bean's definition specifies a method that is to be called on the bean immediately upon instantiation. Similarly, a custom destroy-method specifies a method that is called just before a bean is removed from the container.
E.g., the init-method="setup" in the above example calls setup() method in the bean class when bean is loaded into container and teardown() when bean is removed from container.

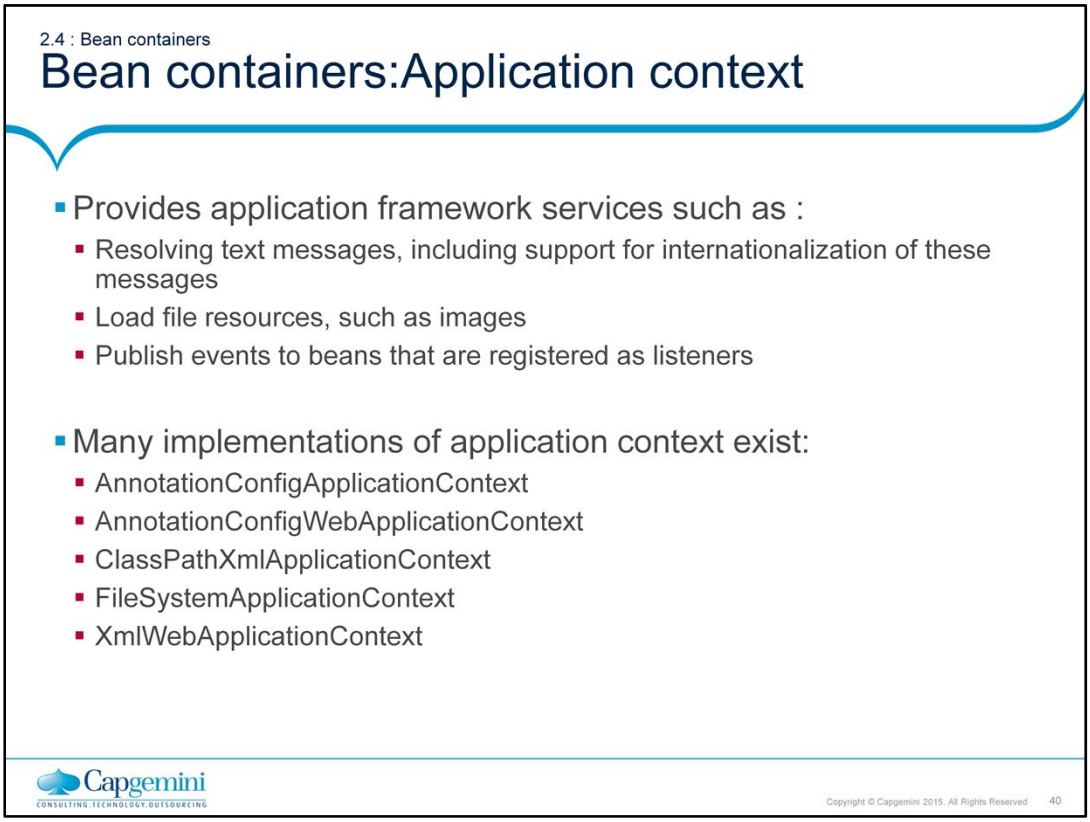
Defaulting init-method and destroy-method:
If many of the beans in a context definition file will have initialization or destroy methods with same name, you don' have to declare init-method or destroy-method on each individual bean. Instead, you can take advantage of the default-init-method and default-destroy-method attributes on the <beans> element.

```
<beans……
default-init-method="tuneApplication"
default-destroy-method="cleanApplication" >
……
</beans>
```



As an option to init-method and destroy-method, we can also rewrite the bean class to implement two special Spring interfaces: InitializingBean and DisposableBean. The Spring container treats beans that implement these interfaces in a special way, by allowing them to hook into the bean lifecycle.
The InitializingBean interface provides a method afterPropertiesSet(). This is called once all specified properties for the bean have been set. This makes it possible for the bean to perform initialization that cannot be performed until all properties have been completely set.
DisposableBean provides a destroy() method which will be called on the other end of a bean-lifecycle., when the bean is disposed by the container.
The benefit of using these interfaces is that Spring container is able to automatically detect beans that implement them without any external configuration. However, the disadvantage of implementing these interfaces is that you couple your application's beans to Spring API.

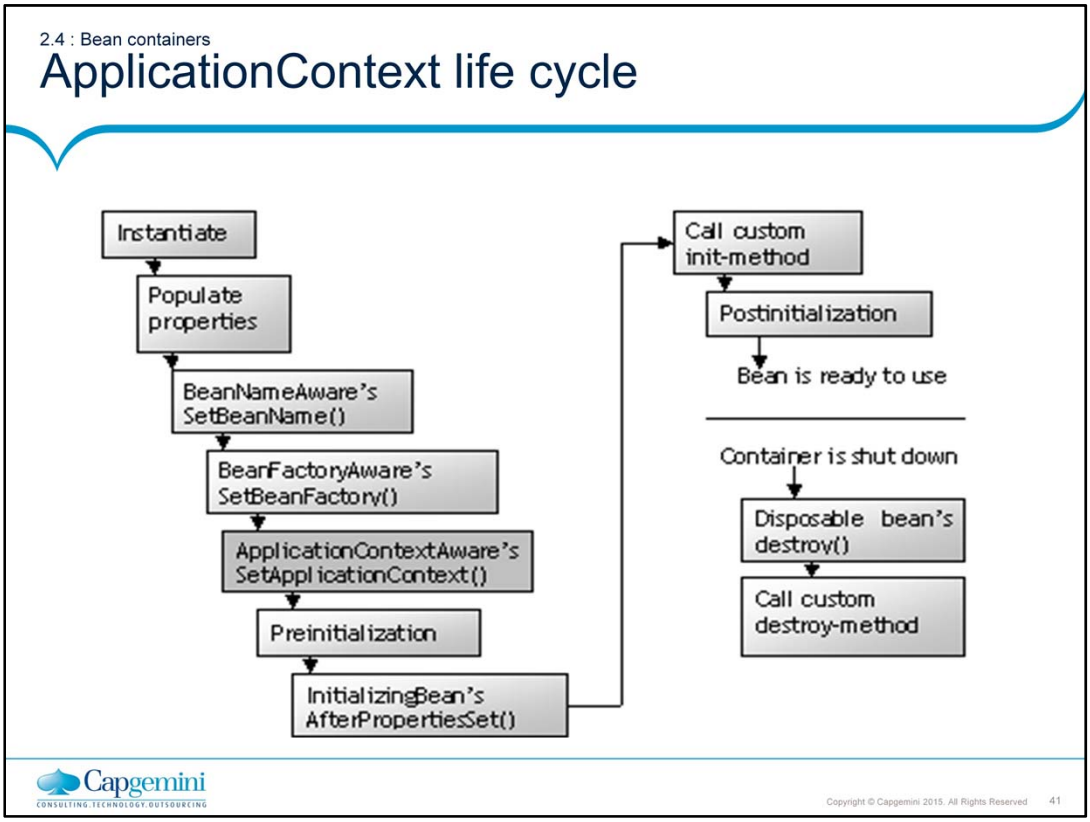
```
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.DisposableBean;
public class SampleBean implements InitializingBean, DisposableBean{
   …
    public void afterPropertiesSet(){…}
    public void destroy(){…}
…}
```



Application contexts build on bean factory by providing application framework services. A bean factory is fine for simple applications, but to take advantage of the full power of Spring framework, we need to use the application context container, which offers services mentioned in slide above. Many implementations of application context exist:
AnnotationConfigApplicationContext : Loads a Spring application context from one or more Java-based configuration classes
AnnotationConfigWebApplicationContext : Loads a Spring web application context from one or more Java-based configuration classes
ClassPathXmlApplicationContext : Loads context definition from a XML file located in the class path.
FileSystemApplicationContext: Loads context definition from an XML file in the file system.
XmlWebApplicationContext : Loads context definition from an XML file contained within a web application
Loading the ApplicationContext: Some examples :

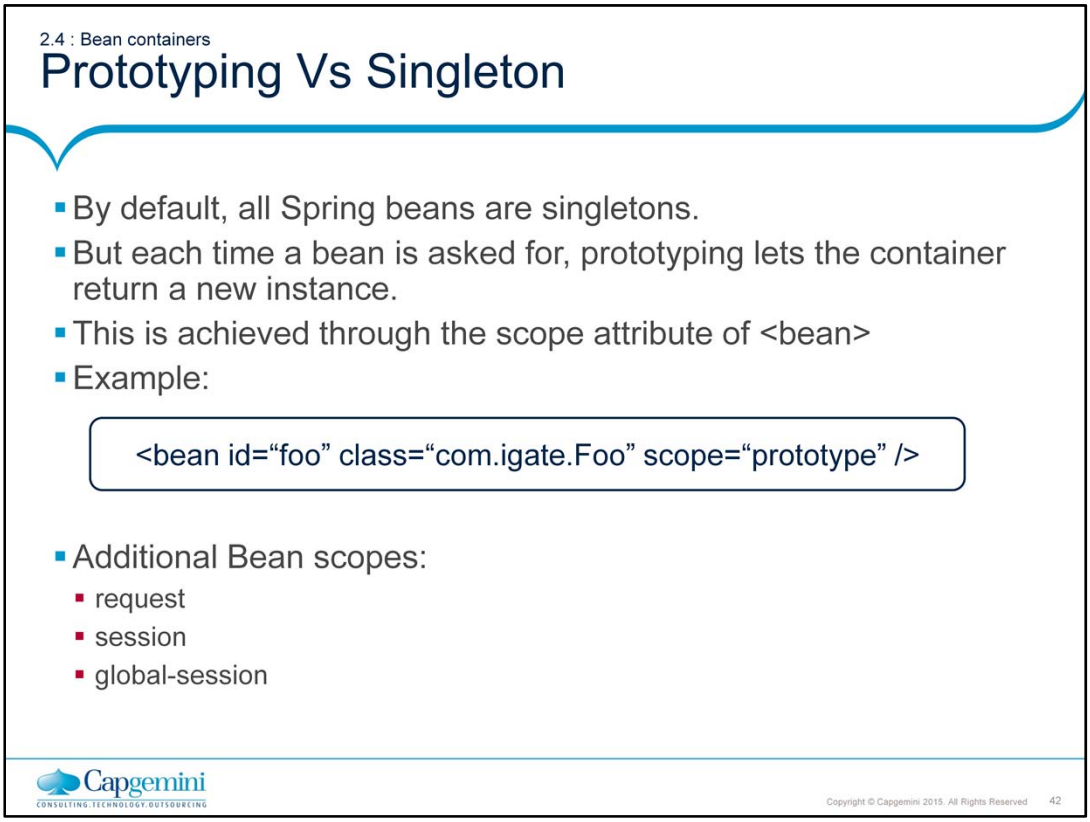
```
ApplicationContext ctx = new ClassPathXmlApplicationContext(“app.xml");
ApplicationContext ctx = new
FileSystemXmlApplicationContext(“/some/file/path/app.xml");
ApplicationContext ctx = new ClassPathXmlApplicationContext( new
String[]{“app1.xml“,”app2.xml”});  // combines multiple xml file fragments
```

Spring support Wildcards in application context constructor resource paths.
The resource paths in application context constructor values may be a simple path which has a one-to-one mapping to a target Resource, or alternately may contain the special "classpath*:" prefix and/or internal Ant-style regular expressions (matched using Spring's PathMatcher utility). Both of the latter are effectively wildcards.
(Continued on next page….)



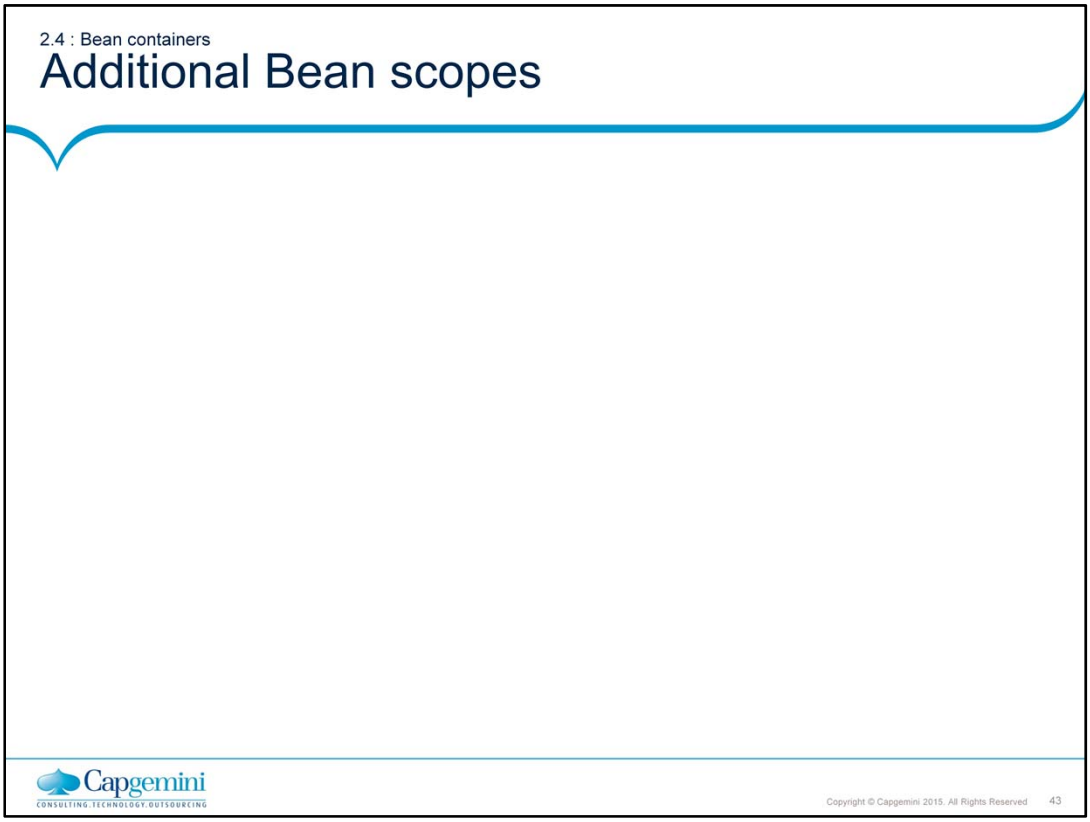
(Continued from previous page)
Eg.
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
classpath:com/mycompany/**/service-context.xml
ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath*:conf/appContext*.xml");

ApplicationContext life cycle:
The life cycle of a bean within a Spring ApplicationContext differs only slightly from that of a bean within a bean factory as shown in above figure.
The only difference is that if a bean implements the ApplicationContextAware interface, the setApplicationContext() method is invoked.

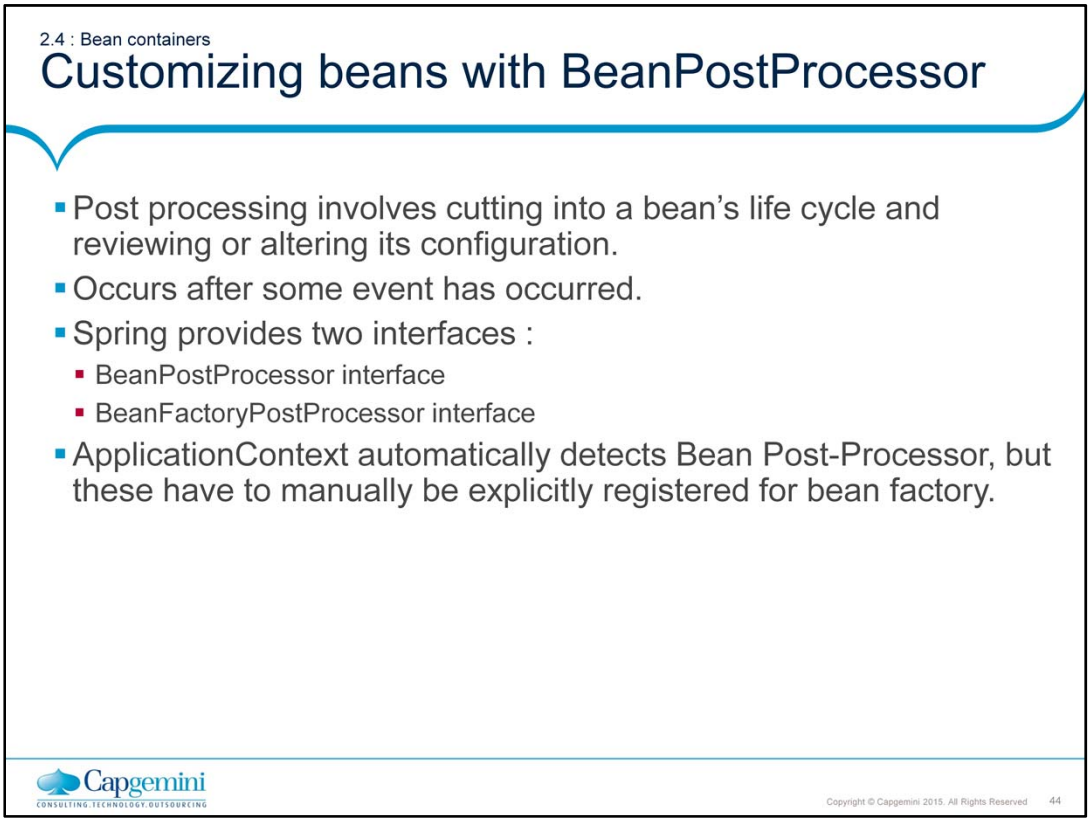


Singleton beans, the default, are created only once by the container and all calls to BeanFactory.getBean() return the same instance. The container will then hold and use the same instance of the bean whenever it is referenced again. This can be significantly less expensive in terms of resource usage than creating a new instance of the bean on each request.
A non-singleton, or prototype bean, may be specified by setting the scope attribute to prototype (see example above). The lifecycle of a prototype bean will often be different than a singleton. When a container is asked to supply a prototype bean, it's initialized and then used, but the container does not hold on to it past that point.
Prototyped beans are useful when you want the container to give a unique instance of a bean each time it is asked for, but you still want to configure one or more properties of the bean through Spring. Thus a new instance is created when getBean() is invoked with the bean's name.

Previous versions of Spring had IoC container level support for exactly two distinct bean scopes (singleton and prototype). Spring 2.0 onwards provides a number of additional scopes depending on the environment in which Spring is being deployed (for example, request and session scoped beans in a web environment).
It also provides integration points so that Spring users can create their own scopes. Beans can be defined to be deployed in one of a number of scopes. See the table in the next page for the different scopes.

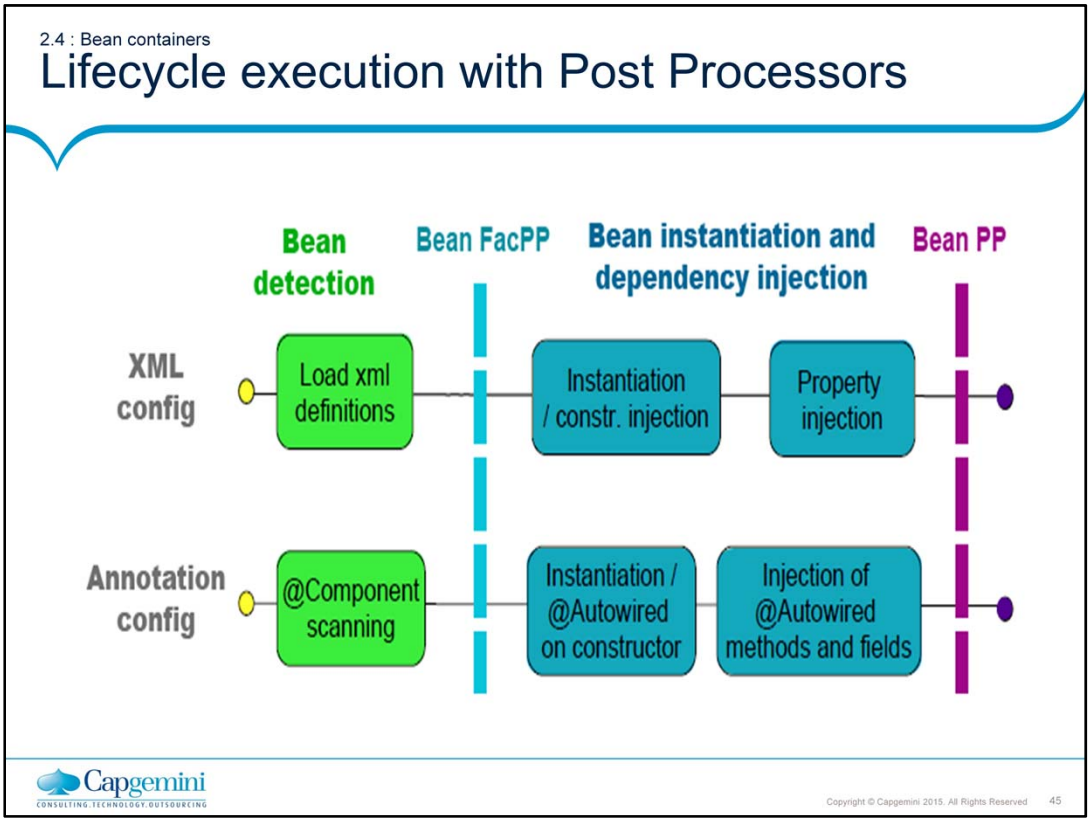


An example:
<bean id="accountService" class="com.foo.DefaultAccountService"
scope="singleton"/>



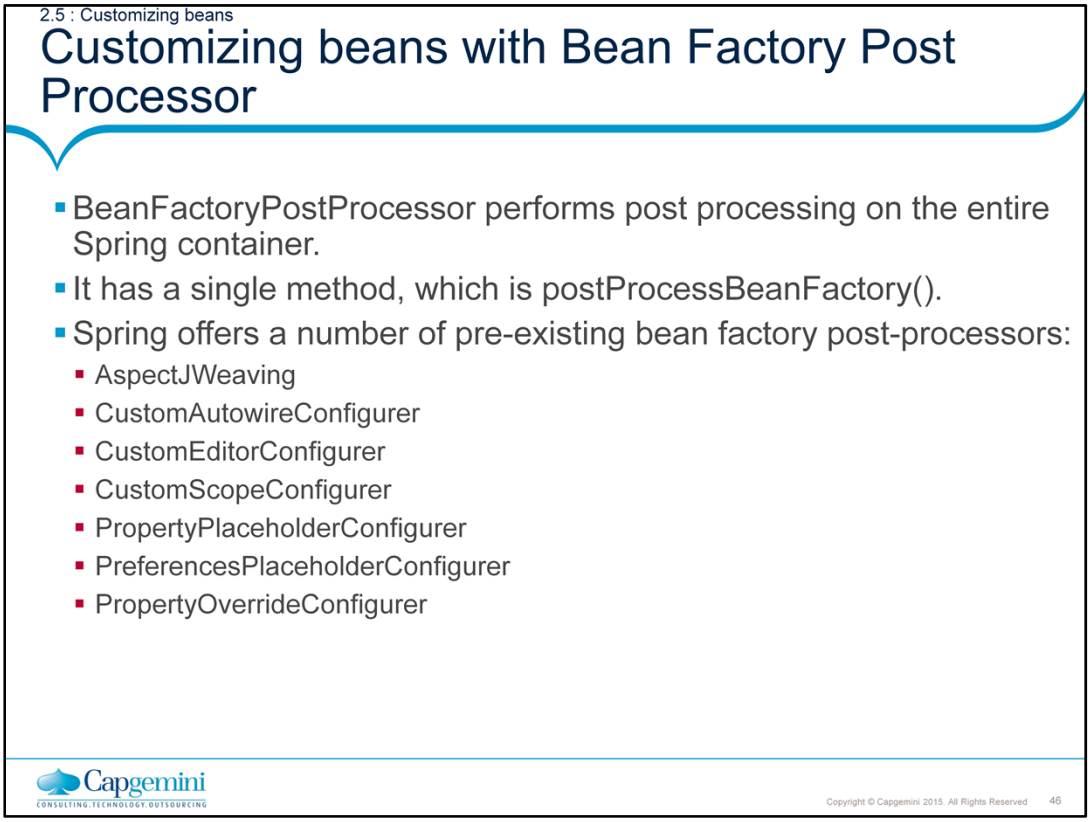
The lifecycle of the BeanFactory and ApplicationContext provide many opportunities to cut into the bean's life cycle to review or alter its configuration. This is called post processing and occurs after some event has occurred. A bean post-processor is a java class which implements the BeanPostProcessor interface, which consists of two callback methods:
postProcessBeforeInitialization: called immediately before bean initialization.
postProcessAfterInitialization: called immediately after bean Initialization.
BeanPostProcessors operate on bean (or object) instances; ie the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.
An ApplicationContext will automatically detect any beans which are deployed into it which implement the BeanPostProcessor interface, and register them as post-processors, to be then called appropriately by the factory on bean creation. Simply deploy the post-processor in a similar fashion to any other bean! However, for BeanFactory, bean post-processors have to manually be explicitly registered, with a code sequence as shown below.
Since this manual registration step is not convenient, and ApplictionContexts are functionally supersets of BeanFactories, it is generally recommended that ApplicationContext variants are used when bean post-processors are needed.

```
ConfigurableBeanFactory bf = new .....; // create BeanFactory
    // now register some beans and any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp); // now start using the factory ...
```



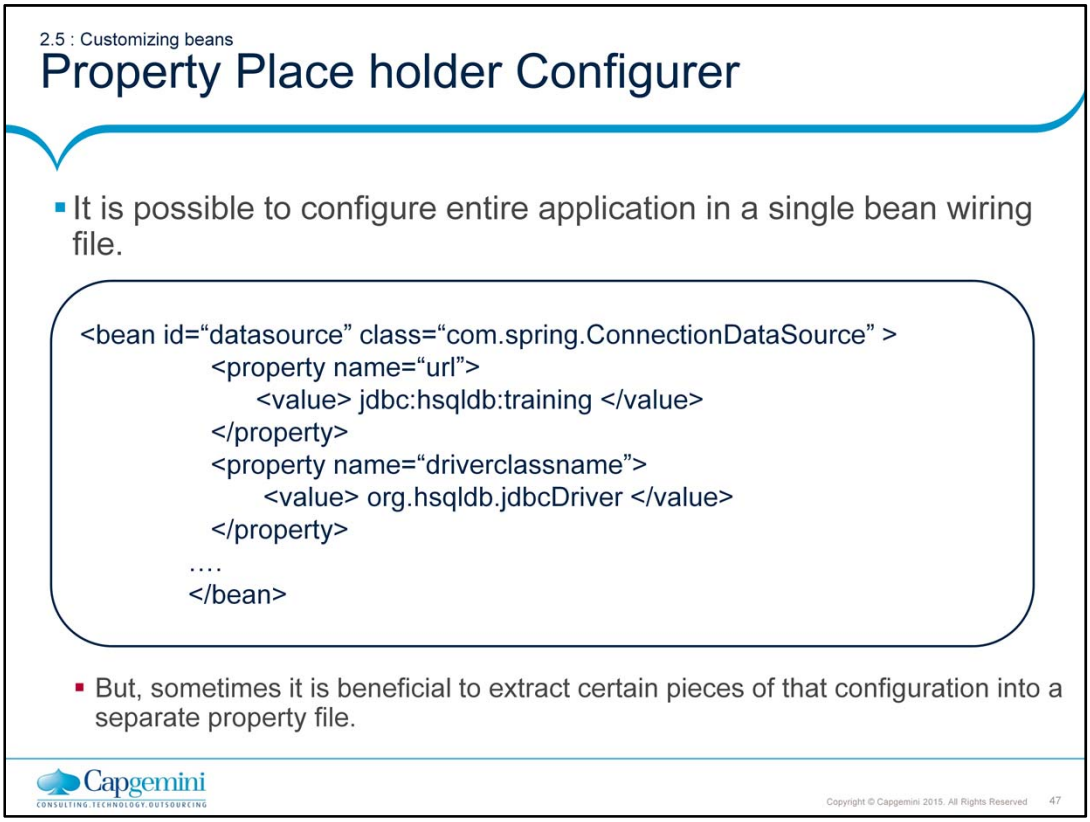
Discuss about annotation config later.

```
ConfigurableBeanFactory bf = new .....; // create BeanFactory
    // now register some beans and any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp); // now start using the factory ...
```

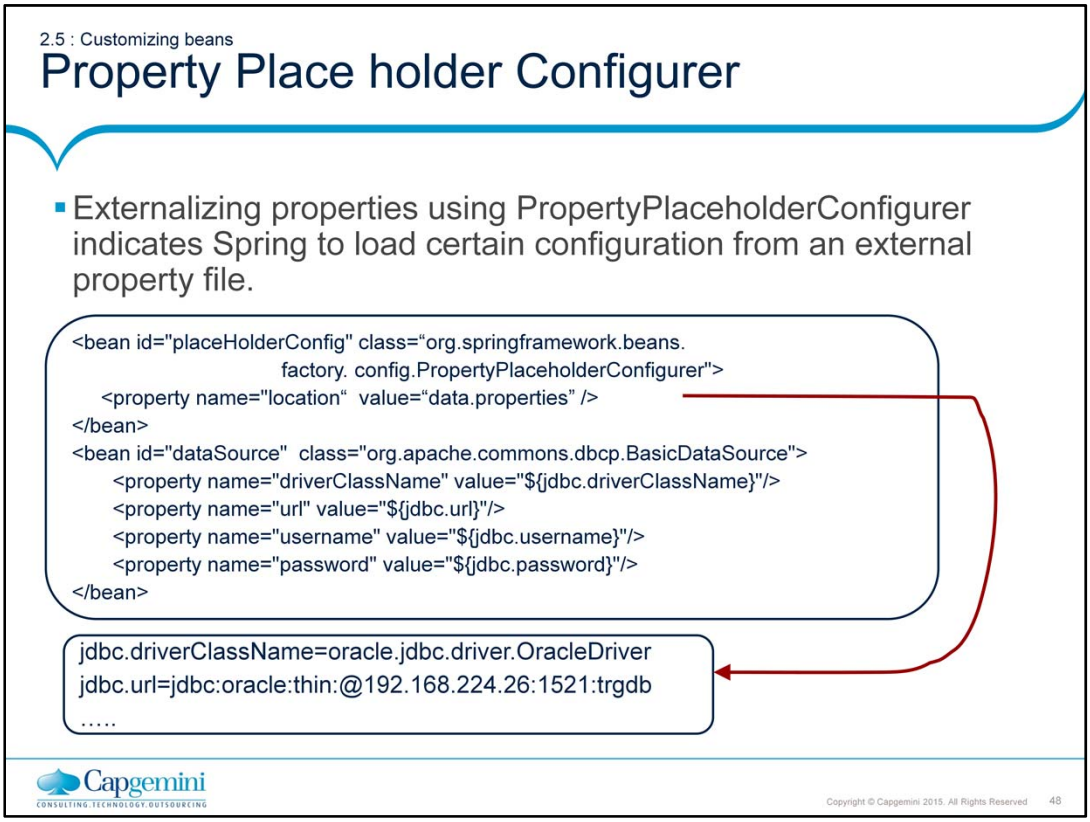


BeanFactoryPostProcessors operate on the bean configuration metadata; that is, the Spring IoC container allows BeanFactoryPostProcessors to read the configuration metadata and potentially change it before the container instantiates any beans other than BeanFactoryPostProcessors.!
Has a single method – postProcessBeanFactory(). This is called by Spring container after all bean definitions have been loaded but before any beans are instantiated (including BeanPostProcessor beans).

Spring offers a number of pre-existing bean factory post-processors. Two very useful implementations are:
PropertyPlaceholderConfigurer : Loads properties from one or more external property files and uses these properties to fill in place holder variables in the bean wiring XML file.
CustomEditorConfigurer : Lets you register custom implementation of java.beans.PropertyEditor to translate property wired values to other property types.
Let’s take a look at how you can use these implementations of BeanFactoryPostProcessor.



For the most part, it is possible to configure entire application in a single bean wiring file. But sometimes it is beneficial to extract certain pieces of that configuration into a separate property file. E.g. , a configuration concern common to many applications is configuring a data source. Traditionally, in Spring, you do this with the following XML in the bean wiring file (see above)
Configuring the data source directly in the bean wiring file may not be appropriate. The database specifics are a deployment detail and must be separated.



Fortunately, externalizing properties in Spring is easy if you are using ApplicationContext as your Spring container. You can use PropertyPlaceholderConfigurer to tell Spring to load certain configuration from an external property file as shown in the code snippet above. The location property tells Spring where to find the property file data.properties.
When Spring creates the bean, the PropertyPlaceholderConfigurer will step in and replace the place holder variables with the values from the property file. It pulls values from a properties file into bean definitions.

2.5 : Customizing beans

Demo: DemoSpring_6

- This demo shows how to use the PropertyPlaceholderConfigurer BeanFactoryPostProcessor

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

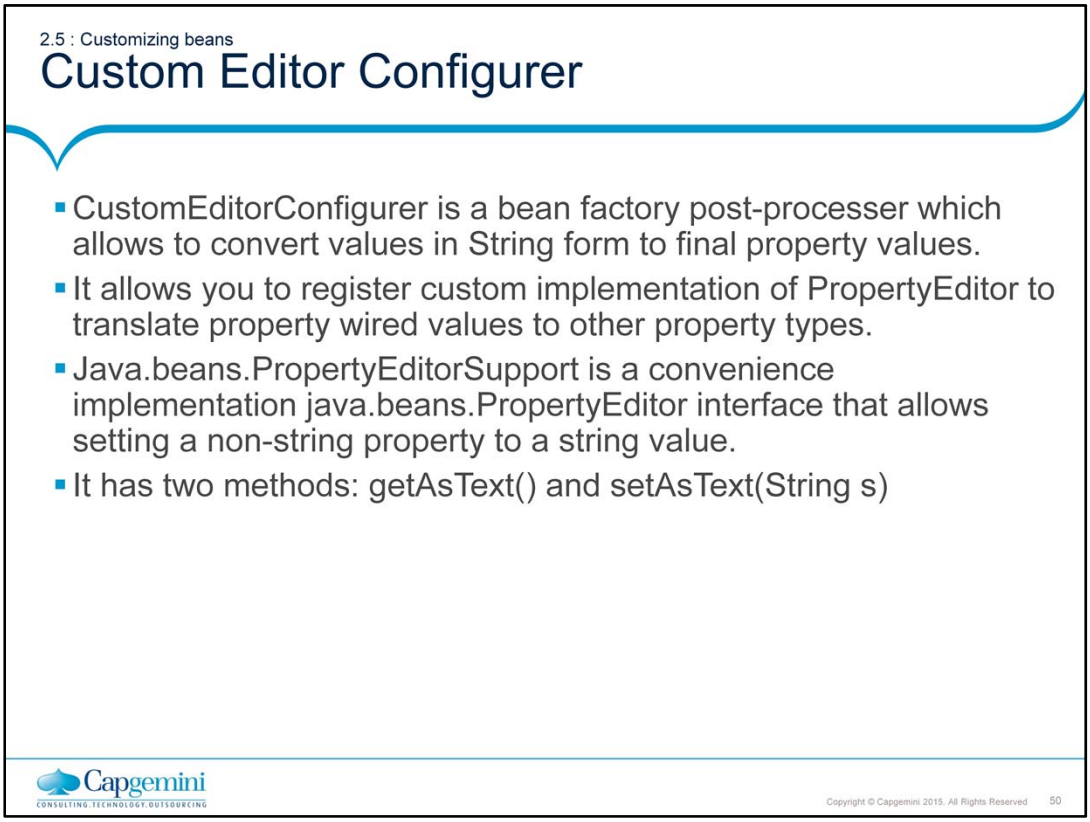
 49

Please refer to demo, DemoSpring_6. In this case user.java is a POJO with two properties – username and password. We shall set the properties of this bean during its instantiation using external properties file. user.properties is a properties file. user.xml has two place holder variables ${username} and ${password}

```
<bean id="user" class="training.spring.User">
  <property name="username"><value>${username}</value></property>
  <property name="password"><value>${password}</value></property>
</bean>
```

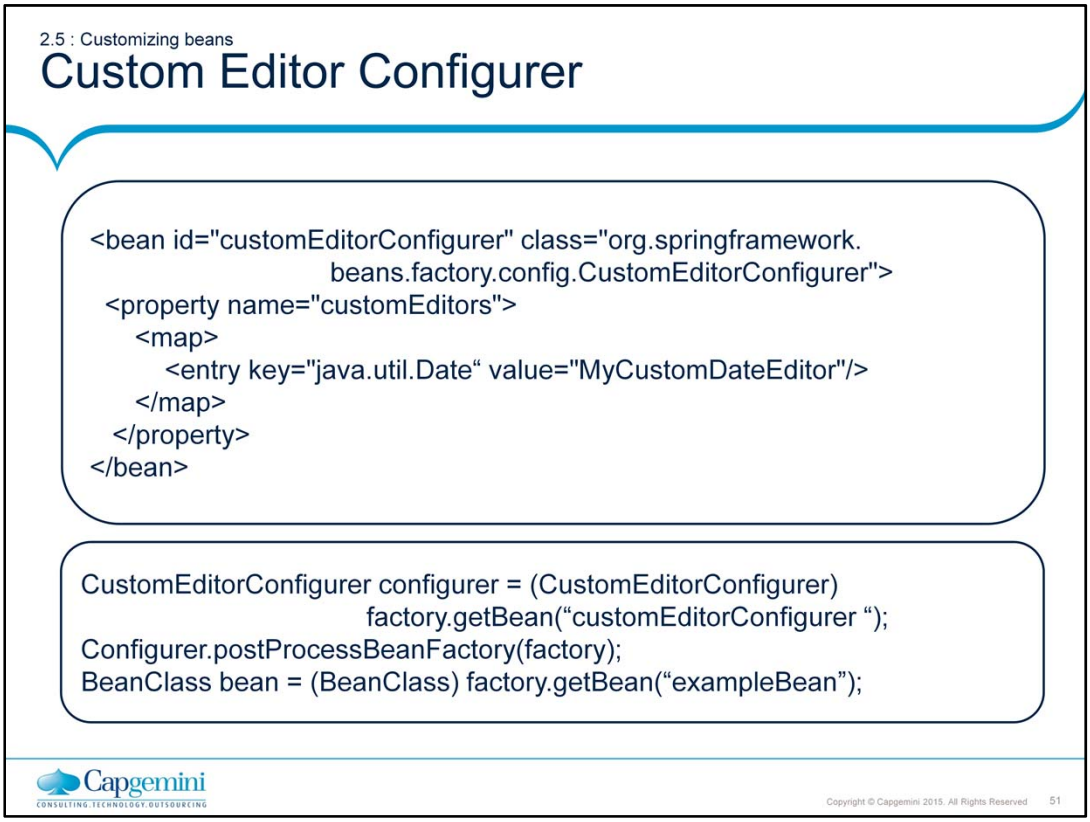
Whenever setter is called, the listener (PropertyPlaceholderConfigurer) is invoked and it will look into the properties file, retrieve values, place them in the place holders and initialize.
If instead of application context, bean factory is used, then the listener would have to be explicitly registered as shown below (also in the commented out code in userClient.java file).

```
XmlBeanFactory factory = new XmlBeanFactory(new
FileSystemResource("user.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("user.properties"));
cfg.postProcessBeanFactory(factory);
```



CustomEditorConfigurer : Lets you register custom implementation of java.beans.PropertyEditor to translate property wired values to other property types. This is a bean factory post-processer which allows to convert values in String form to final property values.

So far, we have seen several examples in which a complex property is set with a simple String value. When setting bean properties as a string value, a BeanFactory ultimately uses standard java.beans. PropertyEditor interface to convert these strings to the complex type of the property. There is a convenience implementation of the above interface called java.beans.PropertyEditorSupport,that has two methods of interest to us:
getAsText() : returns the String representation of a property's value.
setAsText(String value) : sets a bean property value from the string value passed in.
If an attempt is made to set a non-string property to a string value, the setAsText() method is called to perform the conversion. Likewise, the getAsText() is called to return a textual representation of the property's value.
Spring comes with several custom editors based on PropertyEditorSupport. You can also write your own custom editor by extending the PropertyEditorSupport class.



Declarative registration process:
The custom PropertyEditors (MyCustomDateEditor in this case) is injected into the CustomEditorConfigurer class using the Map-typed customEditors property. A map can have multiple entries and each entry in the Map represents a single PropertyEditor with the key of the entry being the name of the class for which the PropertyEditors is used.
In the above first example, the key for the MyCustomDateEditor is java.util.Date, which signifies that this is the class for which the editor should be used.

Programmatic Registration Process:
The second code snippet shows a call to Configurer.postProcessBeanFactory(), which passes in the BeanFactory instance. This is another method of registering custom editors in Spring. You should call this before you attempt to access any beans that need to use the custom PropertyEditors.

However, adding a new PropertyEditor means changing the application code, whereas with the declarative mechanism, you can define your editors as beans, which means you can configure them using DI. Besides, using Application Context means no java code to use the declarative mechanism, which strengthens the argument against programmatic mechanism.

2.5 : Customizing beans

Demo: DemoSpring_7

- This demo shows how to use the CustomEditorConfigurer BeanFactoryPostProcessor

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

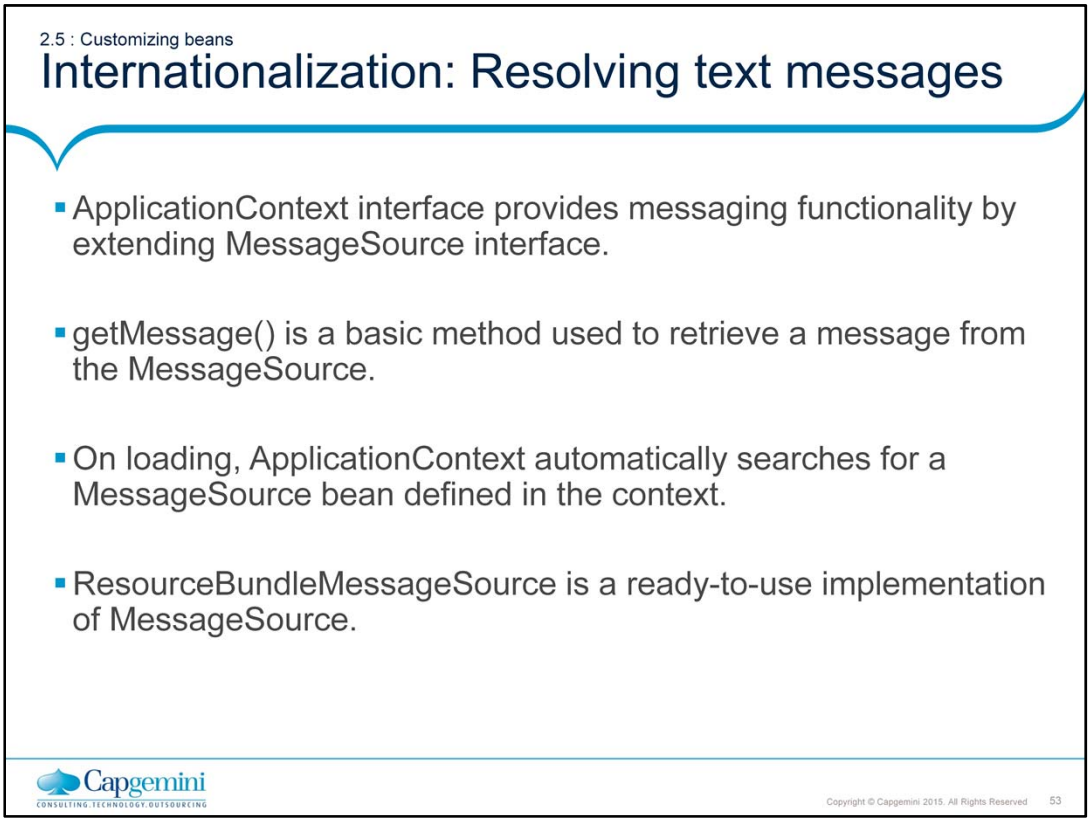
 52

Please refer to DemoSpring_7. The Employee.java is a POJO that holds the date property. Using basic wiring techniques learnt so far, you could set a value into Employee beans' date property. But we have created SQLDateEditor.java extending PropertyEditorSupport class.
Spring must recognize this custom property editor when wiring bean properties using Spring's CustomEditorConfigurer. This BeanFactory PostProcesser internally loads custom editors into the BeanFactory by calling the registerCustomEditor() method. By adding the following bit of XML into the bean configuration file, you will tell Spring to register the SQLDateEditor as a custom editor.

```
<bean id="customEditorConfigurer“ class="org. springframework. beans.
 factory.config.CustomEditorConfigurer">
      <property name="customEditors">
          <map> <entry key="java.sql.Date">
                    <bean class="training.spring.SQLDateEditor" /></entry>
           </map>
       </property>
</bean>
```

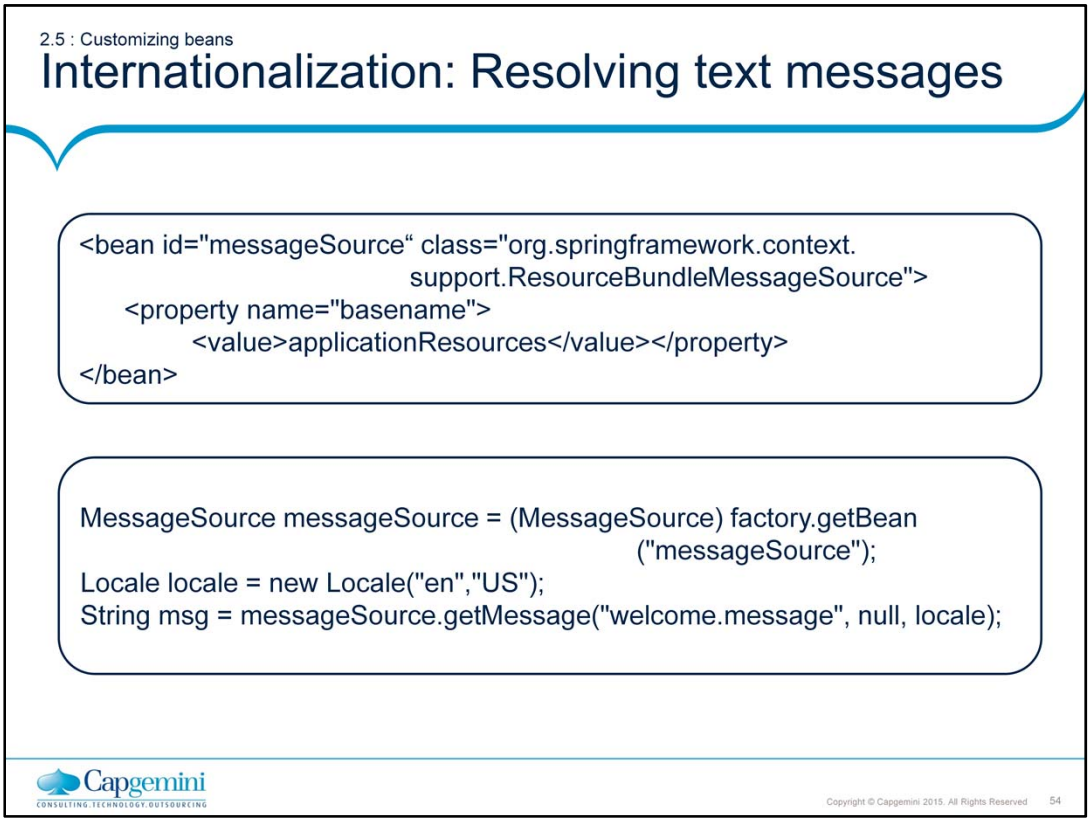
You will now be able to configure the Employee objects date property using a simple string value:

```
<bean id="employee“   class="training.spring.Employee">
       <property name="date"><value>2006-01-01</value></property>
</bean>
```



Many times you may not want to hard-code certain text that will be displayed to the user. This may be because text is subject to change or perhaps your application will be internationalized and you will display text in the user's native language.
Java's support for parameterization and internationalization of messages enables you to define one or more properties files that contain the text that is to be displayed in your application. There should always be a default message file along with optional language-specific message files. For ex: if name of the application's message bundle is "MsgText", you may have the following set of message property files:
MsgText.properties: Default messages when a locale cannot be determined or locale-specific properties file is not available.
MsgText_en_US.properties: text for English speaking users in US.

Spring's ApplicationContext supports parameterized messages by making them available to the container through the MessageSource interface that has a single method – getMessage(). Spring comes with a ready-to-use implementation of MessageSource – the ResourceBundleMessageSource. This simply uses Java's own java.util. ResourceBundle to resolve messages. A ResourceBundleMessageSource works on a set of properties files that are identified by base names. When looking for a message for a particular Locale, the ResourceBundle looks for a file that is named as a combination of base name and the Locale name. For e.g. : applicationResources_fr will match a French locale.



To use ResourceBundleMessageSource, add the above xml entry (the first code snippet) to the bean wiring file.

It is very important that this bean be named messageSource because the ApplicationContext will look for a bean specifically by that name when setting up its internal message source.

The basename property specifies the base name of the bundle. The bundle will normally look for messages in properties files with names that are variations of base name depending on locale.

You will never need to inject the messageSource bean into your application beans but will instead access messages via ApplicationContext's own getMessage() methods. For e.g. to retrieve the message whose name is ”welcome.message”, please refer to the second code snippet above.

You will likely be using parameterized messages in the context of a web application, displaying the text on a web page. In that case, you can use Spring's <spring:message> jsp tag to retrieve messages and need not need directly access the ApplicationContext.

< spring:message code=“welcome.message” />

Alternatively, you can use the second argument of the getMessage() to pass in an array of arguments that will be filled in for params within the message

2.5 : Customizing beans

Demo SpringI 18N

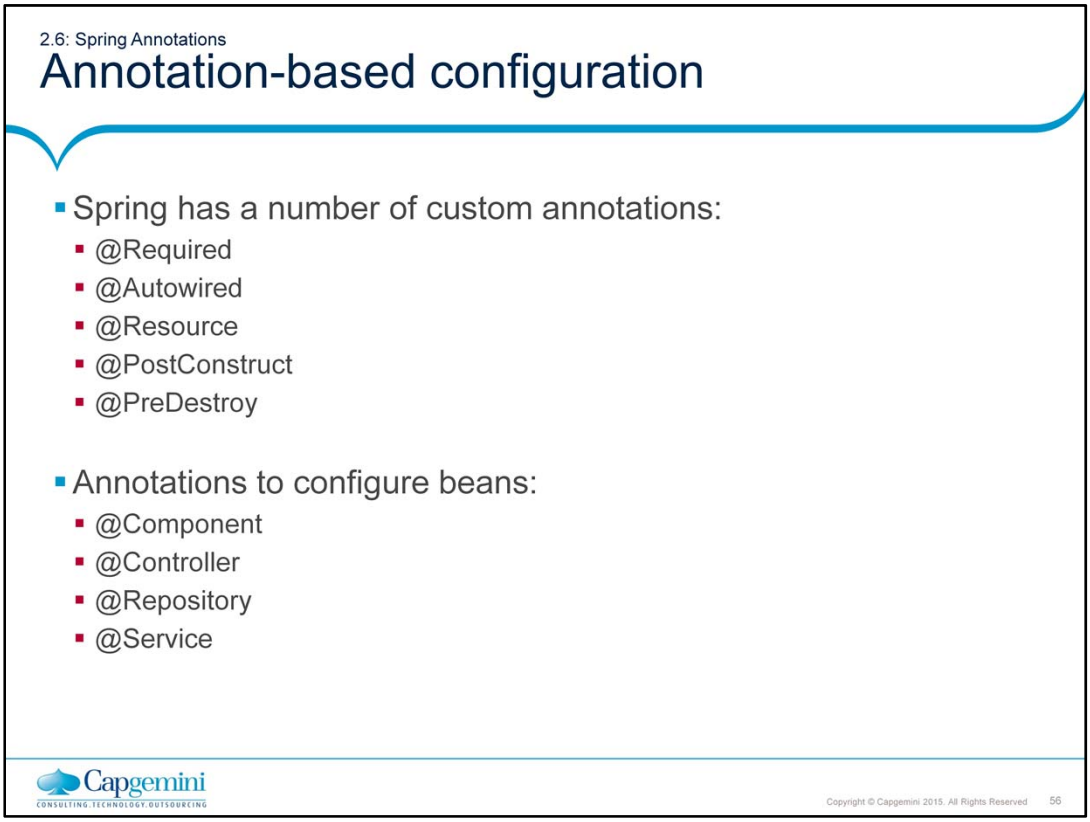
- This demo shows how to provide messaging functionality in the application context.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

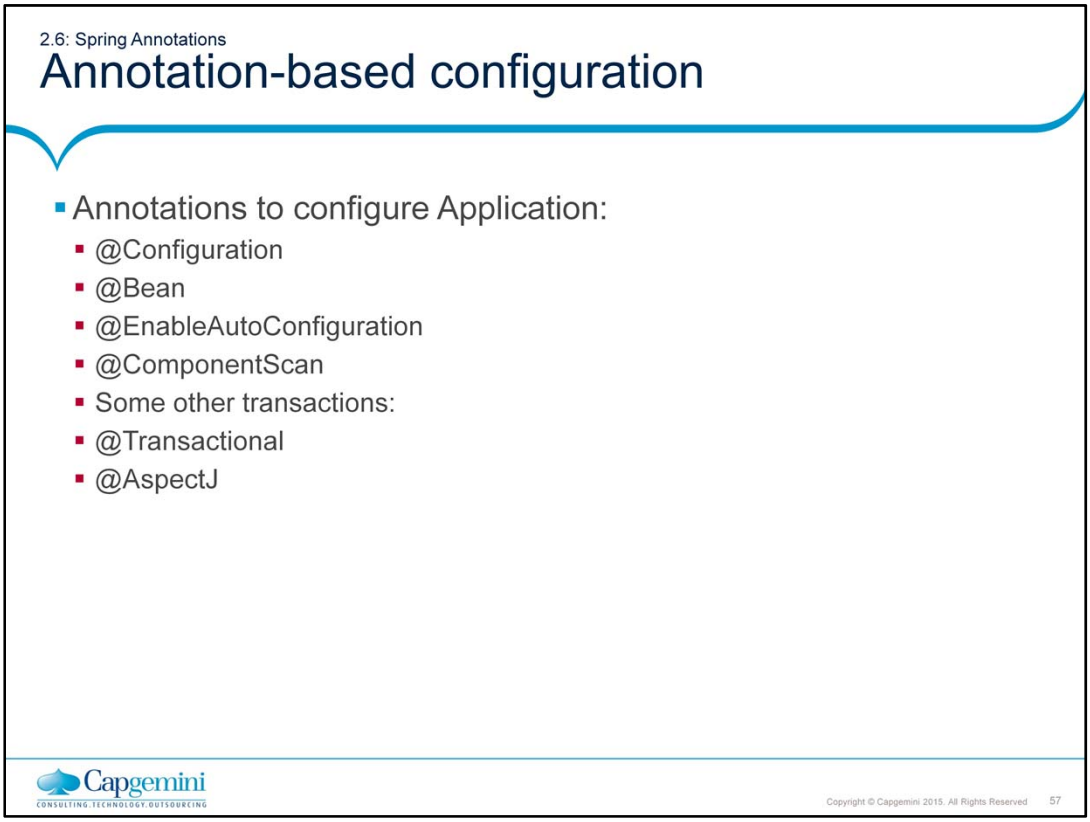
 55

Please refer to DemoSpringI18N. There are two resource bundles defined :
applicationResources_en_GB.properties
applicationResources_en_US.properties
Message source has been defined in message.xml
MessageClient.java creates a new locale object and uses the getMessage() to access messages from the appropriate resource bundle based on locale. Notice that the message is parameterized :
welcome.message = Welcome {0}, in UK

Hence we need to send value for {0} parameter in this manner:
String msg = messageSource.getMessage("welcome.message", new Object[]{"Majrul"},locale);
This will set the value of the {0} parameter to “Majrul”. We can pass in many parameters for a single message, by using this object array in the getMessage() method.



So far, we have configured our beans in the XML configuration files. Starting with Spring 2.0, we can use annotations to configure our beans. By annotating your classes, you state their typical usage or stereotype. Spring has a number of custom (Java5+) annotations.
@Required : The @Required annotation is used to specify that the value of a bean property is required to be dependency injected. That means, an error is caused if a value is not specified for that property
@Autowired: Prior to Spring 2.5, autowiring could be configured for a number of different approaches: constructor, setters by type, setters by name, or autodetect – which offer a large degree of flexibility, but not very fine-grained control. For eg, it has not been possible to autowire a specific subset of an object's setter methods or to autowire some of its properties by type and others by name.
By using @Autowired, you can eliminate the additional XML in your configuration file that specifies the relationship between two objects. Also, you no longer need methods to set the property in the owning class. Just include a private or protected variable and Spring will do the rest.
@Resource : Declares a reference to a resource such as a data source, Java Messaging Service (JMS) destination, or environment entry. This annotation is equivalent to declaring a resource-ref, message-destination-ref, env-ref, or resource-env-ref element in the deployment descriptor.



So far, we have configured our beans in the XML configuration files. Starting with Spring 2.0, we can use annotations to configure our beans. By annotating your classes, you state their typical usage or stereotype. Spring has a number of custom (Java5+) annotations.
@Required : The @Required annotation is used to specify that the value of a bean property is required to be dependency injected. That means, an error is caused if a value is not specified for that property
@Autowired: Prior to Spring 2.5, autowiring could be configured for a number of different approaches: constructor, setters by type, setters by name, or autodetect – which offer a large degree of flexibility, but not very fine-grained control. For eg, it has not been possible to autowire a specific subset of an object's setter methods or to autowire some of its properties by type and others by name.
By using @Autowired, you can eliminate the additional XML in your configuration file that specifies the relationship between two objects. Also, you no longer need methods to set the property in the owning class. Just include a private or protected variable and Spring will do the rest.
@Resource : Declares a reference to a resource such as a data source, Java Messaging Service (JMS) destination, or environment entry. This annotation is equivalent to declaring a resource-ref, message-destination-ref, env-ref, or resource-env-ref element in the deployment descriptor.

- **@PostConstruct :** Specifies a method that the container will invoke after resource injection is complete but before any of the component's life-cycle methods are called.
- **@PreDestroy :** Specifies a method that the container will invoke before removing the component from service.

When Spring discovers one of these annotations, it creates the appropriate bean by matching the stereotype.

Annotations to configure beans:

@Component: Is the basic stereotype. Classes annotated with this will become spring beans.

@Controller: Classes annotated with this annotation will be considered as a Controller in Spring MVC support.

@Repository: Classes with @Repository annotation represent a repository (eg a data access object)

@Service: This annotation marks classes that implement a part of the business logic of the application.

Annotations to configure application

@Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions.

@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.

@**EnableAutoConfiguration** annotation will trigger automatic loading of all the beans the application requires

@ComponentScan : An equivalent for Spring XML's <context:component-scan/> is provided with the @ComponentScan annotation.

Some other annotations:

@Transactional: is an alternative to the XML-based declarative approach to transaction configuration. All methods of a Spring bean instantiated from a class with the @Transactional annotation will be transactional (thus indirectly, all methods will execute in a transaction). The functionality offered by the @Transactional annotation and the support classes is only available to you if you are using at least Java 5

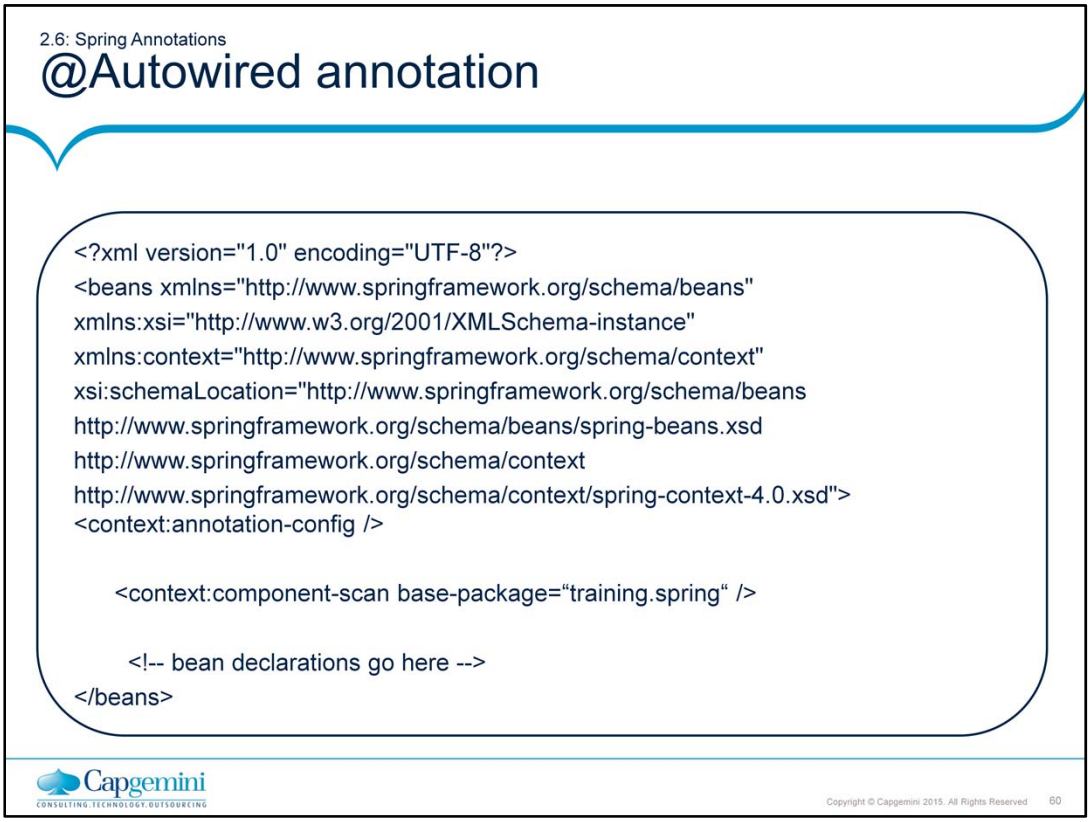
@Aspect : This annotation on a class marks it as an aspect along with @Pointcut definitions and advice (@Before, @After, @Around)

We shall be covering each of these as we move forwards into AOP, database programming and web applications.

Notice that in all the XML configurations so far, we have used the beans namespace only. But the core Spring Framework comes with ten configuration namespaces as shown in next slide.

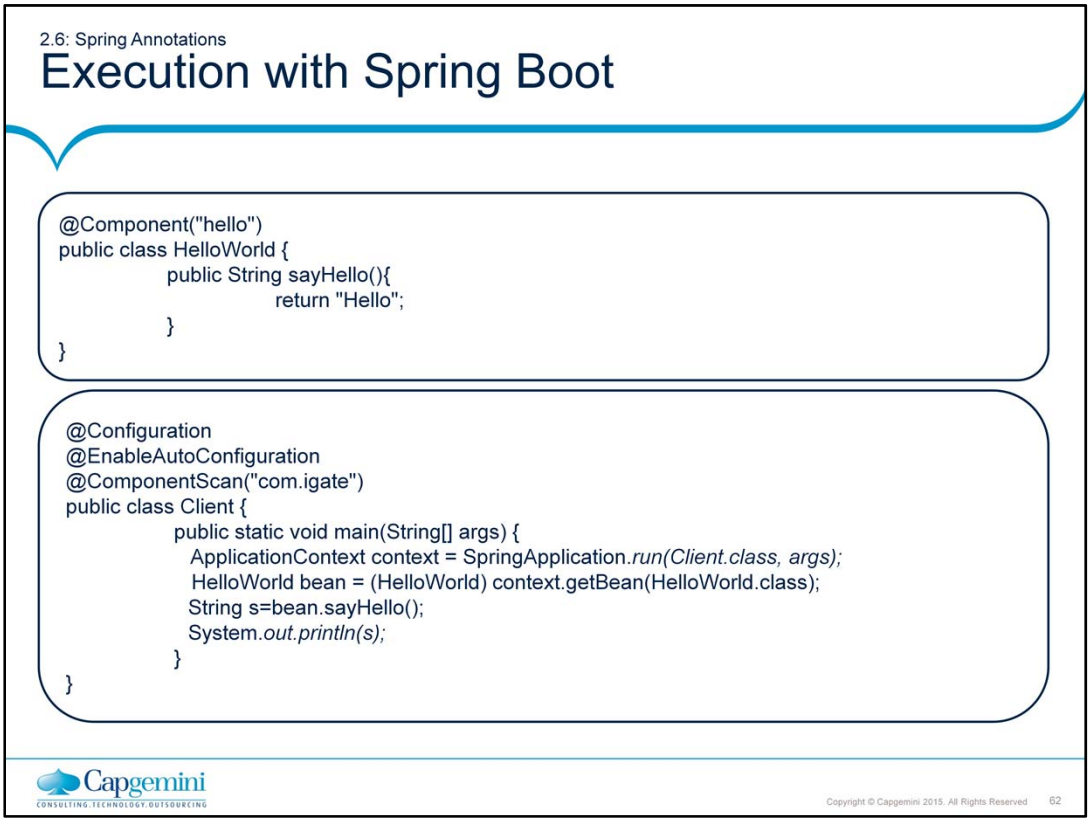
XML namespaces through which you can configure the Spring container

Name	Purpose
aop	Provides elements for declaring aspects and for automatically proxying @AspectJ annotated classes as Spring aspects.
beans	The core primitive Spring namespace, enabling declaration of beans and how they should be wired.
context	Comes with elements for configuring the Spring application context, including the ability to autodetect and autowire beans and injection of objects not directly managed by Spring.
jee	Offers integration with Java EE APIs such as JNDI and EJB.
jms	Provides configuration elements for declaring message-driven POJOs.
lang	Enables declaration of beans that are implemented as Groovy, JRuby, or BeanShell scripts.
mvc	Enables Spring MVC capabilities such as annotation-oriented controllers, view controllers, and interceptors.
oxm	Supports configuration of Spring's object-to-XML mapping facilities.
tx	Provides for declarative transaction configuration.
util	A miscellaneous selection of utility elements. Includes the ability to declare collections as beans and support for property placeholder elements.



We can use annotations to automatically wire bean properties. It is similar to autowire attribute in configuration file, but allows more fine-grained autowiring, where you can selectively annotate certain properties for autowiring.
However, simply annotating your classes is not enough to get an annotation's behavior. You need to enable a component that is aware of the annotation and that can process it appropriately. This component (a special BeanPostProcessor implementation) will be different for all annotations. For eg, the RequiredAnnotationBeanPostProcessor class is necessary for @Required annotation.
Annotation wiring isn't turned on in the Spring container by default. So, before we can use annotation-based autowiring, we'll need to enable it in our Spring configuration. The simplest way to do that is with the <context:annotation-config> element from Spring's context configuration namespace.
<context:annotation-config> tells Spring that you intend to use annotation-based wiring in Spring. Once it's in place you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. The implicitly registered post-processors include AutowiredAnnotationBeanPostProcessor, CommonAnnotationBeanPostProcessor, PersistenceAnnotationBeanPostProcessor, as well as the RequiredAnnotationBeanPostProcessor.

So far, we have seen how <context: annotation-config> eliminates most uses of <property> and <constructor-arg> elements from your Spring configuration. But we still need to explicitly declare beans using <bean>.
The <context:component-scan> element is similar to <context:annotation-config>. But it also configures Spring to automatically discover beans and declare them, that is, most of the beans in your Spring application can be declared and wired without using <bean>.
The <context:component-scan> element scans a package and all of its subpackages, looking for classes that could be automatically registered as beans in the Spring container. The base-package attribute tells <context:component-scan> package to start its scan.
How does this work? By default, <context:component-scan> looks for classes that are annotated with one of the following: @Component, @Controller, @Repository, @Service.
For example, suppose that our application context only has the currencyConverter and exchangeService beans in it. We can eliminate the explicit <bean> declarations from the XML configuration by using <context:component-scan> and annotating the CurrencyConverterImpl and ExchangeServiceImpl classes with @Component.



We can use annotations to automatically wire bean properties. It is similar to autowire attribute in configuration file, but allows more fine-grained autowiring, where you can selectively annotate certain properties for autowiring.
However, simply annotating your classes is not enough to get an annotation's behavior. You need to enable a component that is aware of the annotation and that can process it appropriately. This component (a special BeanPostProcessor implementation) will be different for all annotations. For eg, the RequiredAnnotationBeanPostProcessor class is necessary for @Required annotation.
Annotation wiring isn't turned on in the Spring container by default. So, before we can use annotation-based autowiring, we'll need to enable it in our Spring configuration. The simplest way to do that is with the <context:annotation-config> element from Spring's context configuration namespace.
<context:annotation-config> tells Spring that you intend to use annotation-based wiring in Spring. Once it's in place you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. The implicitly registered post-processors include AutowiredAnnotationBeanPostProcessor, CommonAnnotationBeanPostProcessor, PersistenceAnnotationBeanPostProcessor, as well as the RequiredAnnotationBeanPostProcessor.

2.6: Spring Annotations

DemoSpring_Anno

- This demo illustrates autowired annotation

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 63

Please refer to demo, DemoSpring_Anno, This example uses the anno.xml for configuring the beans.

Notice how the CurrencyConverterImpl.java depends on the com.igate.anno.ExchangeServiceImpl.java class for services. Notice how the exchangeService property has been annotated with @Autowired. The configuration file now just contains beans declaration and no instructions on wiring!

2.6: Spring Annotations

Annotating beans for autodiscovery

- Refer to demos, DemoSpring_Anno

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

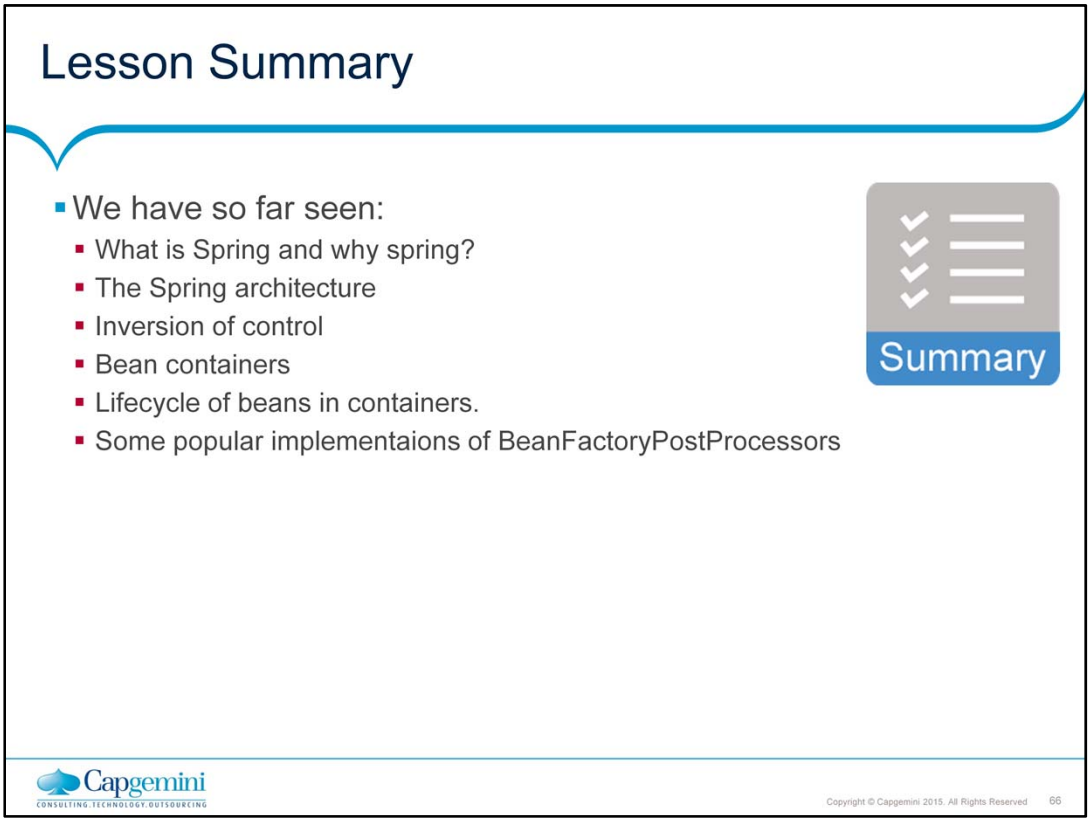
 64

Refer to demo, DemoSpring_Anno , Notice the anno.xml configuration file. It contains no beans! Notice the bean classes themselves are annotated with @Component, @PostConstruct, @Autowired etc.

Lab

- From the lab guide
  - Lab-1 problem-statement-1 2 and 3

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



Thus we have seen that Spring is the most popular and comprehensive of the lightweight J2EE frameworks that have gained popularity since 2003.
We saw how Spring is designed to promote architectural good practice. A typical spring architecture will be based on programming to interfaces rather than classes.
We have seen what is Inversion of control and dependency injection.
We also saw Bean containers and lifecycle of beans in containers. We saw how to hook into the lifecycle of a bean and make it aware of the Spring environment.

Review Questions

- Question 1: The <constructor-arg> element has an optional _____attribute that specifies the ordering of the constructor arguments.
  - Option 1: By index
  - Option 2: By type
  - Option 3: By order

- Question 2: A _____ bean lets the container return a new instance each time a bean is asked for in a non-web application
  - Option 1: Singleton
  - Option 2: Prototype
  - Option 3: Request
  - Option 4: session

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 67

Review Questions

- Question 3: Specifying the _____ tag will allow Spring to validate at deployment time that the other bean actually exists.
  - Option 1: idref
  - Option 2: ref
  - Option 3: local

- Question 4: The BeanPostProcessor performs post processing on the entire Spring container.
  - Option 1: True
  - Option 2: false

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 68

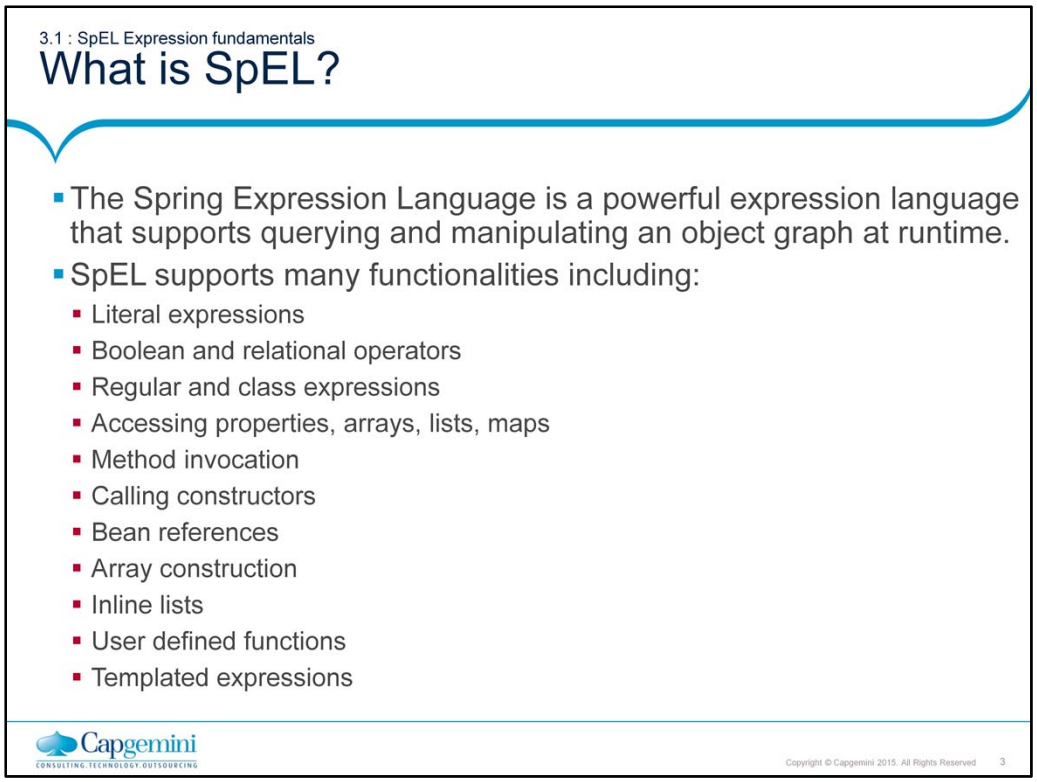
Basic Spring 4.0

Lesson 3: Spring Expression Language (SpEL)

Lesson Objectives

- Introduction to SpEL (Spring Expression Language)
  - SpEL Expression fundamentals
  - Expression Language features
  - Reduce configuration with @Value

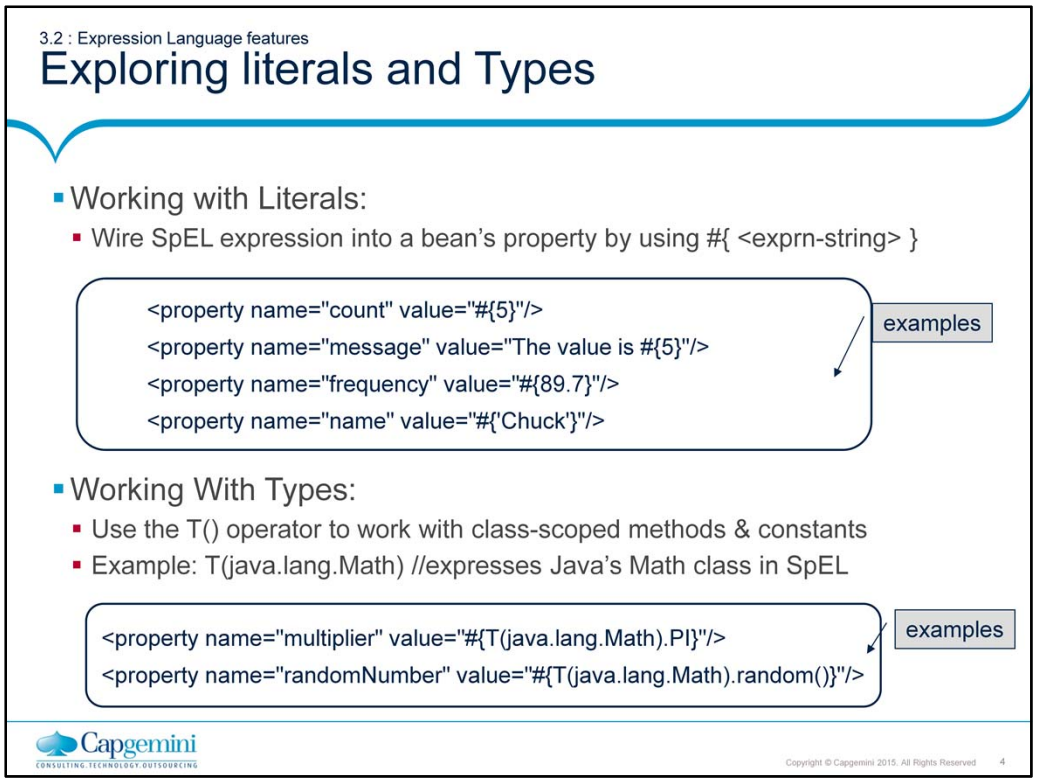
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



So far we have seen how to wire dependencies using setter and constructor injections. But these have been statically defined in the Spring configuration file. When we wired the exchangeService property into the CurrencyConverter bean, that value was determined at development time. Likewise, when we wired references to other beans, those references were also statically determined in the Spring configuration.

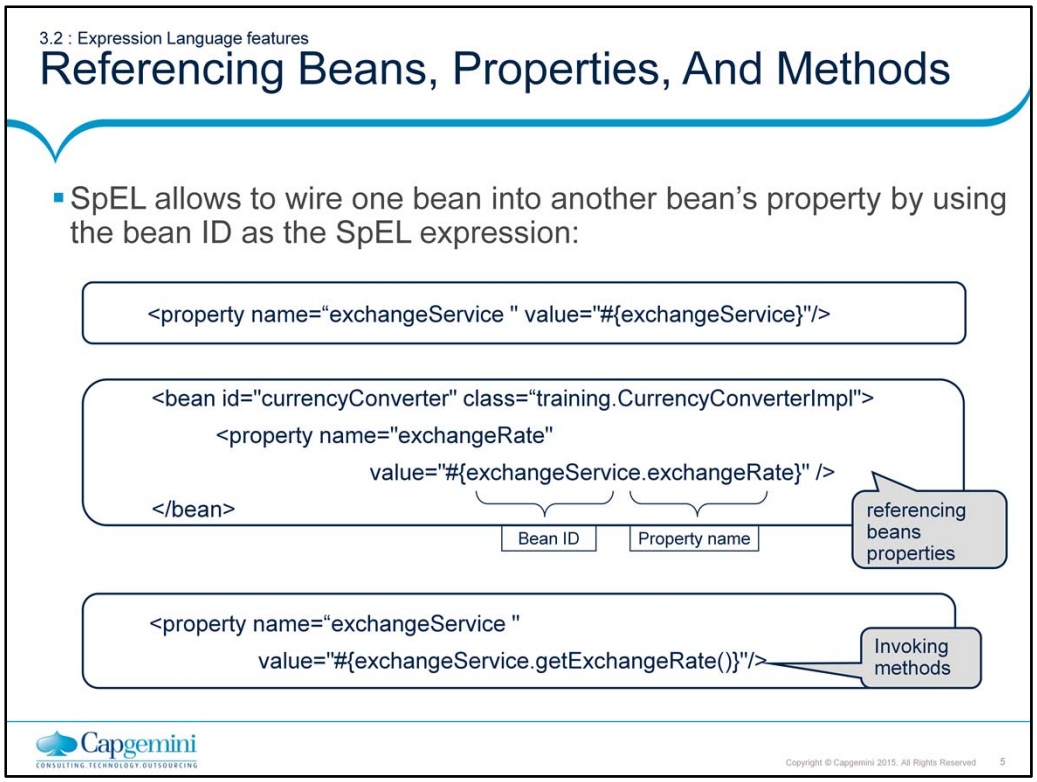
What if we want to wire properties with values at runtime? Spring 3's Spring Expression Language (SpEL) is a powerful way of wiring values into a bean's properties or constructor arguments using expressions that are evaluated at runtime. SpEL syntax is similar to Unified EL but offers additional features like method invocation and basic string templating functionality.

SpEL is based on a technology agnostic API allowing other expression language implementations to be integrated should the need arise. It thus can be used independently. It supports many functionalities as listed above.

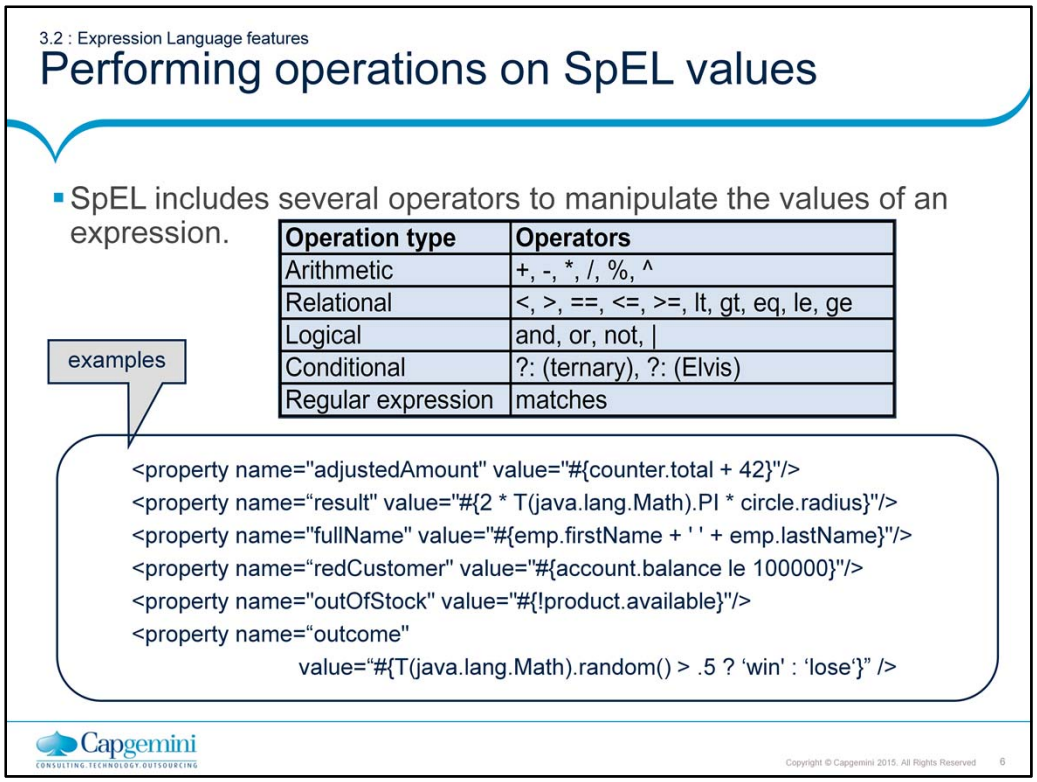


Literal Values: SpEL expressions, like any other expression, are evaluated for some value. SpEL can evaluate literal values, references to a bean's properties, a constant on some class etc. The simplest SpEL expression for example is 5, which evaluates to an integer value of 5. We can wire this value into a bean's property by using #{ } markers in a <property> element's value attribute, as shown in example above. The #{ } markers indicate that the content that they contain is a SpEL expression.
Similarly, see example above for expressing Floating-point numbers. Literal String values can be expressed in SpEL with either single or double quote marks. You can also use Boolean true and false values. Eg- <property name="enabled" value="#{false}"/>

Working With Types: The result of the T() operator is a Class object that represents the given class. The T() operator thus gives us access to static methods and constants on a given class. Eg, to wire the value of pi into a bean property, use:
<property name="multiplier" value="#{T(java.lang.Math).PI}"/>
Likewise, static methods can also be invoked on the result of the T() operator. Eg, to wire a random number (between 0 and 1) into a bean property, use:
<property name="randomNumber" value="#{T(java.lang.Math).random()}"/> When the application is starting up and Spring is wiring the randomNumber property, it'll use the Math.random() method to determine a value for that property!



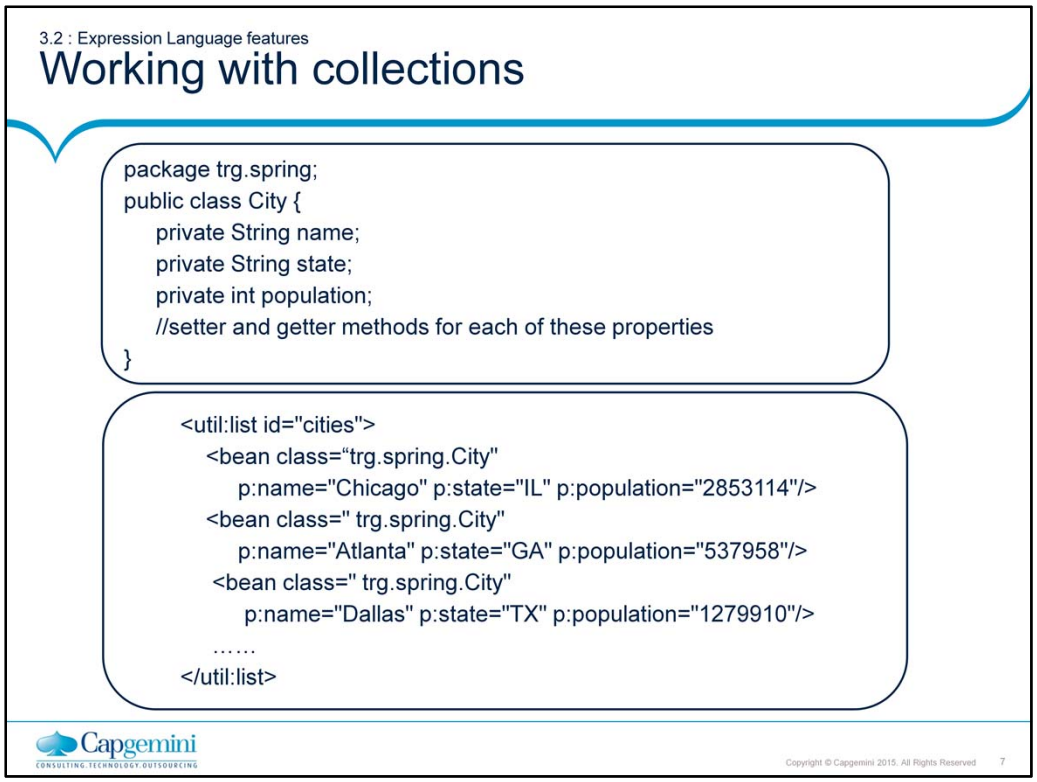
A SpEL expression can reference another bean by its ID. See first example above. We used SpEL to wire the bean whose ID is “exchangeService” into an exchangeService property. We can do this by using the ref attribute too! <property name="exchangeService" ref="exchangeService"/>The outcome is the same. But let us see how to take advantage of being able to wire bean references with SpEL.
The second example configures a new CurrencyConverterImpl bean whose ID is currencyConverter. This is wired to whatever exchangeRate the exchangeService bean provides. This is equivalent to:
CurrencyConverterImpl currencyConverter = new CurrencyConverterImpl ();
currencyConverter.setExchangeRate(exchangeService.getExchangeRate());



Like in Java, + operator is overloaded to perform concatenation on String values. We know that the less-than (<) and greater-than (>) operators are used to compare different values. Unfortunately, they pose a problem when using these expressions in Spring's XML configuration (since they have special meaning in XML). So, when using SpEL in XML,5 it's best to use SpEL's textual alternatives like le (<), gt (>) etc.

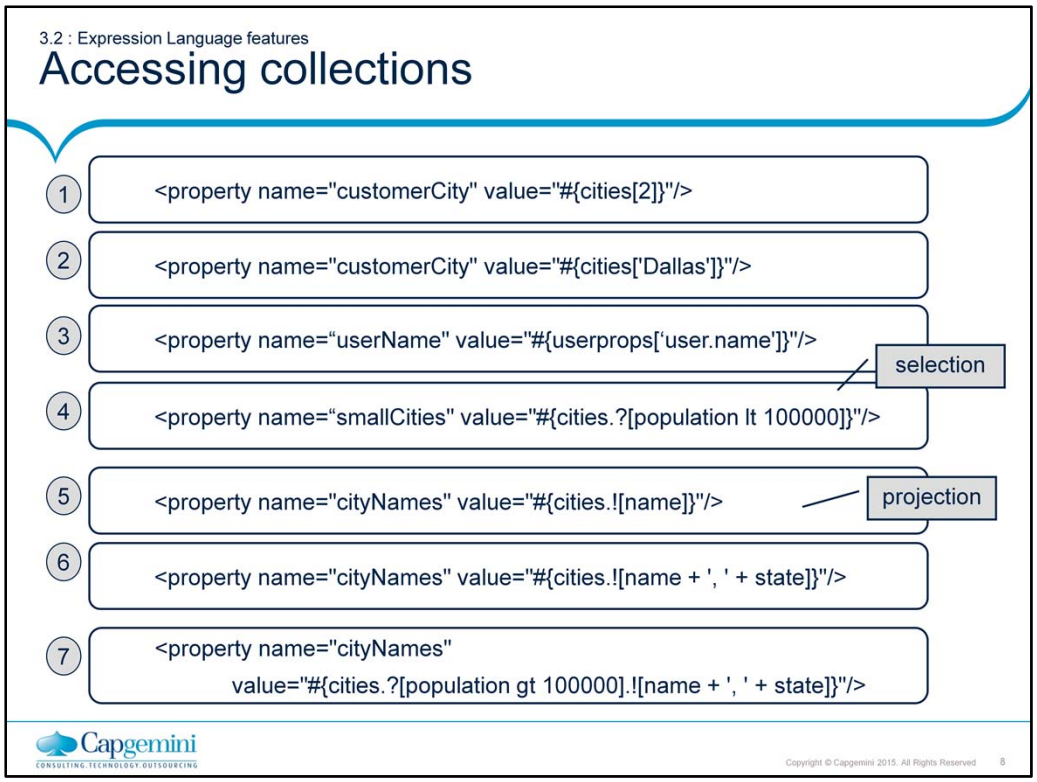
SpEL supports regular expressions using the matches operator, which is a relational operator returning true or false. Example:

```
<util:map id="regExpsSamples" value-type="java.lang.Object"
                                    key-type="java.lang.String">
  <entry key="# {'a string' matches 'a.*'}"
         value="#{'a string' matches 'b.*'}"/>
</util:map>
```



Let us digress a bit and see how Spring allows us to create collection via configuration file.
From Spring 2.5 onwards, in addition to standard <property> tag, you can use the p namespace to set properties of beans.
The <util:list> element comes from Spring's util namespace. It creates a bean of type java.util.List that contains all of the values or beans that it contains. In this case, that's a list of City beans.
Example for creating Map:

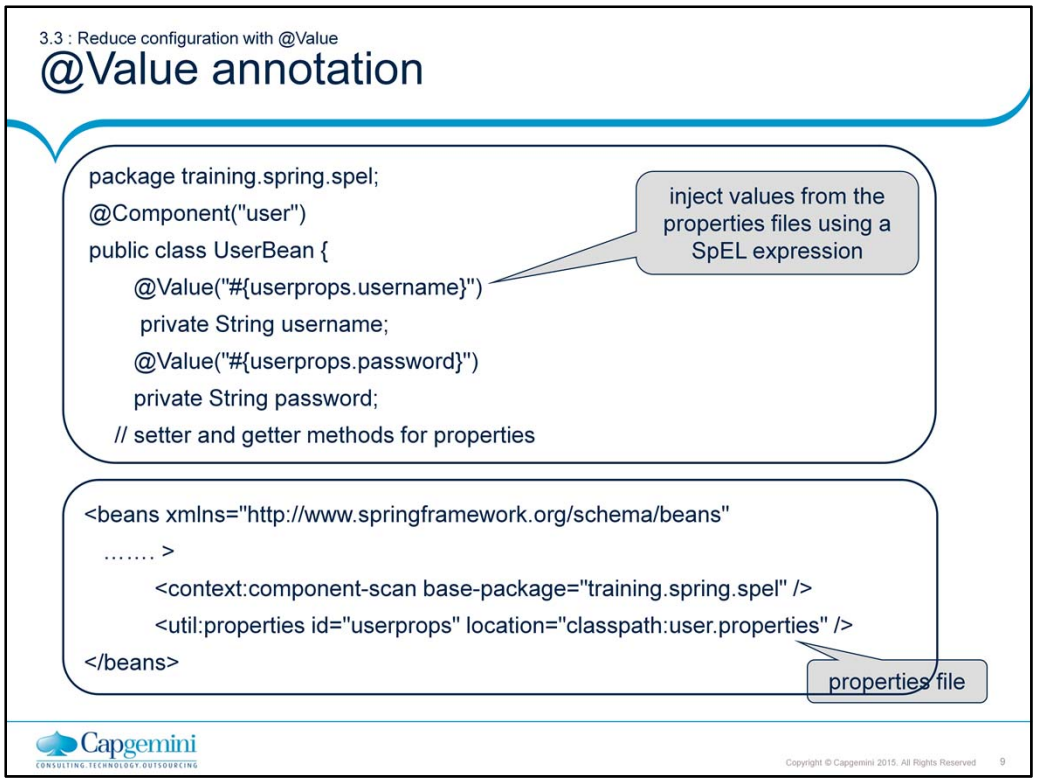
```
<util:map id="numbersMap" value-type="java.lang.Integer"
                                key-type="java.lang.String">
  <entry key="one" value="1"/>
  <entry key="two" value="2"/>
  <entry key="three" value="3"/>
  <entry key="four" value="4"/>
  <entry key="five" value="5"/>
</util:map>
```



Example -1 : This selects the third city from the list to assign to customerCity
Example -2: Assuming that cities is a java.util.Map collection, with city-name as the key. The example shows how to retrieve the entry for Dallas.
Example -3: We can use the [] operator is to retrieve a value from a java.util.Properties collection too. First: load a properties configuration file into Spring using the <util:properties> element as follows:

```
<util:properties id="userprops" location="classpath:user.properties"/>
```

The userprops bean is a java.util.Properties that contains all of the entries in the file named user.properties. Accessing a property from that file is similar to accessing a member of a Map. The 3rd example above reads a property whose name is user.name from the userprops bean
Example -4: We would like a list of cities whose population is less than 100,000. Use the selection operator (.?[]) when doing the wiring. The selection operator creates a new collection whose members include only those members from the original collection that meet the criteria expressed between the square braces. In this case, the smallCities property will be wired with a list of City objects whose population property is less than 100,000.
Example -5: Projecting collections means collecting a particular property from each of the members of a collection into a new collection. Use SpEL's projection operator (.![]) to do this. Example-5 retrieves a list of city names from the collection of City objects. You can also retrieve multiple members as the next example shows.
The final example (7) is a combination of selection and projection.



SpEL combined with @Value annotation is great. We have already seen how component scan and autowiring reduces the size of XML configuration files. However, we still have to deal with beans that need literal values as input. With @Value you can inject values from your properties files using SpEL. Assume that you have a properties file configured as shown above.

To use SpEL in annotation, you must register your component via annotation. If you register your bean in XML and define @Value in Java class, the @Value will fail to execute.

Demo : SpringDemo_SPEL

- Using SpEL

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 10

Please refer to demo, SpringDemo_SPEL

Lesson Summary

- We have so far seen:
  - SpEL Expression fundamentals
  - Expression Language features
  - Reduce configuration with @Value

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 11

Review Questions

- Question 1: _____ markers indicate that the content that they contain is a SpEL expression.
  - Option 1: ${ }
  - Option 2: #{ }
  - Option 3: %{ }

- Question 2: SpEL can access instances of java.lang.Class using the _____ operator
  - Option 1: #{ }
  - Option 2: T
  - Option 3: Type

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 12

Review Questions

- Question 3: The _____ effectively creates a bean of type java.util.Map that contains all of the values or beans that it contains .
  - Option 1: <util:list>
  - Option 2: <util:properties>
  - Option 3: <util:map>

- Question 4: If @Value annotation is used in a component, it is mandatory that the component be annotated with @Component
  - Option 1: True
  - Option 2: False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 13

Basic Spring 4.0

Lesson 4: Spring MVC framework

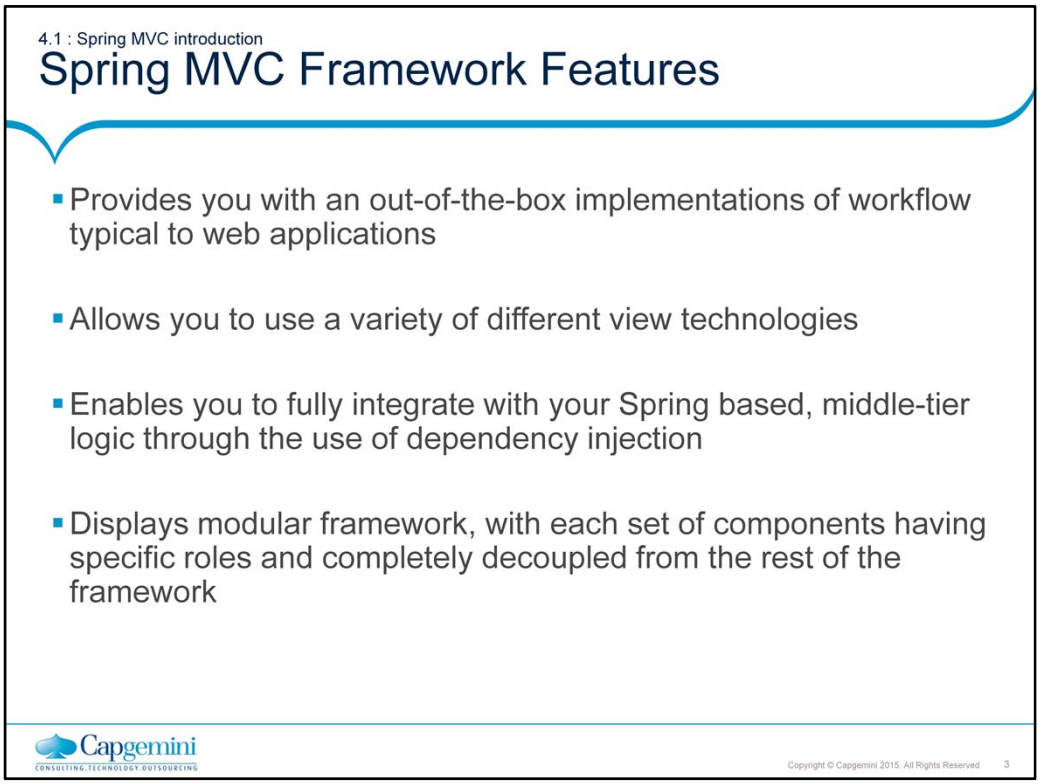
Lesson Objectives

- Introduction to Spring MVC framework
  - Learn how to develop web applications using Spring
  - Understand the Spring MVC architecture and the request cycle of Spring web applications
  - Understand components like handler mappings, ViewResolvers and controllers
  - Use MVC Annotations like @Controller, @RequestMapping and @RequestParam
  - Introduction to REST web Services
  - REST Controllers on the top of MVC

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

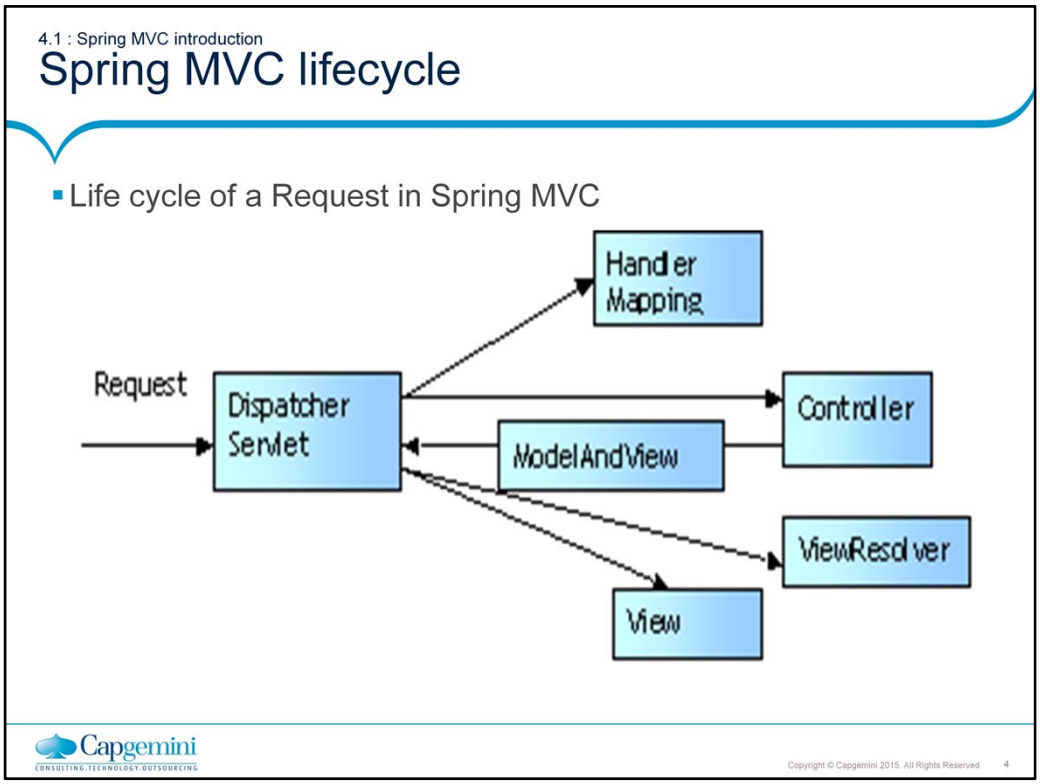
 2

Web applications have become a very important part of any enterprise system. The key requirements for a web framework is to simplify development of the web tier as much as possible. Spring provides a web framework based on the MVC (Model view Controller) paradigm. Although it is similar in some ways to other popular MVC frameworks such as Struts and WebWork, Spring web MVC provides significant advantages over those frameworks.
Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in separating the business logic, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.



The Spring MVC is a web framework built within the Spring Framework. There are a number of challenges while creating a web-based application - like state management, workflow and validation, which are addressed by Spring's web framework. This framework can be used to automatically populate your model objects from incoming request parameters while providing validation and error handling as well. The entire framework is modular, with each set of components having specific roles and completely decoupled from the rest of the framework. This allows you to develop the front end of your web application in a very pluggable manner.

MVC provides out-of-the-box implementations of workflow typical to web applications. Its highly flexible, allowing you to use a variety of different view technologies. It also enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection.
You can use Spring web MVC to make services that are created with other parts of Spring available to your users, by implementing web interfaces.



Life cycle of a request in Spring MVC: From the time that a request is received by Spring until the time that a response is returned to the client, many pieces of Spring MVC framework are involved.

The process starts when a client (typically a web browser) sends a request. It is first received by a DispatcherServlet. Like most Java-based MVC frameworks, Spring MVC uses a front-controller servlet (here DispatcherServlet) to intercept requests. This in turn delegates responsibility for a request to other components of an application for actual processing.

The Spring MVC uses a Controller component for handling the request. But a typical application may have several controllers. To determine which controller should handle the request, DispatcherServlet starts by querying one or more HandlerMappings. A HandlerMapping typically maps URL patterns to Controller objects.

Once the DispatcherServlet has a Controller object, it dispatches the request to the Controller which performs the business logic (a well-designed Controller object delegates responsibility of business logic to one or more service objects). Upon completion of business logic, the Controller returns a ModelAndView object to the DispatcherServlet. The ModelAndView object contains both the model data and logical name of view. The DispatcherServlet queries a ViewResolver with this logical name to help find the actual JSP. Finally the DispatcherServlet dispatches the request to the View object, which is responsible for rendering a response back to the client.

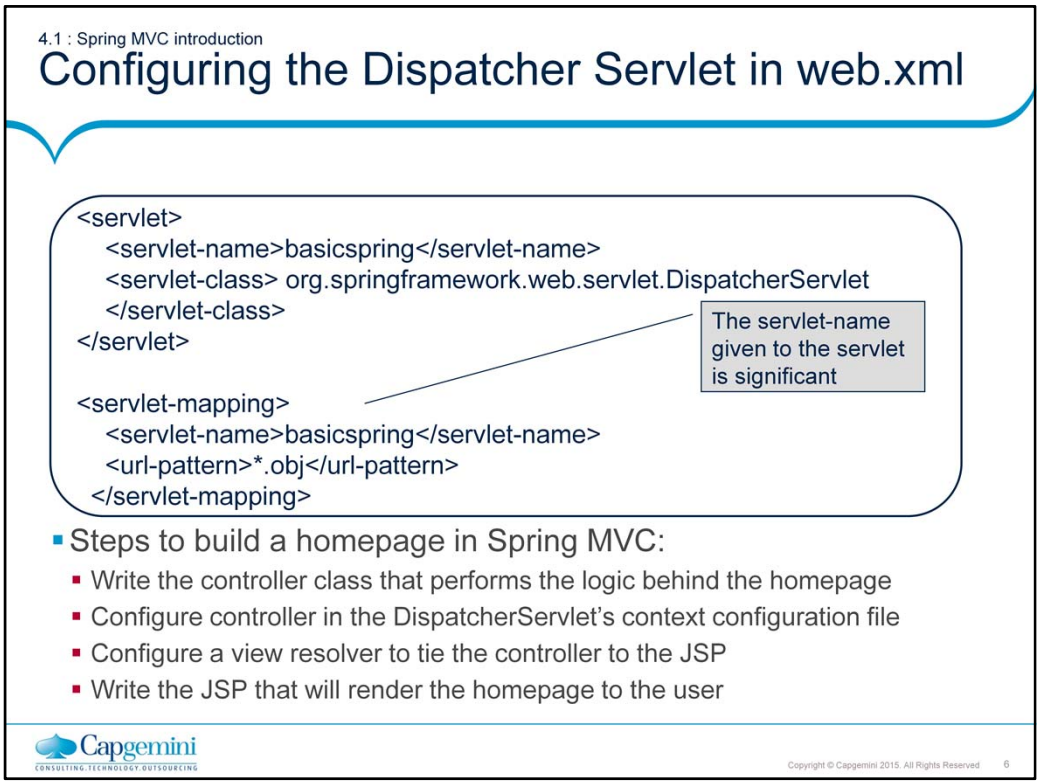
4.1 : Spring MVC introduction

Dispatcher Servlet

- The central component of Spring MVC is DispatcherServlet .
- It acts as the front controller of the Spring MVC framework
- Every web request must go through it so that it can manage the entire request-handling process.

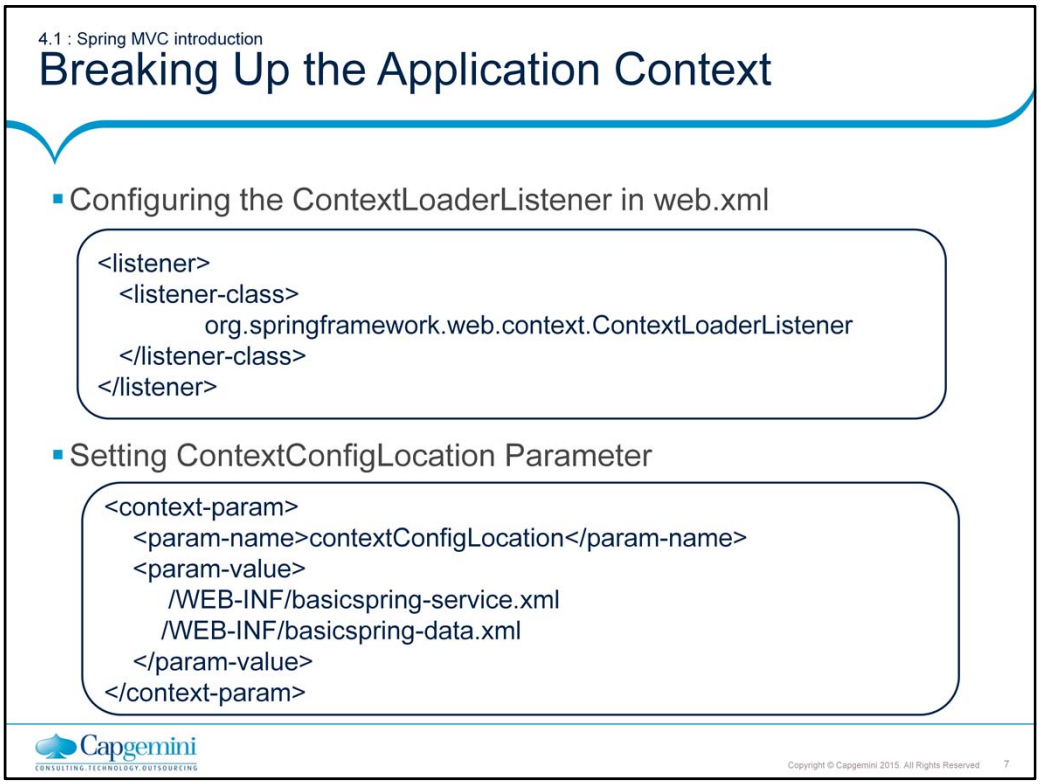
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 5



Configuring the DispatcherServlet
The dispatcher servlet is at the heart of the Spring MVC and functions as Spring MVC's front controller. Like any other servlet, it must be configured in web.xml file. Place the <servlet> declaration (in the first listing above) in the web.xml file.
The servlet-name given to the servlet is significant. By default, when DispatcherServlet is loaded, it will load the Spring application context from an xml file whose name is based on the name of the servlet. In the above example, the servlet-name is basicspring and so the DispatcherServlet will try to load the application context from a file called basicspring-servlet.xml.
Next, you must indicate which URL's will be handled by DispatcherServlet. Add the following <servlet-mapping> tag (the second xml listing) to web.xml to let DispatcherServlet handle all url's that end in .obj. The URL pattern is arbitrary and could be any extension.
DispatcherServlet is now configured and ready to dispatch requests to the web layer of your application.

Pls. see the above steps that are followed while building a simpleMVC application. These steps are explained in detail in the coming slides.



We have seen that DispatcherServlet will load the Spring application context from a single XML file whose name is <servlet-name>-servlet.xml. But you can also split the application context across multiple XML files. Ideally splitting it into logical pieces across application layers can make maintenance easier by keeping each of the Spring configuration files focused on a single layer of the application. To ensure that all the configuration files are loaded, you will need to configure a context loader in your web.xml file. A context loader loads context configuration files in addition to the one that DispatcherServlet loads. The most commonly used context loader is a servlet listener called ContextLoaderListener that is configured in web.xml as shown in the first listing above.

With ContextLoaderListener configured, we need to tell it the location of the Spring configuration file(s) to load. If not specified, context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml. We will specify one or more Spring configuration file(s) by setting the contextConfigLocation parameter in the servlet context as seen in the second listing above. The context loader will use contextConfigLocation to load the two context configuration files – one each for the service and data layer.

4.2 Annotation-based controller configuration

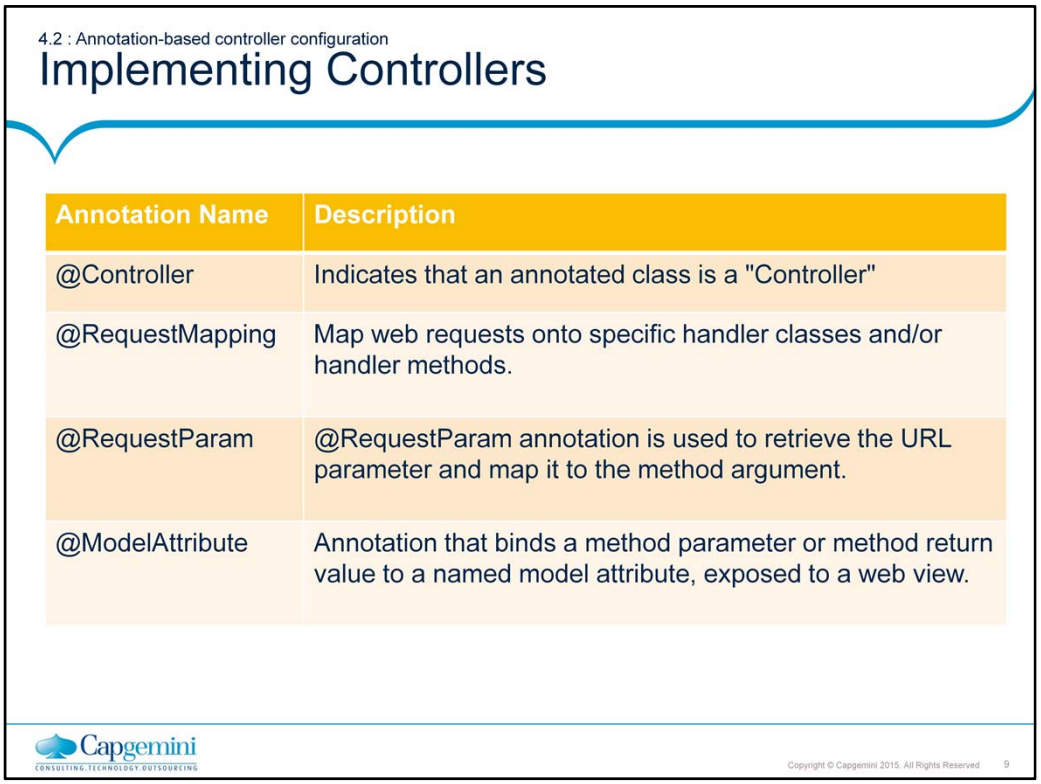
Controller

- The most typical handler used in Spring MVC for handling web requests is a controller.
- From Spring2.5 onwards @Controller annotation is used to design a controller class

```
@Controller
 public class HelloController {
    @RequestMapping("/helloWorld")
    public String showMessage() {
                return "hello";
}}
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 8



@Controller: Indicates that an annotated class is a "Controller"

Example:
@Controller
Public class LoginController{}

@RequestMapping: @RequestMapping can be applied to the controller class as well as methods. It maps web requests onto specific handler classes and/or handler methods.

Example:
@Controller
@RequestMapping(“loginController”)
Public class LoginController{
@RequestMapping(“loadForm”)
public String loadData(){}
}

URL need to be used for making the request for the above mentioned controller is

http://localhost:8080/projectname/loginController/loadForm.obj

Example:

```
@Controller
Public class LoginController{
                @RequestMapping(“loadForm”)
                public String loadData(){}
}
```

URL need to be used for making the request for the above mentioned controller is

http://localhost:8080/projectname/loadForm.obj

@RequestParam annotation is used to retrieve the URL parameter and map it to the method argument.

Example:

```
@Controller
Public class LoginController{
                @RequestMapping(“checkLogin”)
                public String isValidUser(@RequestParam(“username”
                String uname)
                {}
}
```

URL need to be used for making the request for the above mentioned controller is

http://localhost:8080/projectname/checkLogin.obj?username=igate

@ModelAttribute Annotation that binds a method parameter or method return value to a named model attribute, exposed to a web view.

Example:

```
@Controller
Public class LoginController{
                @RequestMapping(“checkLogin”)
                public String isValidUser(@ModelAttribute(“user”)
                UserBean userBean)
                {}
}
```

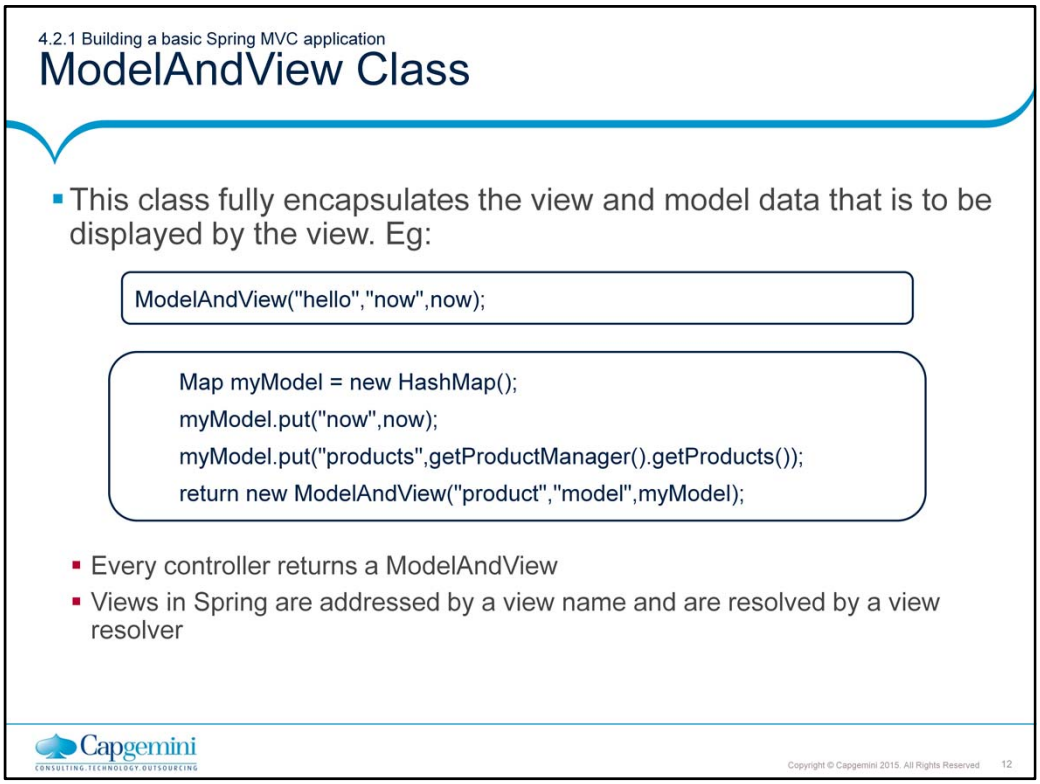
4.2.1 Building a basic Spring MVC application

Handler Mapping

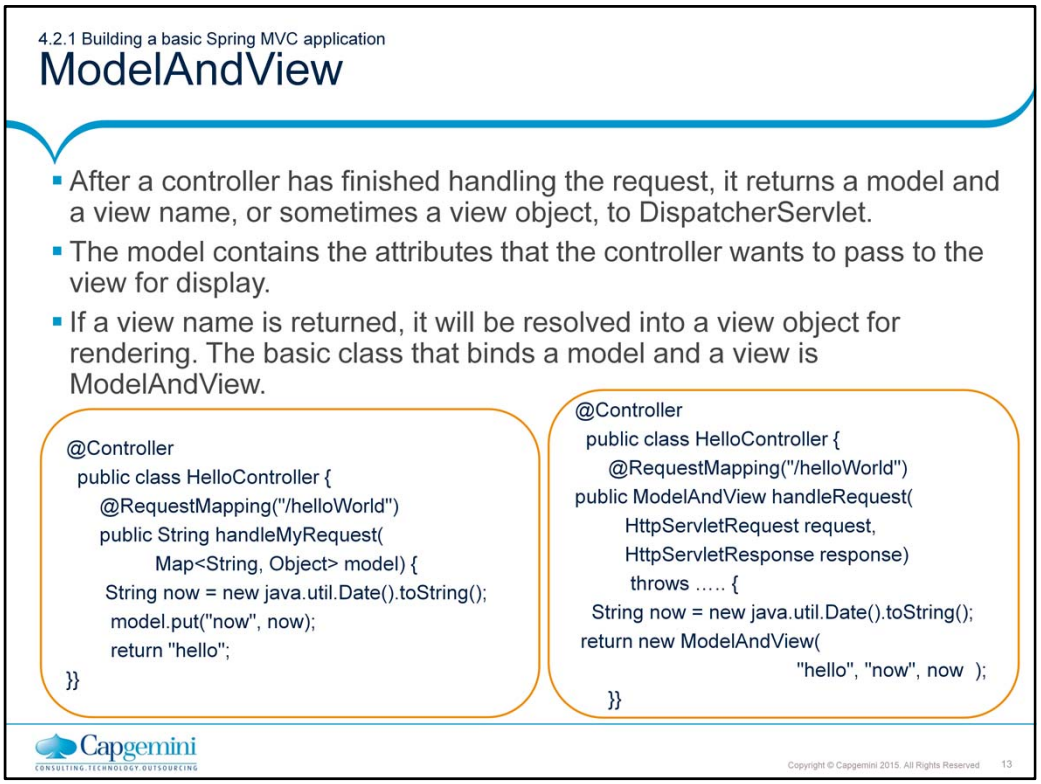
- A handler mapping is a bean configured in the web application context that implements the Handler Mapping interface. It is responsible for returning an appropriate handler for a request.
- Handler mappings usually map a request to a handler according to the request's URL. It is an arbitrary Java object that can handle web requests.
- The most typical handler used in Spring MVC for handling web requests is a controller.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 11



The ModelAndView class allows you to specify a response to the client, resulting from actions performed in controller. This object holds both – the view the client will be presented with and the model used to render the view. Every controller must return a ModelAndView. See the two examples above.
The first parameter is the logical name of a view component that will be used to display the output from this controller. The next two parameters represent the model object that will be passed to the view and its value.
In the second example, the view component is product and the model is an object that is to be returned and its value is MyModel which is a HashMap object containing multiple values.
In the end, the dispatcher servlet needs a concrete view instance to render the view.



Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @Controller, @RequestMapping, @RequestParam, @ModelAttribute etc. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces.
See the listing above for a simple example.
@Controller annotation indicates that this class is a controller class. This is a specialization of the @Component annotation. Thus <context: component-scan> will pick up and register @Controller-annotated classes as beans, just as if they were annotated with @Component.
@RequestMapping annotation serves two purposes. First, it identifies handleRequest() as a request-handling method. Second, it specifies that this method should handle requests whose path is /helloWorld. See the second listing above.
Notice that handleMyRequest(), a user-defined method, replaces the handleRequest() method. @RequestMapping annotation identifies it as a request-handling method.
The handleMyRequest() takes a Map as a parameter, which represents the model—the data that's passed between the controller and a view. But the signature of a request-handling method can have anything as an argument.
Notice that handleMyRequest() returns a String value which is the logical name of the view that should render the results. ViewResolver uses this to resolve actual view.
Note : @RequestMapping annotation can be used at class level too to map URLs onto an entire class. The @RequestMapping at class level defines the root URL path that the controller will handle. Method-level @RequestMappings narrow the scope of class-level @RequestMapping.

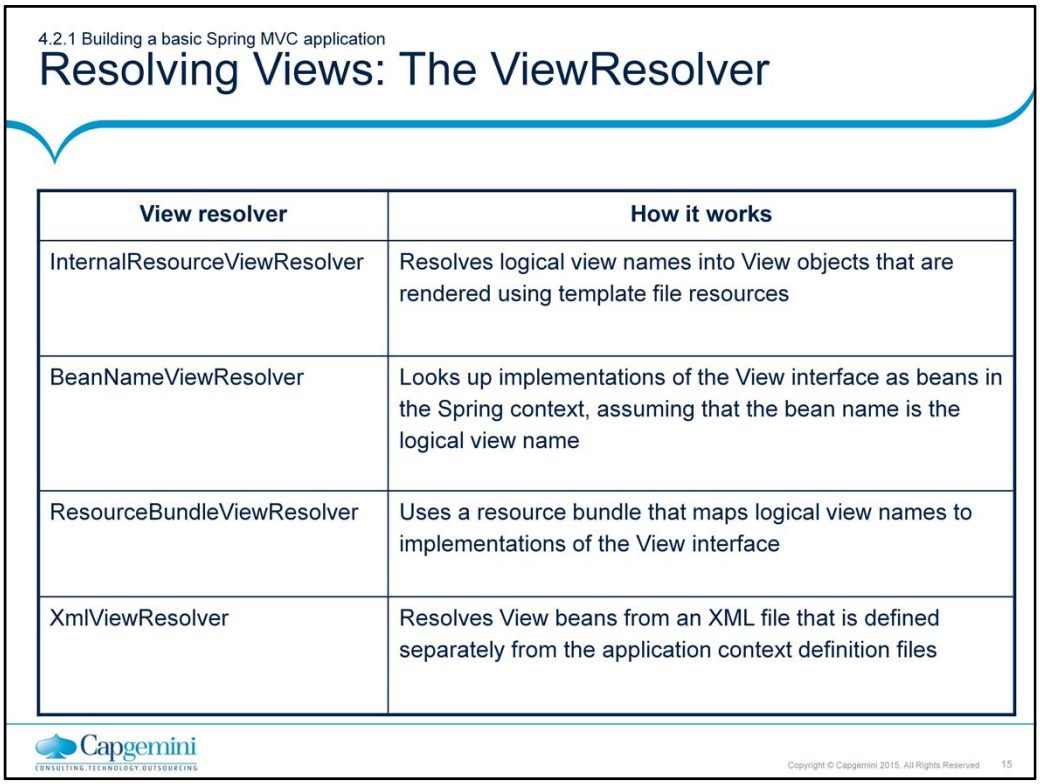
4.2.1 Building a basic Spring MVC application

ViewResolver

- When DispatcherServlet receives a model and a view name, it will resolve the logical view name into a view object for rendering.
- DispatcherServlet resolves views from one or more view resolvers.
- A view resolver is a bean configured in the web application context that implements the ViewResolver interface.
- Its responsibility is to return a view object for a logical view name.

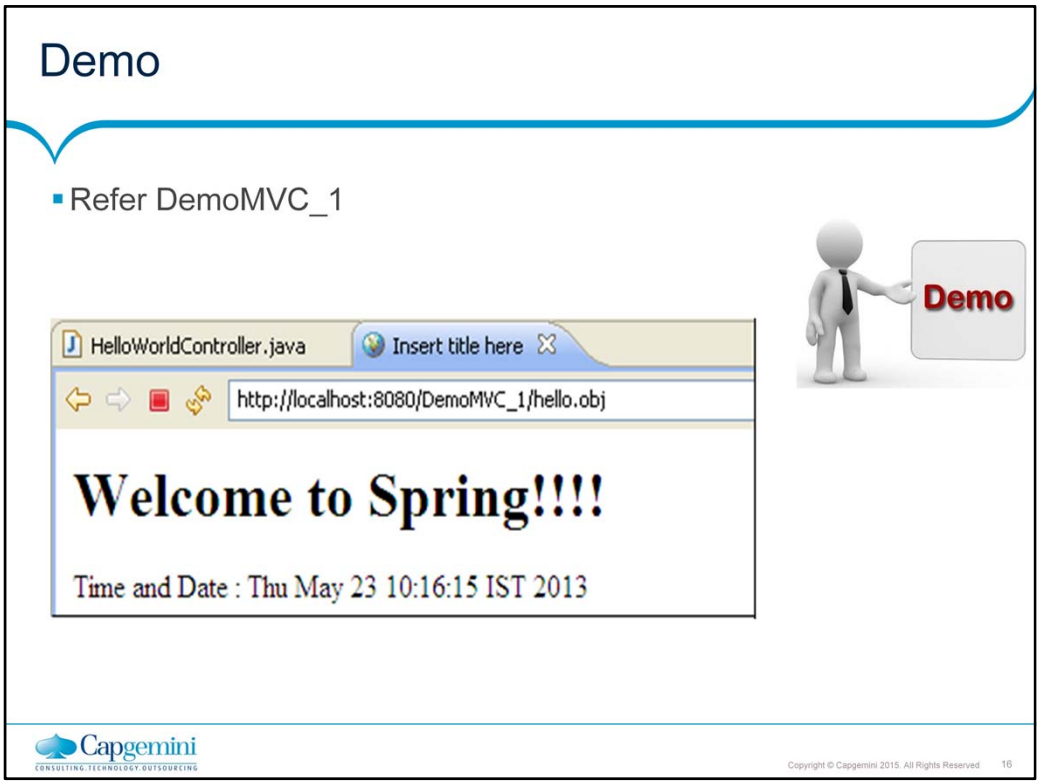
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 14

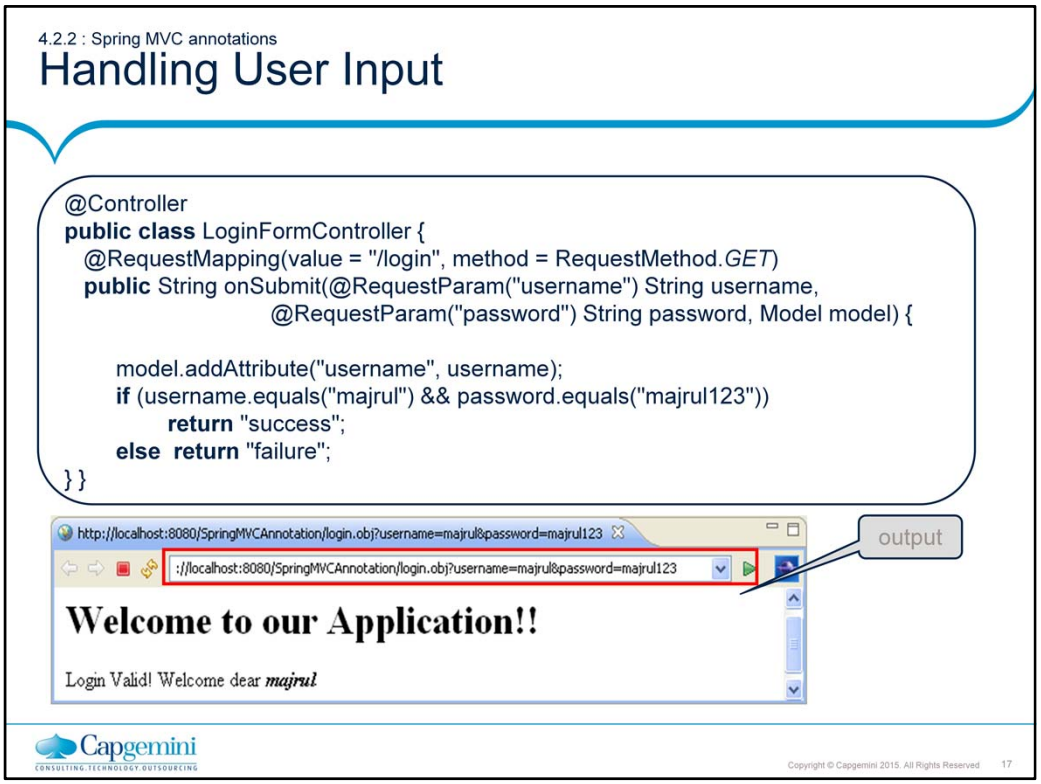


So far, we have seen how model objects are passed to the view through the ModelAndView object. In Spring MVC, a view is a bean that renders results to the user. The view most likely is a JSP. But you could also use other view technologies like Velocity and FreeMarker templates or even views that produce PDF and MS-Excel documents.
View resolvers resolve the view name given by the ModelAndView object to a View bean. Spring provides a number of useful view resolvers, some of which are shown in the table above. See Spring docs for more.
InternalResourceViewResolver resolves a logical view name by affixing a prefix and a suffix to the view name returned by the ModelAndView object. It then loads a View object with the path of the resultant JSP. By default, the view object is an InternalResourceView, which simply dispatches the request to the JSP to perform the actual rendering. But, if the JSP uses JSTL tags, then you may replace InternalResourceView with JstlView as seen in the code demos earlier.

```
<bean id="viewResolver" class=
"org.springframework.web.servlet.view.InternalResourceViewResolver ">
  <property name="viewClass">
     <value>org.springframework.web.servlet.view.JstlView </value>
   </property>
  <property name="prefix"><value>/</value></property>
  <property name="suffix"><value>.jsp</value></property>
</bean>
```

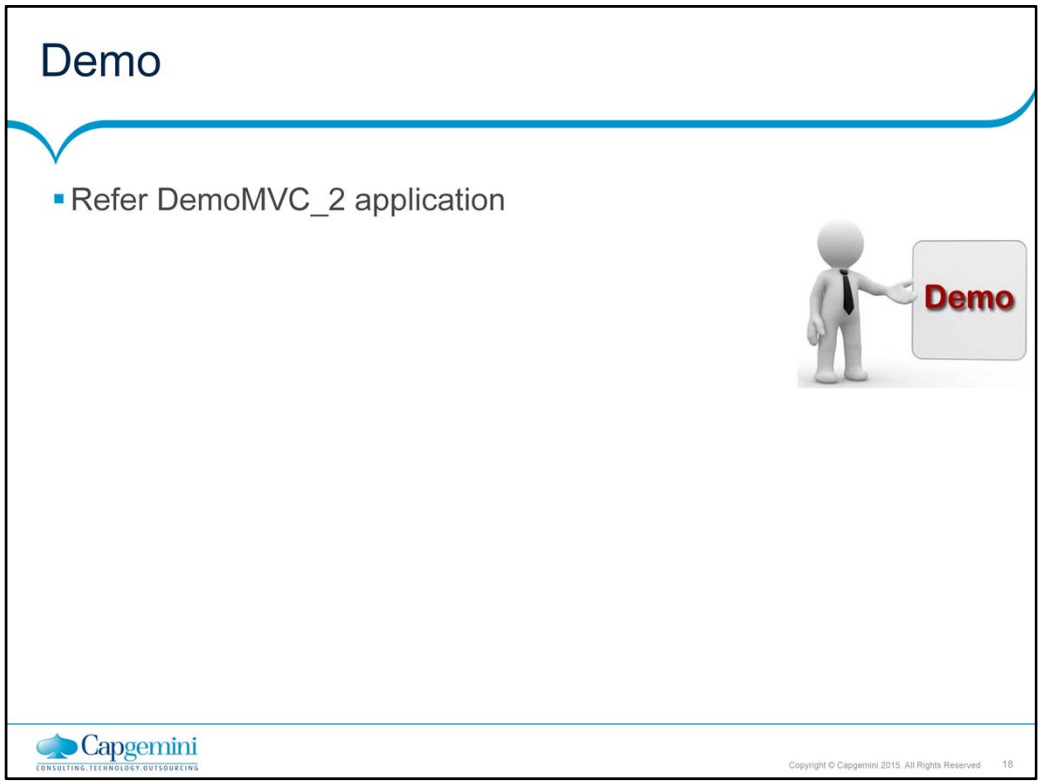


Please refer to the DemoMVC_1 web project.

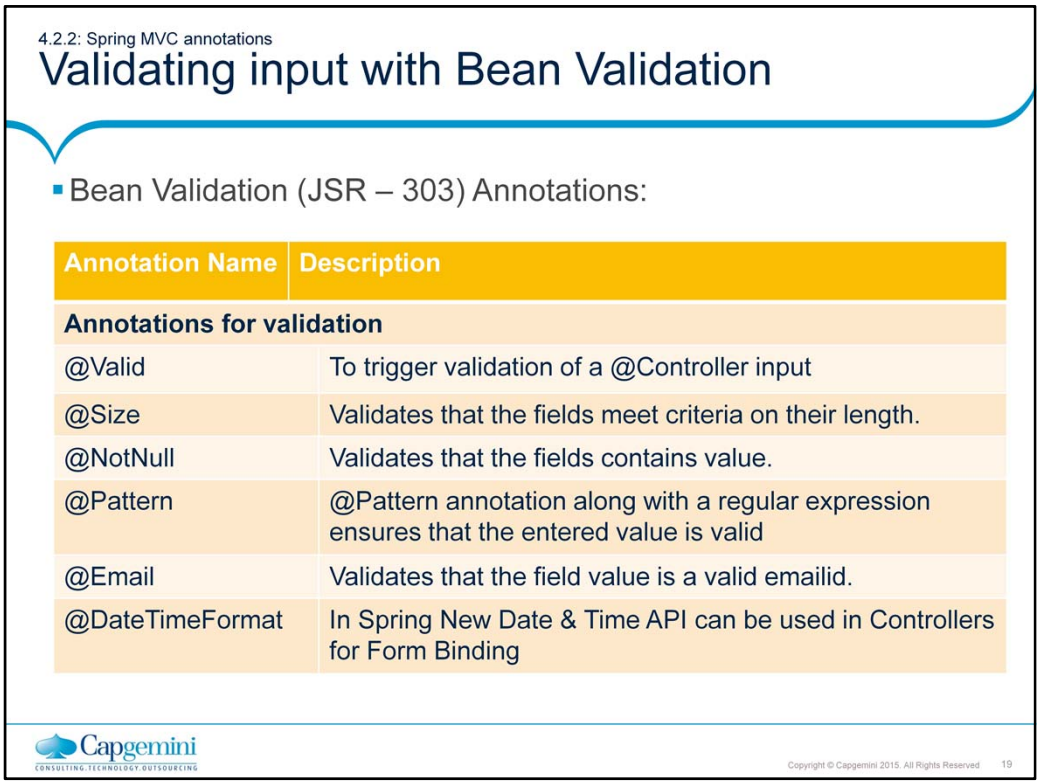


We have seen simple Controllers so far. But Controllers may need to perform business logic using information passed in as URL parameters or as submitted form data.
See the code listing above for an example. Here, LoginFormController is mapped to /login at the method level. This means that onSubmit() handles requests for /login. “method” attribute is set to GET indicating that this method will only handle HTTP GET requests for /login. The onSubmit() method takes two String parameters (username and password) and a Model object as parameters. The username parameter is annotated with @RequestParam(“username") to indicate that it should be given the value of the username query parameter in the request. Same goes for password too. onSubmit() will use these parameters to authenticate user.
Now, see the model parameter. In earlier examples we passed in a Map<String, Object> to represent the model. Here we’re using a new Model parameter. Model provides a few convenient methods for populating the model, such as addAttribute(). The addAttribute() method is similar to Map’s put() method, except that it finds out key part of the map by itself.
Notice how the values set in addAttribute() can be accessed in the jsp page:

```
<!-- success.jsp →
<body>
<h1>Welcome to our Application!!</h1>
  Login Valid! Welcome dear <i><b>${username} </b></i>
</body>
```



Please refer to the DemoMVC_2web project.
Refer to HelloWorldController.java.
Invoke the application as :
http://localhost:8080/DemoMVC_2/hi/hello.obj?name=CapGemini



Before an object can be processed further, it is essential to ensure that all the data in the object is valid and complete. Faulty form input must be rejected. For example, username must not contain spaces, password must be minimum 6 characters long, email must be correct etc.

The @Valid annotation (part of the JavaBean validation specification) tells Spring that the User object should be validated as it's bound to the form input. If anything goes wrong while validating the User object, the validation error will be carried to the processForm() method via the BindingResult that's passed in on the second parameter. If the BindingResult's hasErrors() method returns true, then that means that validation failed.

How do we declare validation rules?

JSR-303 defines some annotations that can be placed on properties to specify validation rules. The code above shows the properties of the User class that are annotated with validation annotations.

@Size annotation validates that the fields meet criteria on their length.

@Pattern annotation along with a regular expression ensures that the value given to the email property fits the format of an email address and that the username is only made up of alphanumeric characters with no spaces.

Notice how we've set the message attribute with the message to be displayed in the form when validation fails. With these annotations, when a user submits a registration form to AddUserFormController's processForm() method, the values in the User object's fields will be validated. If any of those rules are violated, then the handler method will send the user back to the form.

@Valid : To trigger validation of a @Controller input

```
@Controller
Public class LoginController{
                @RequestMapping(“checkLogin”)
                public String isValidUser(@ModelAttribute(“user”)
                @Valid UserBean userBean)
                {}
}
```

@Size: Validates that the fields meet criteria on their length.
@Pattern annotation along with a regular expression ensures that the entered value is valid

Example:

```
public class UserBean
{
                @Size(min=7,max=10,message="Phone Number Should Accept Only 10 digits")
                @Pattern(regexp = "^[0-9]+$", message = "Phone Number should contain only 10 digits")
                private String phoneNumber;
                //Getter and Setter methods
}
```

@NotNull / @NotEmpty: Validates that the fields contains value.

Example:

```
                @NotEmpty(message="Please Enter Address")
                private String address;
```

@Email: Validates that the field value is a valid emailid.

Example:

```
                @Email(message="Please enter valid Email ID")
                private String email;
```

@DateTimeFormat: In Spring New Date & Time API can be used in Controllers for Form Binding
Example:

```
                @DateTimeFormat(pattern="M/d/yy h:mm")
                private LocalDateTime birthDate;
```

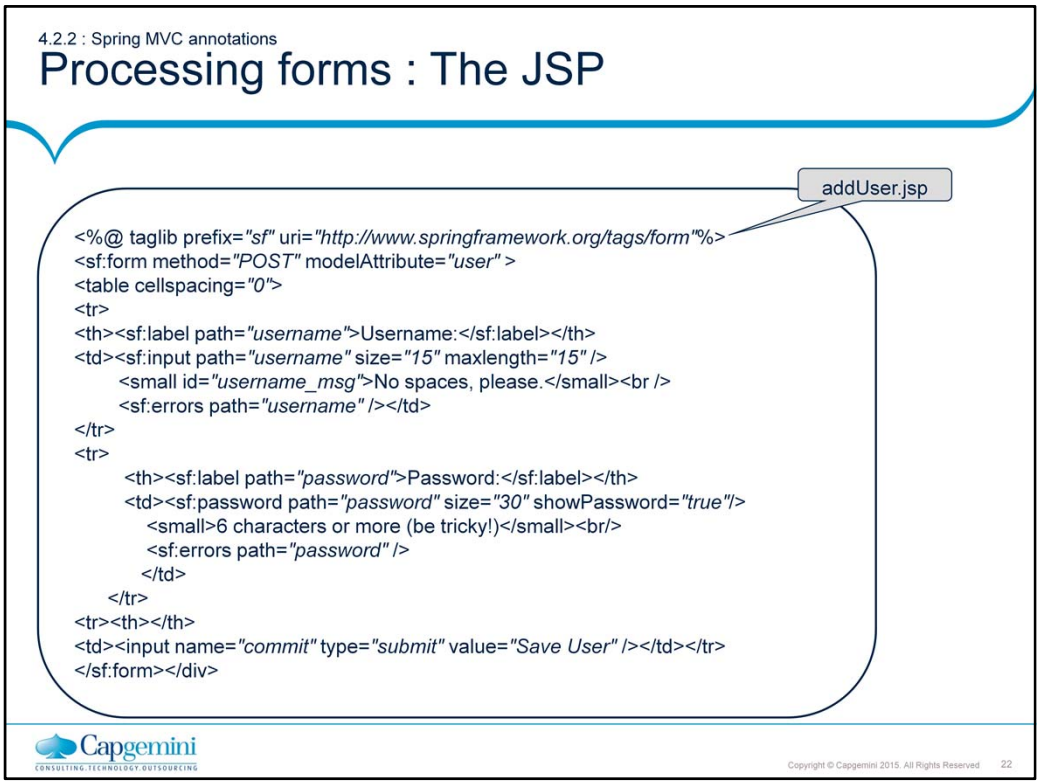
4.2.2: Spring MVC annotations

Validating input : declaring validation rules

```
public class User {
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20
characters long.")
    @Pattern(regexp = "^[a-zA-Z0-9]+$", message = "Username must be alphanumeric
with no spaces")
    private String username;

    @Size(min = 6, max = 20, message = "The password must be at least 6 characters
long.")
     private String password;

     @Pattern(regexp = "[A-Za-z0-9]+@[A-Za-z0-9.-]+[.][A-Za-z]{2,4}", message =
"Invalid email address.")
     private String email;

//getter and setter methods for all these properties
}
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 21

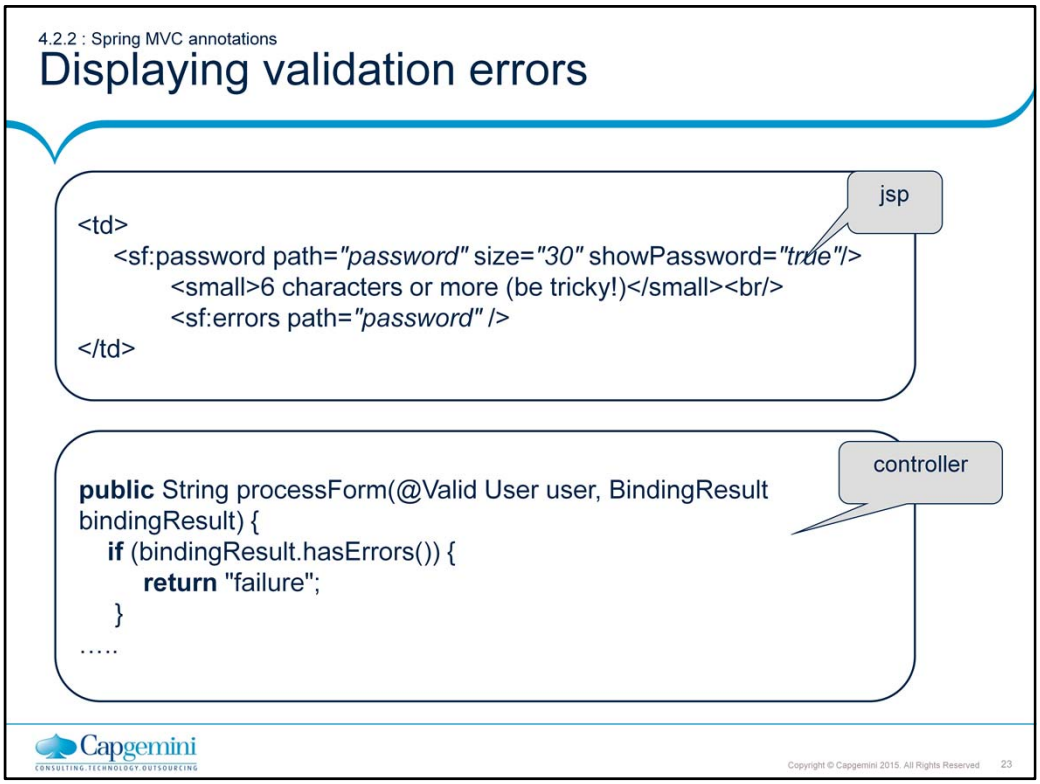


The addUser.jsp page uses Spring's form binding library. The <sf:form> tag binds the User object (identified by the modelAttribute attribute - see above) that showForm() placed into the model to the various fields in the form. The <sf:input> and <sf:password> tags have a path attribute that references the property of the User object that the form is bound to. When the form is submitted, whatever values these fields contain will be placed into a User object and submitted to the server for processing.

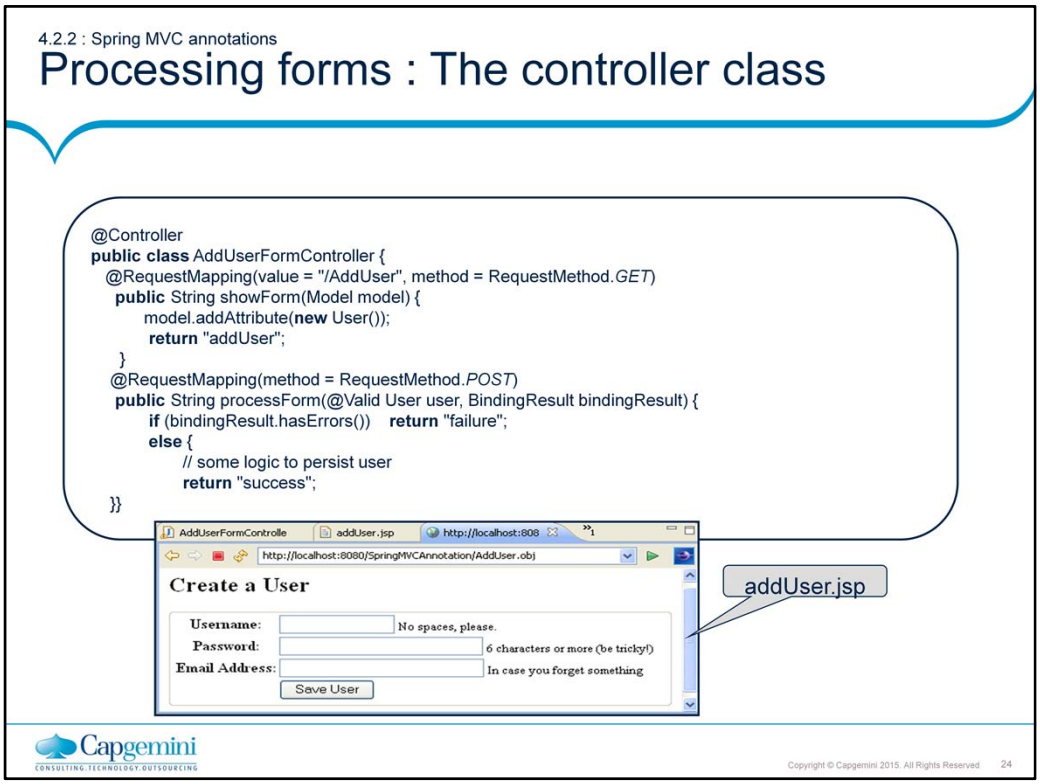
Note that the <sf:form> specifies that it'll be submitted as an HTTP POST request. We thus now need another handler method that accepts POST requests. See the processForm() method in the code listing in the previous page which will process form submissions.

When the addUser.jsp form is submitted, the fields in the request will be bound to the User object (passed as an argument to processForm(). From there, some logic can be employed to persist user object into database.

Notice that User parameter is annotated with @Valid. This indicates that the User should pass validation before being persisted. We shall cover validation next.



We now need to tell jsp to display the validation messages.
We have passed the BindingResult parameter to processForm(). This knows whether the form had any validation errors via its hasErrors() method. But the actual error messages are also in there, associated with the fields that failed validation. To display those errors to the users, use Spring's form binding JSP tag library to display the errors. ie, the <sf:errors> tag can render field validation errors. See code listing above to see how this is done.
The <sf:errors> tag's path attribute specifies the form field for which errors should be displayed. For example, the above listing displays errors (if any) for the field whose name is password. If there are multiple errors for a single field, they'll all be displayed, separated by an HTML
 tag.



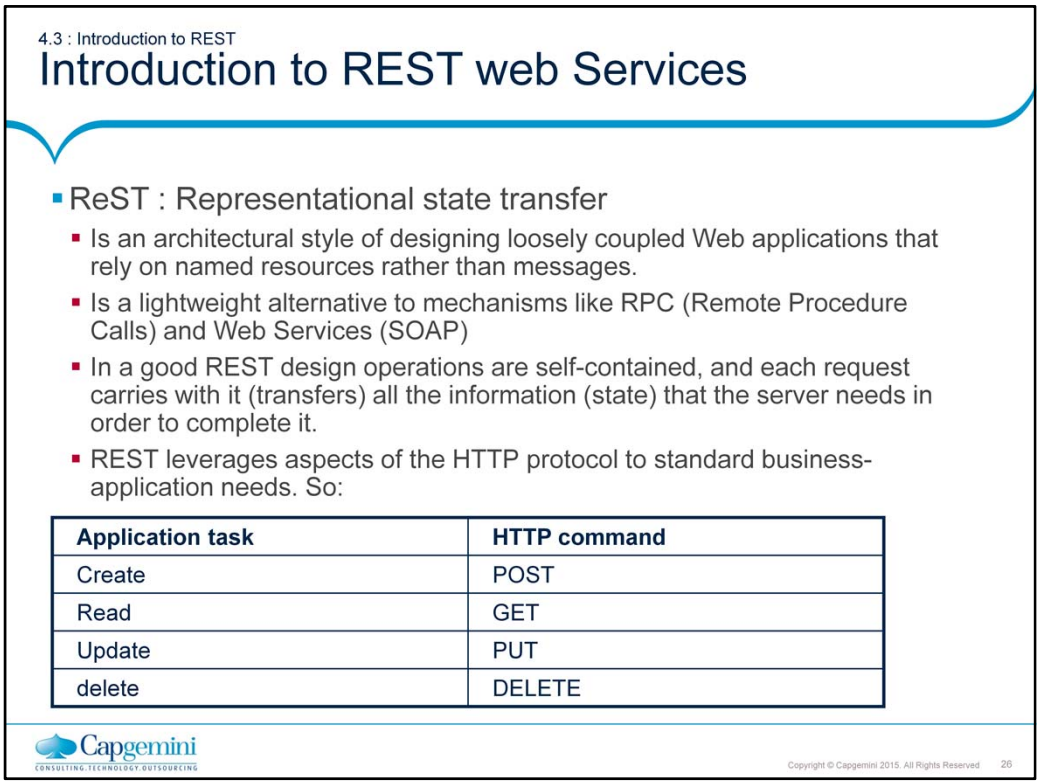
Working with forms in a web application involves two operations: displaying the form and processing the form submission.
Our example allows to register new User. For this we have defined two handler methods to AddUserFormController to handle each of the operations. We first need to display the form in the browser before it can be submitted. The first method (showForm()) displays the registration form. See the figure above.
Once the form is displayed, it'll need a User object to bind to the form fields. The showForm() method creates a User object and places it in the model.

Demo

- Refer the following Demos:
  - DemoMVC_3
  - DemoMVC_4
  - DemoMVC_5
  - DemoMVC_6
  - DemoMVC_7
  - DemoMVC_Complete

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 25



REST is a style of designing loosely coupled Web applications that rely on named resources — in the form of Uniform Resource Identifiers (URIs, for instance — rather than messages. REST leverages aspects of the HTTP protocol such as GET and POST requests.

These requests map quite nicely to standard business-application needs such as create read, update, and delete (CRUD)

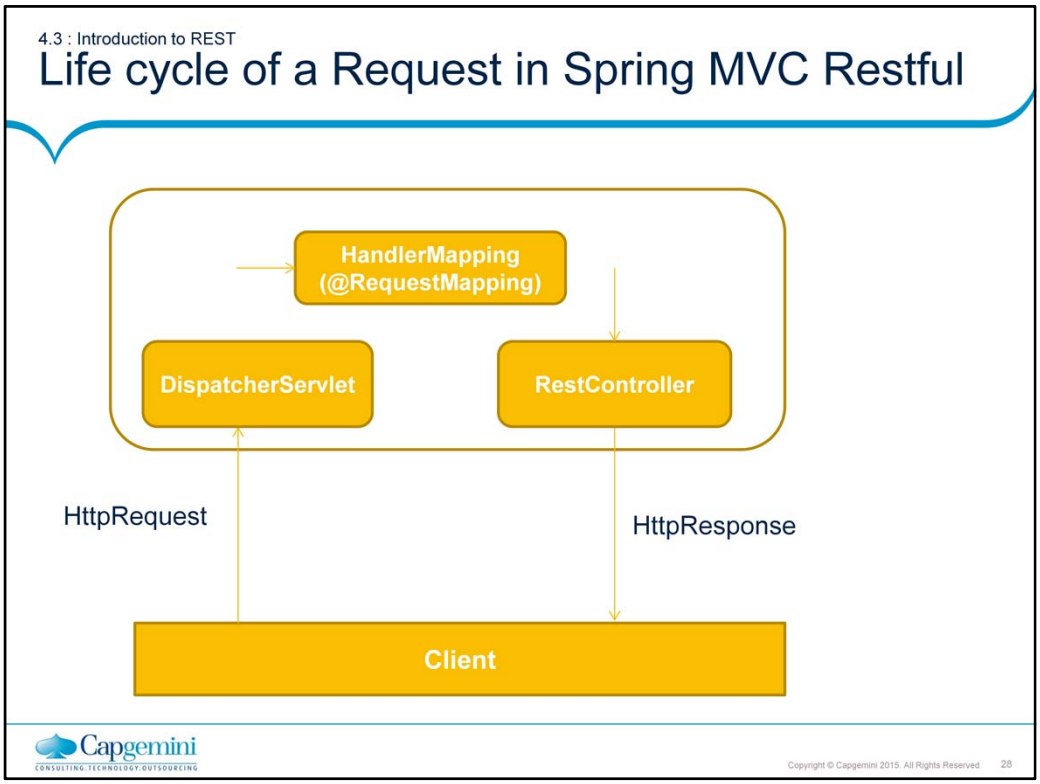
4.3 : Introduction to REST

Introduction to REST web Services

- Principles of REST web Services
  - Use HTTP methods explicitly
  - Be stateless
  - Expose directory structure-like URIs
  - Transfer XML, JavaScript Object Notation (JSON), or both

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 27



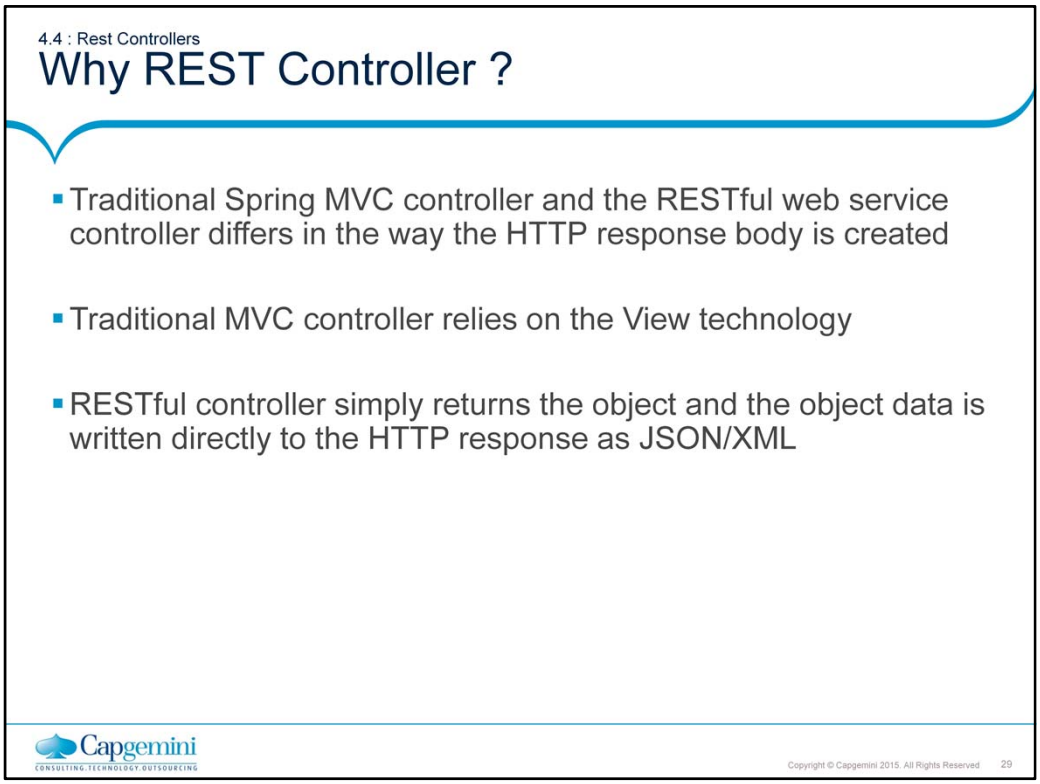
REST(Representational State Transfer) is an architectural style with which Web Services can be designed that serves resources based on the request from client. A Web Service is a unit of managed code, that can be invoked using HTTP requests. You develop the core functionality of your application, deploy it in a server and expose to the network. Once it is exposed, it can be accessed using URI's through HTTP requests from a variety of client applications. Instead of repeating the same functionality in multiple client (web, desktop and mobile) applications, you write it once and access it in all the applications.

In the above diagram, from the time that a request is received by Spring until the time that a response is returned to the client, many pieces of Spring Restful webservices are involved.

The process starts when a client (typically a web browser) sends a request. It is first received by a DispatcherServlet. Like most Java-based MVC frameworks, Spring MVC uses a front-controller servlet (here DispatcherServlet) to intercept requests. This in turn delegates responsibility for a request to other components of an application for actual processing.

The Spring MVC uses a Controller component for handling the request. But a typical application may have several controllers. To determine which controller should handle the request, DispatcherServlet starts by querying one or more HandlerMappings. A HandlerMapping typically maps URL patterns to RestControllers.

Once the DispatcherServlet has a appropriate RestController selected, it dispatches the request to that Controller which performs the business logic (a well-designed RestController object delegates responsibility of business logic to one or more service objects). Upon completion of business logic, HTTPResponse is generated and sent back to the client.



The key difference between a traditional Spring MVC controller and the RESTful web service controller is the way the HTTP response body is created. While the traditional MVC controller relies on the View technology, the RESTful controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.

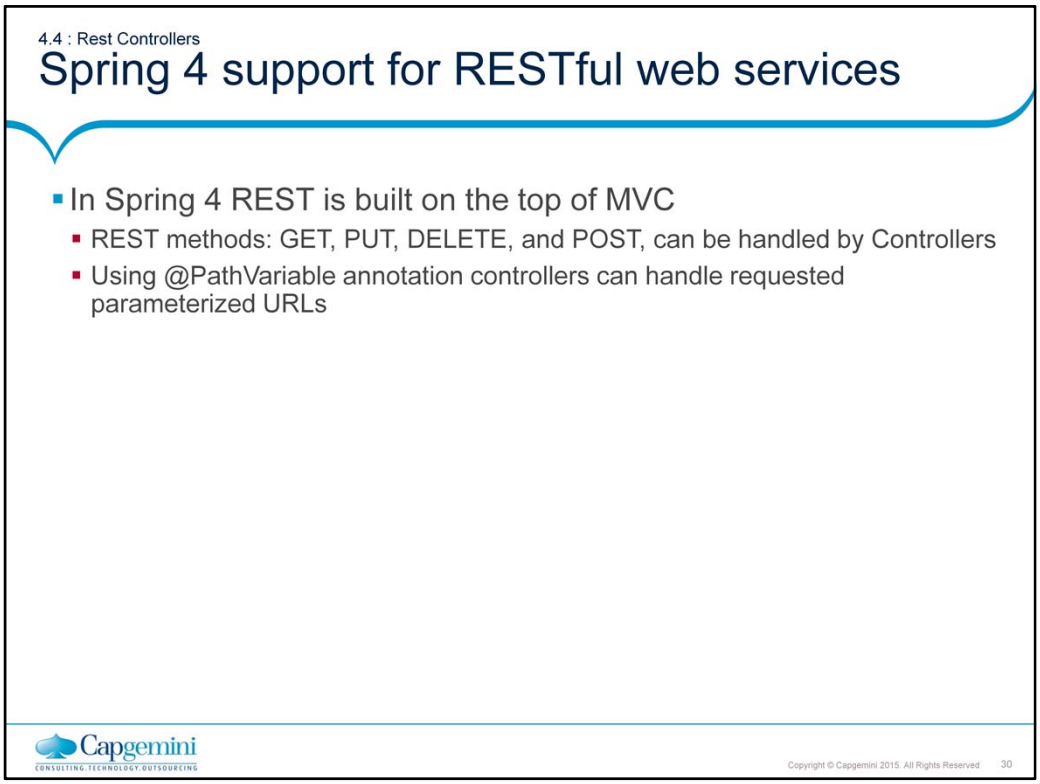
In Spring 3 @ResponseBody was used as an annotation over the Rest Controller methods on the return type indicating the HTTP response as JSON/XML, along with this we had to specify the MIME type as “application/json” or “application/xml”.

For example: Spring 3 Rest Controller method:

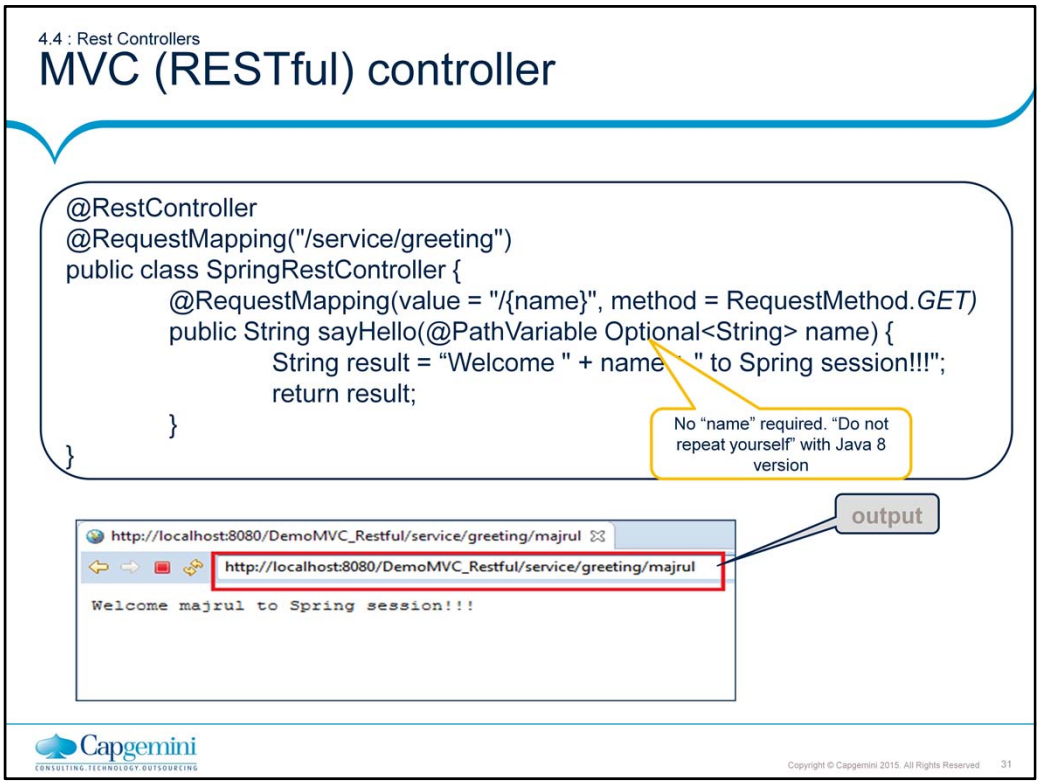
```
(
@Controller
@RequestMapping("employees")
public class EmployeeController {

@RequestMapping(value = "/{name}", method = RequestMethod.GET, produces =
"application/json")
   public @ResponseBody Employee getEmployeeInJSON(@PathVariable String name) {
                        employee.setName(name);
                        employee.setEmail("employee1@info.com");
                        return employee;
    }
)
```

In Spring 4 @RestController is a combination of @Controller + @ResponseBody annotations. Thus in Spring 4 we do not need to use @ResposneBody; we directly can use @RestController and return view and or the object to the HTTP response.

In our example discussed in next subsequent slides we have kept view (jsp) itself as the ResponseBody.



Spring supports different Request methods like GET, PUT, DELETE, POST corresponding to respective HTTP methods.



Let us see what a RESTful controller looks like.

Restful controller
@RequestMapping annotation which says that this controller will handle requests for /service/greeting. It implies and supports the fact that this controller is focused on displaying greeting message.

URL (uniform resource locator) is a means of locating a resource. But, since, no two resources can share the same URL, URL could also be a means of uniquely identifying a resource ie URI. Many URLs don't locate or identify anything—they make demands.

Rather than identify something, they demand some action be taken. For example consider below URL:

http://localhost:8080/DemoMVC_Restful/service/greeting?name=capgemini, this kind of URL will be handled by the SpringRestController's sayHello() method. Here, the URL is not locating or identifying a resource. The base portion of the URL is verb-oriented ie "sayXXX()".

RESTful URLs on the other hand are resource-oriented. So, the URL in the example above should look like the following :
http://localhost:8080/DemoMVC_Restful/service/greeting/capgemini

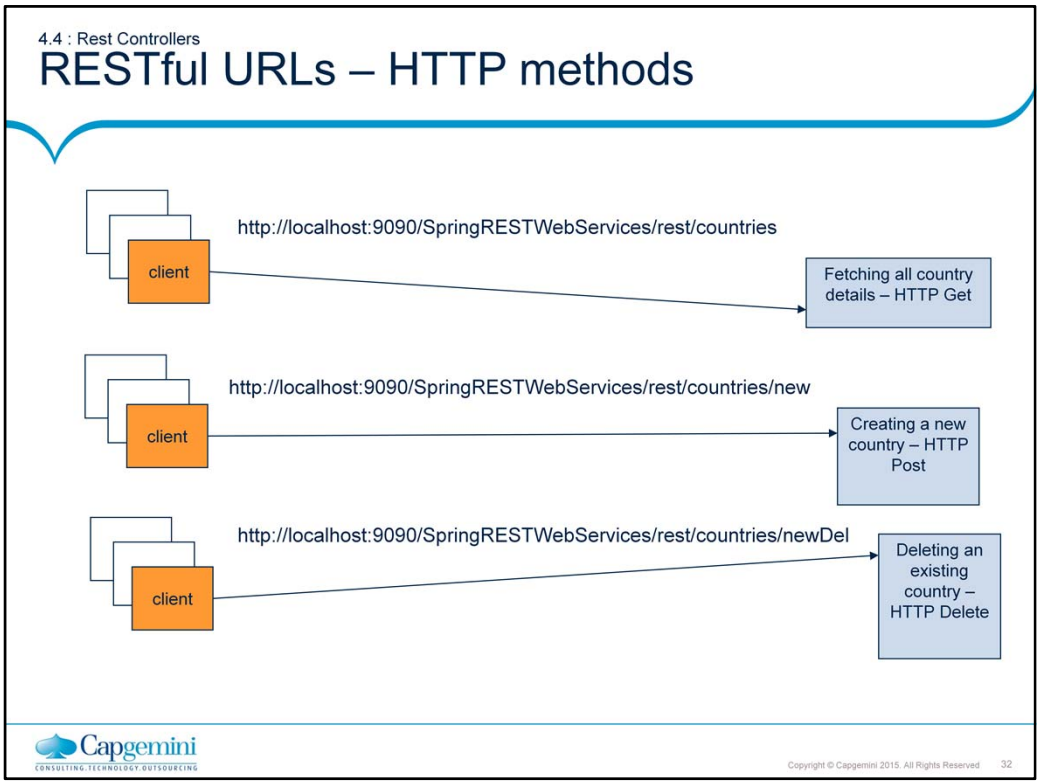
This URL now correctly locates and identifies a resource. Notice, the base portion of the URL is resource-oriented. Also notice that it has no query parameters, instead it has a parameterized path. The RESTful URL's input is part of the URL's path.

But how will you code the controller to take input from the URL's path?

At the method level, {name} in the @RequestMapping annotation, is a placeholder through which variable data will be pass into the method.

The sayHello() method is designed to handle GET requests for URLs that take the form /{name}. The @PathVariable annotation corresponds to this id.
So, if a GET request comes in for
http://localhost:8080/DemoMVC_Restful/service/greeting/capgemini, the sayHello() method will be called with capgemini passed in for the name parameter.

The method then uses that value to generate welcome message and place it into the model.



The slide demonstrates different RESTful URLs with respect to different HTTP methods.

Demo: SpringRESTWebServices can be used as a reference.

Note: At times if the HTTP Get is used over a couple of RestController methods it has to be combined with URL patterns to create unique identifications.

In the above slide first URL pattern demonstrates : HTTP Get method to fetch all country details.

Similarly if details for a particular country need to be fetched then the country id can be appended in URL and extracted via the @PathVariable

http://localhost:9090/SpringRESTWebServices/rest/countries/3 -> With this URL country details are fetched for country Id = 3

2. The second URL pattern demonstrates : HTTP Post method to create a new country

3. The third URL pattern demonstrates : HTTP Delete method to delete an existing country

Note: As HTML supports only Get and Post methods for the method attribute in the form tag; we also need to map the HTTP PUT(update) and HTTP DELETE (delete) methods to update and delete the resources respectively.

For this Spring provides us with a Filter-mapping which is to be given in web.xml file:

```
<filter>
   <filter-name>httpMethodFilter</filter-name>
   <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
 </filter>
 <filter-mapping>
   <filter-name>httpMethodFilter</filter-name>
   <servlet-name>dispatcher</servlet-name>
 </filter-mapping>
```

By using this Spring in-built filter the different methods of HTTP specification will be mapped to their actual HTTP implementations.
Here the filter will be intercepted for all the requests coming to DispatcherServlet.

Also in the JSP pages for updating and deleting a country need to pass a HTML hidden parameter : For example

<input type="hidden" name="_method" value="delete"/> to pass the “real” HTTP method to Spring Rest Controller.

4.4 : Rest Controllers

Demo: DemoMVC_Restful

- DemoMVC_Restful
- SpringRESTWebServices

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 33

Summary

- We have so far seen:
  - How to use Spring MVC architecture to build flexible and powerful web applications.
  - Components like handler mappings, ViewResolvers and controllers
  - MVC Annotations like @Controller, @RestController, @RequestMapping , @RequestParam, @PathVariable

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 34

Review Questions

- Question 1: If multiple handler mappings have been declared in an application, select the property that indicates which handler mapping has precedence?
  - Option 1: Order
  - Option 2: Sequence
  - Option 3: Index
  - Option 4: An application cant have multiple handler mappings
- Question 2: To figure out which controller should handle the request, DispatcherServlet queries _____
  - Option 1: HandlerMappings
  - Option 2: ModelAndView
  - Option 3: ViewResolver
  - Option 4: HomeController

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Basic Spring 4.0

Spring JPA Integration

Lesson Objectives

- After completing this lesson, participants will be able to understand:
  - Spring support for JPA
  - Implementing Spring JPA integration
  - Spring Data JPA

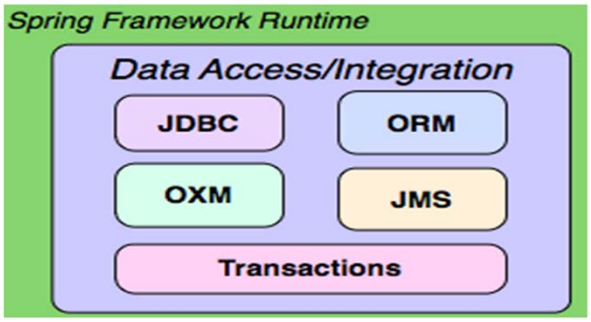
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2

5.1 : Spring Support for JPA

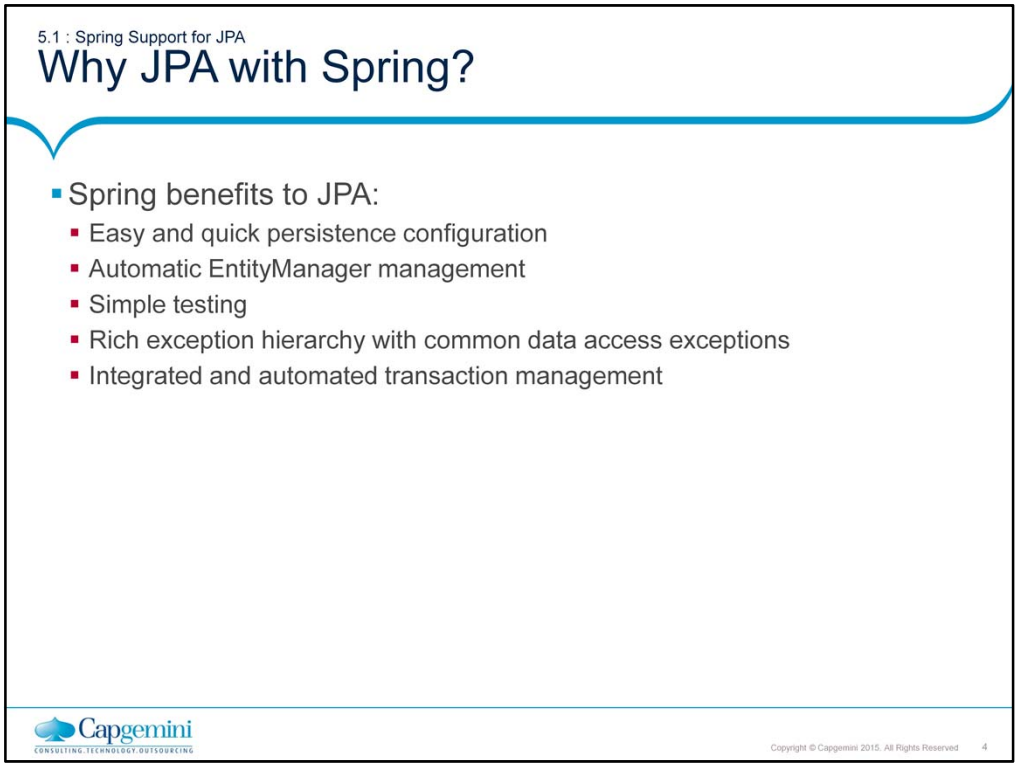
Spring JPA Integration : Overview

- The Spring Framework supports integration with Hibernate, Java Persistence API (JPA) for
  - resource management,
  - data access object (DAO) implementations, and
  - transaction strategies



Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 3



Why Spring JPA Integration?

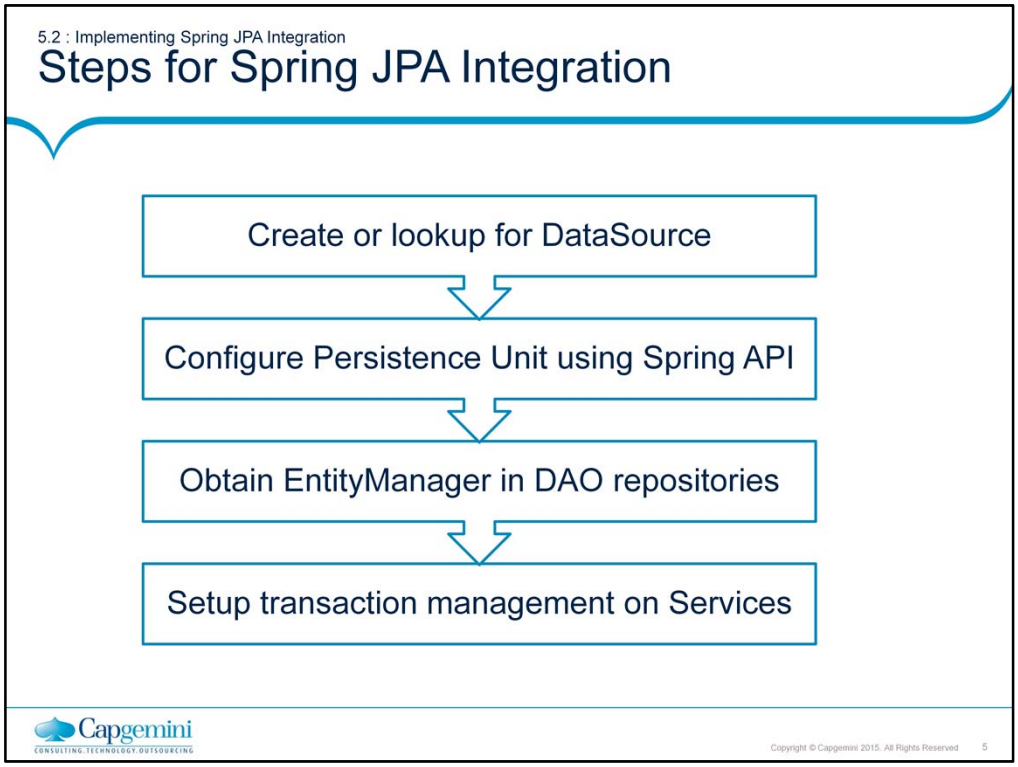
The client application that accesses the database using JPA has to depend on the JPA APIs like PersistenceContext, EntityManagerFactory, EntityManager and Transaction. These objects will continue to get scattered across the code throughout the Application.

Moreover, the Application code has to manually maintain and manage these objects. In the case of Spring, the business objects can be highly configurable with the help of IOC Container. Which means that now it is possible to use the JPA objects as Spring Beans and they can enjoy all the facilities that Spring provides.

The EntityManager instance is managed automatically, and can be injected into data access object (DAO) beans; so there is no need to manage it manually in application code.

Another advantage of integrating JPA in Spring is that Spring manages starting, committing, and rolling back transactions automatically on your behalf. You will never use it directly and you'll only configure an implementation.

The JPA defines a modest exception hierarchy starting at PersistenceException, but even it is missing some key features like an exception to indicate that a unique key violation occurred. Spring Framework solve this problem by defining a thorough hierarchy of persistence exceptions.



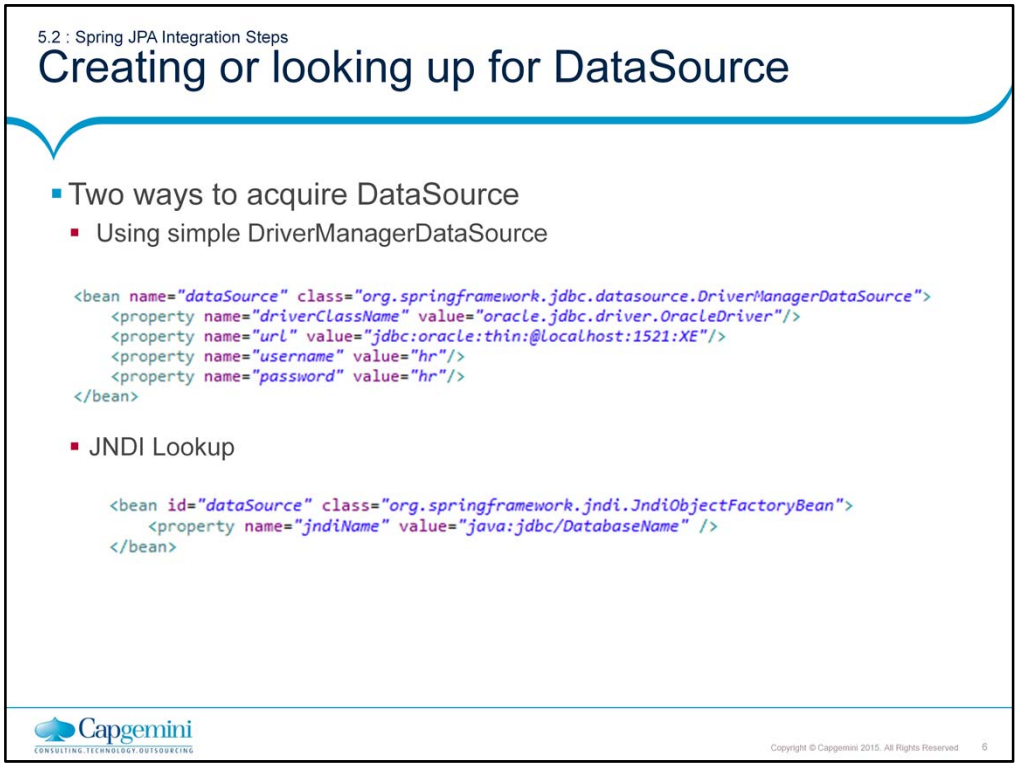
Integration Steps:

Integration work starts by defining or obtaining data source, which contains information about database url, username and password etc.

Once we obtain datasource, we need configure JPA related beans using spring configuration file. It is used in place of persistence.xml.
Though, we are using Spring configuration, optionally we may keep persistence.xml just to hold the persistence unit reference. This file may be required in future for advance configuration like caching etc.

After configuration, we can work on entities and DAO repositories to define our database operations.

Finally, we need to instruct Spring to handle all JPA transactions, this is done by annotating and marking DAO operations with various transaction demarcation policies.

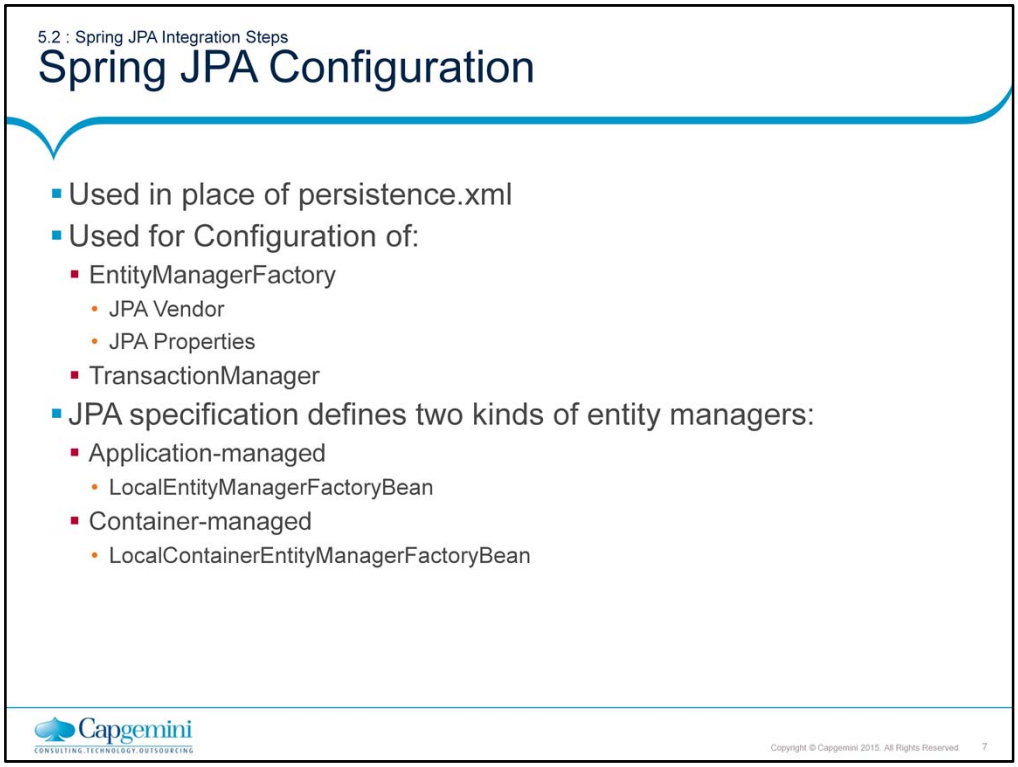


The first step towards integration is making DataSource available to application. There are different ways to do this. For example, if you need to simply test something quickly, use Spring's DriverManagerDataSource to create a DataSource on demand.

However, this creates a simple DataSource that returns single use Connections. Because it does not provide connection pooling, it really should never be used in a production environment. For production environment, we can define DataSource on server and look that DataSource up application.

Slide demonstrates both ways to acquire DataSource for our application.

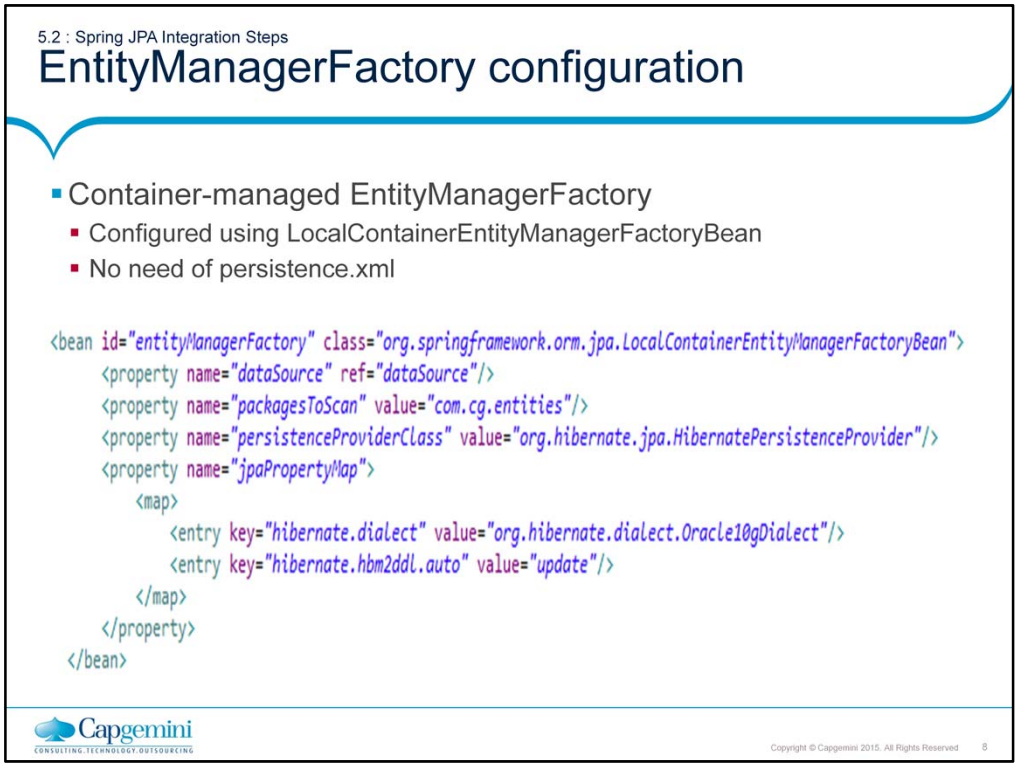
Note: To keep configuration simple, we are using DriverManagerDataSource throughout integration.



The EntityManagerFactory is what we use to start up JPA inside our application. Without Spring, we use persistence.xml file to configure this object. While working with Spring, we don't need configuration via persistence.xml, instead we can define all JPA related objects as spring beans.
Spring allows two ways to define this object, either application managed or container managed (used throughout this integration).

In addition to EntityManagerFactory, Spring provides TransactionManager to handle all JPA related transactions, which is also configured inside spring configuration file.

Let us see in subsequent slides, how to bootstrap this configuration file.



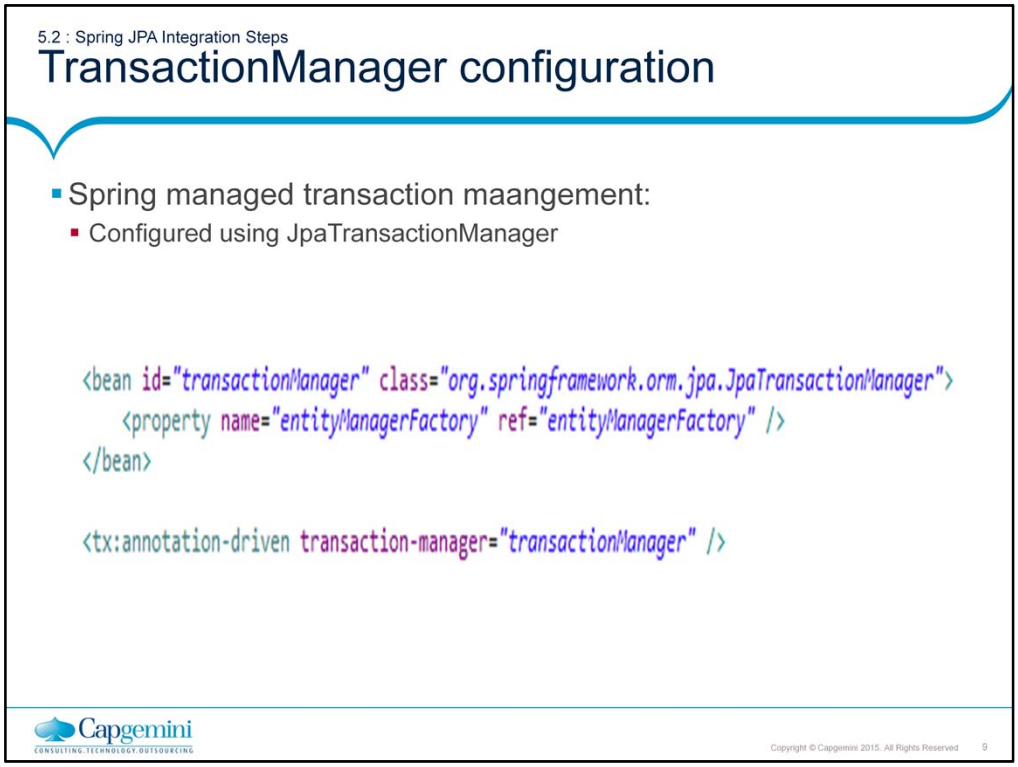
Spring offers three different options to configure EntityManagerFactory in a project:
LocalEntityManagerFactoryBean
EntityManagerFactory lookup over JNDI
LocalContainerEntityManagerFactoryBean

LocalEntityManagerFactoryBean is the most basic and limited one. It is mainly used for testing purposes and standalone environments. It reads JPA configuration from /META-INF/persistence .xml, doesn't allow you to use a Spring-managed DataSource instance, and doesn't support distributed transaction management.

Use EntityManagerFactory lookup over JNDI if the run time is Java EE 5 Server.

LocalContainerEntityManagerFactoryBean is the most powerful and flexible JPA configuration approach Spring offers. It gives full control overEntityManagerFactory configuration, and it's suitable for environments where fine-grained control is required. It enables you to work with a Spring-managedDataSource, lets you selectively load entity classes in your project's classpath, and so on. It works both in application servers and standalone environments.

Above slide demonstrates on how to configure LocalContainerEntityManagerFactoryBean in spring XML configuration file.

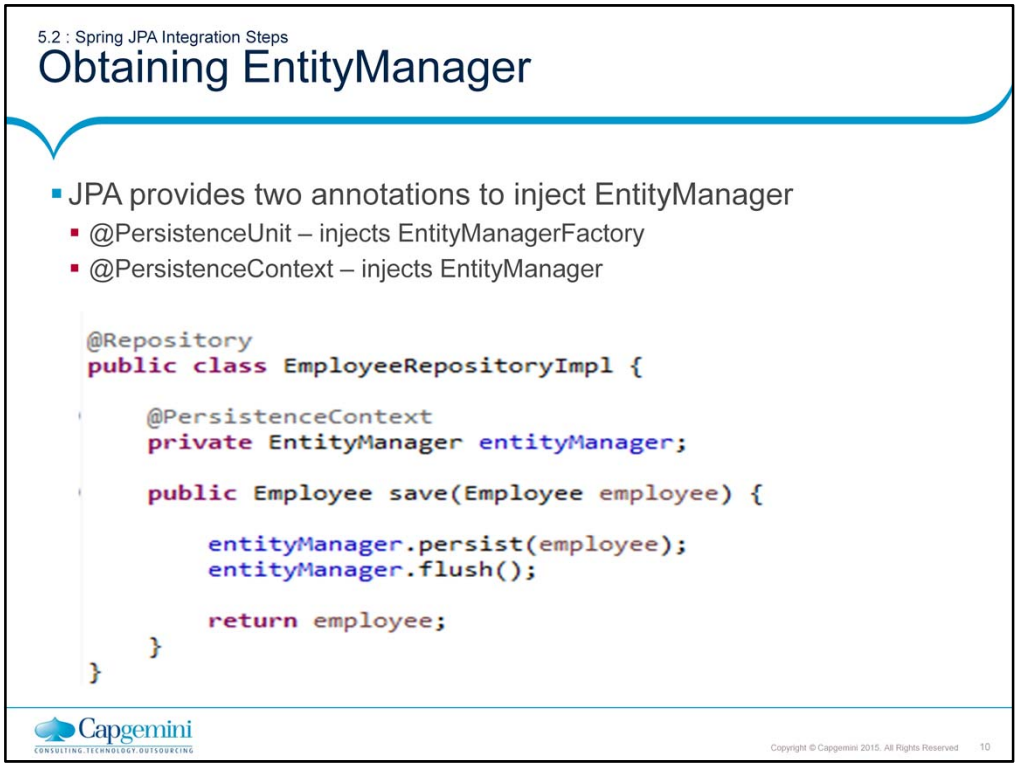


Configuration of TransactionManager:

Spring provides a class org.springframework.orm.jpa.JpaTransactionManager to work with JPA specific transactions. Slide demonstrate how to configure this class in spring configuration file. It is required to inject entitymanagerfactory created in earlier step in this class using setter injection.

Basically all of JPA interactions will be wrapped in transaction by this TransactionManager and we are instructing spring to create a bean of this class for application transaction management.

Once this transaction manager is made available, instead of opening and closing transactions manually, we can instruct Spring to handle transaction using annotations. To do so, we must use <tx:annotation-driven/> injecting the transaction manager instance.



You can obtain EntityManagerFactory or EntityManager from a container via dependency injection, and you can automatically participate in the current transaction.

Note: You need to configure the PersistenceAnnotationBeanPostProcessor of Spring in order for annotations given in the slide to be processed. You can either define it as bean, or enable a context namespace and add <context:annotation-config/> or <context:component-scan/> elements into your XML bean configuration file.

5.2 : Spring JPA Integration

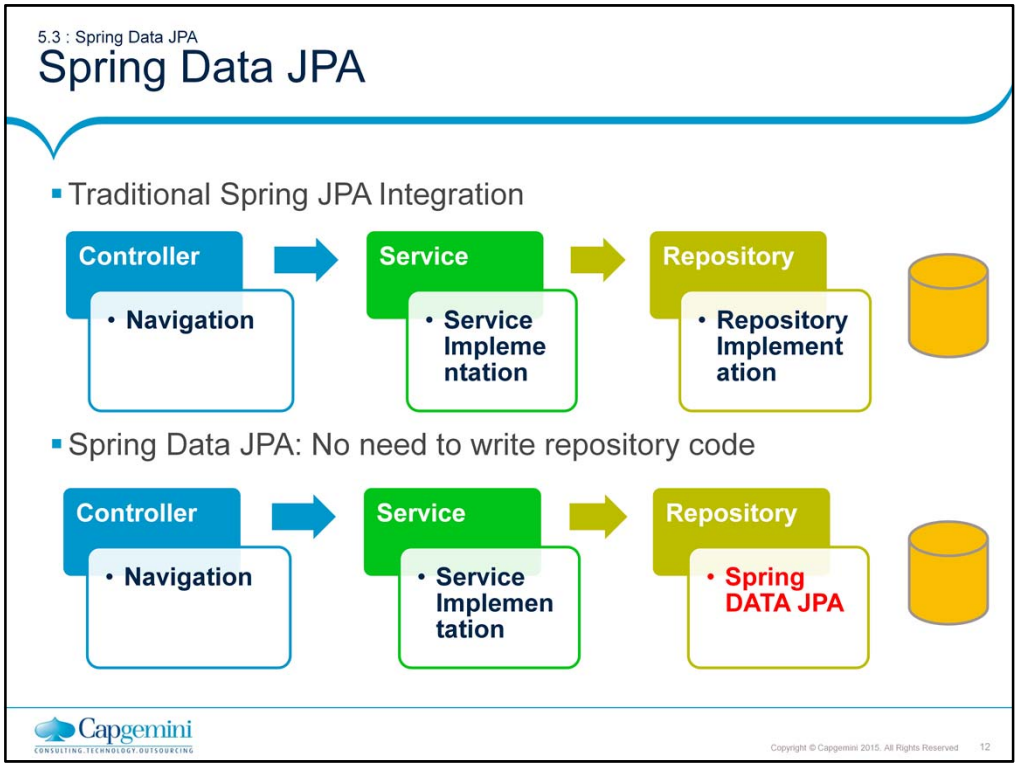
Demo

- JPASpringMVC

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 11

Add the notes here.



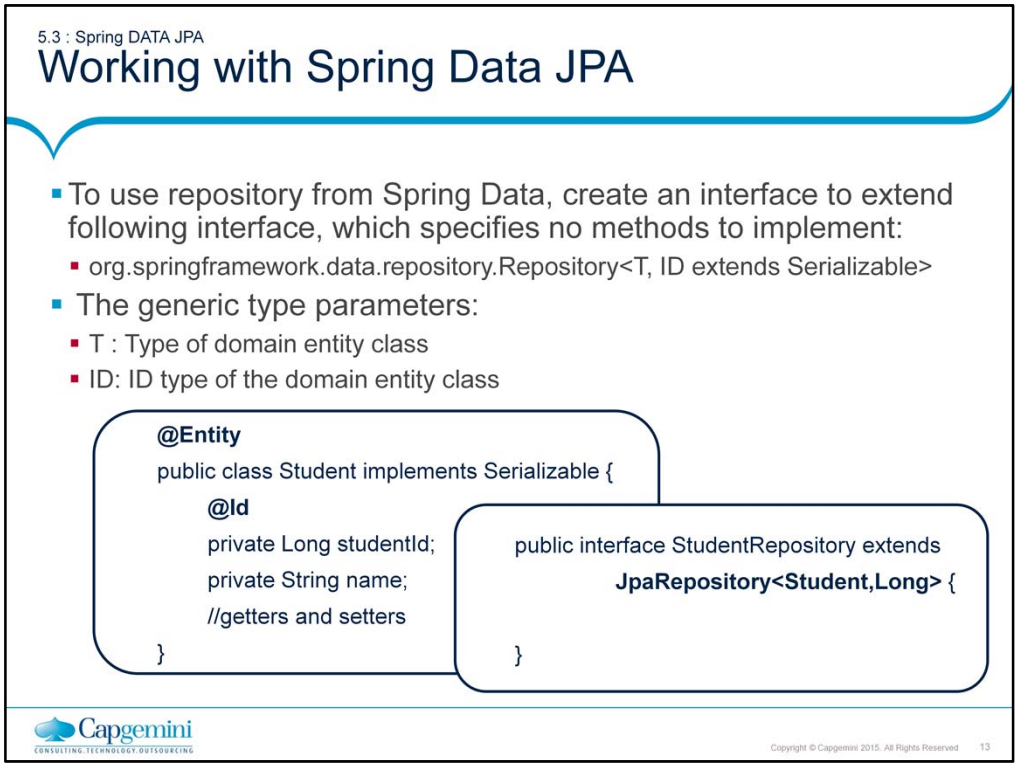
What is Spring Data JPA?

Spring Data supports a variety of data access methodologies, including JPA, JdbcTemplate, NoSQL, and more. Its primary subproject, Spring Data Commons, provides a core toolset that all other subprojects use to create repositories. The Spring Data JPA subproject provides support for repositories implemented against the Java Persistence API.

Spring Data JPA, a separate Spring project but dependent on Spring Framework, can write your repositories on behalf of developers.

Spring Data JPA is another layer of abstraction for the support of persistence layer in spring context. The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores. We just have to write repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

Note: To configure Spring Data using XML required a new Spring Data JPA namespace, http://www.springframework.org/schema/data/jpa and to enable auto scanning of application repositories, one need to specify <jpa:repositories> element in spring configuration.



As a first step we define a domain class-specific repository interface. The interface must extend JpaRepository and be typed to the domain class and an ID type. JpaRepository is a child inteface of following:

CrudRepository<T,ID>,
PagingAndSortingRepository<T,ID>,
QueryByExampleExecutor<T>,
Repository<T,ID>

The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed. One can add additional query methods to this repository interface. Below listed are few methods of CrudRepository:

count() returns a long representing the total number of unfiltered entities extending T.
delete(T) and delete(ID) delete the single, specified entity and deleteAll() deletes every entity of that type.
exists(ID) returns a boolean indicating whether the entity of this type with the given key exists.
findAll() returns all entities of type T, whereas findOne(ID) retrieves a single entity of type T given its key.
save(T) saves the given entity (insert or update).

Note: One can define additional query methods using @Query. Please see the demo for more details.

5.3 : Spring Data JPA

Demo

- JPASpringDataMVC

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 14

Add the notes here.

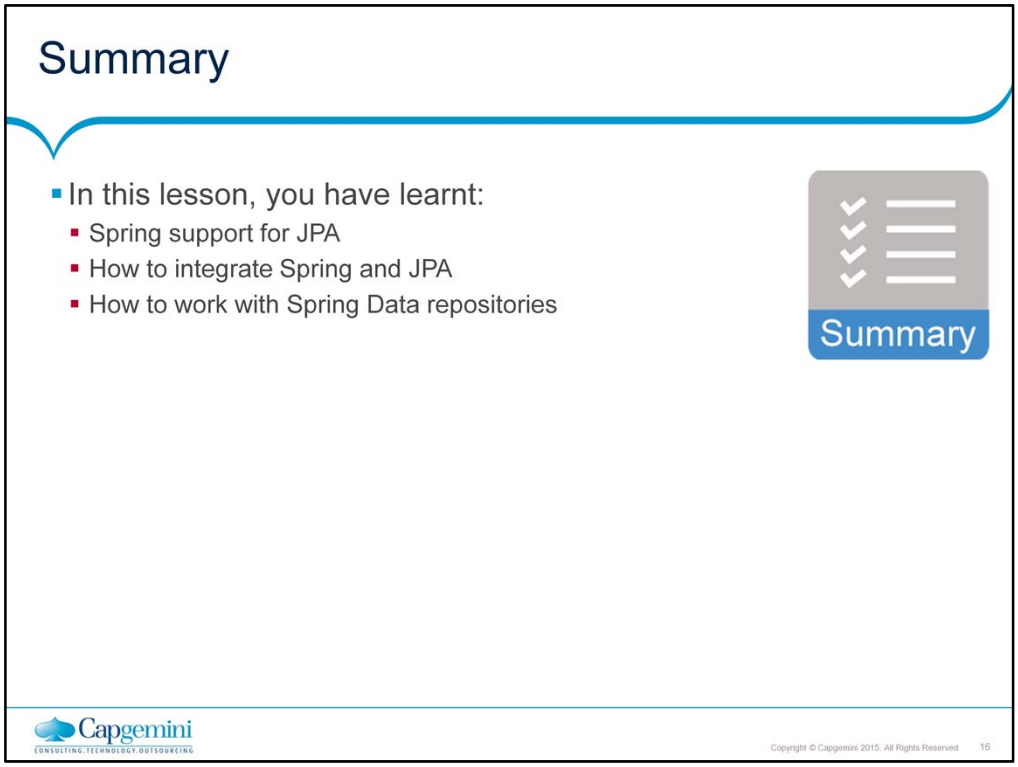
Lab

- Lab 2

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 15

Add the notes here.



Add the notes here.

Review Question

- Question 1 Which one of the following is valid type of entity manager in JPA?
  - Option 1: Container managed
  - Option 2: JVM Managed
  - Option 3: Server Managed
- Question 2 LocalEntityManagerFactoryBean doesn't allow you to use a Spring managed DataSource instance.
  - True/False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 17

Add the notes here.

Basic Spring 4.0

Lesson 6: Aspect Oriented Programming (AOP)

Lesson Objectives

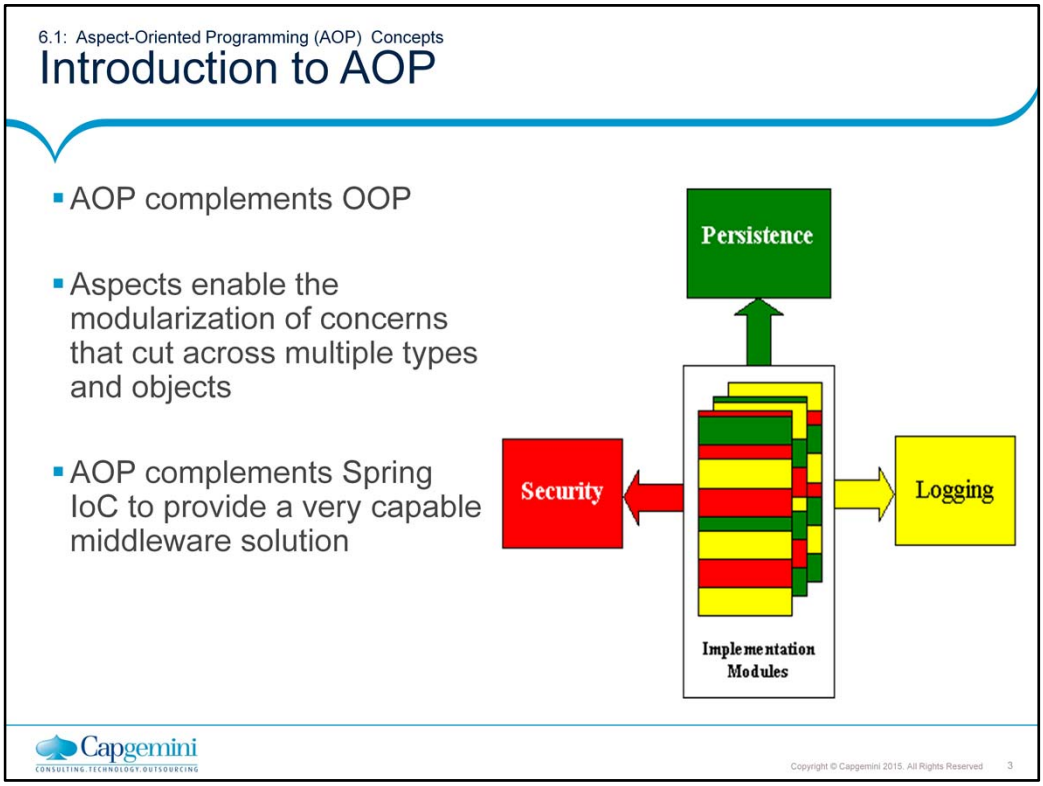
- Introduction to Spring AOP
  - Learn AOP basics and terminologies
  - Understand key AOP terminologies
  - Understand the different ways that Spring supports AOP

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2

Topics to be covered:

- Introduction to AOP
- AOP concepts
- AOP support in Spring using @AspectJ support
- AOP support in Spring using Schema-based AOP support

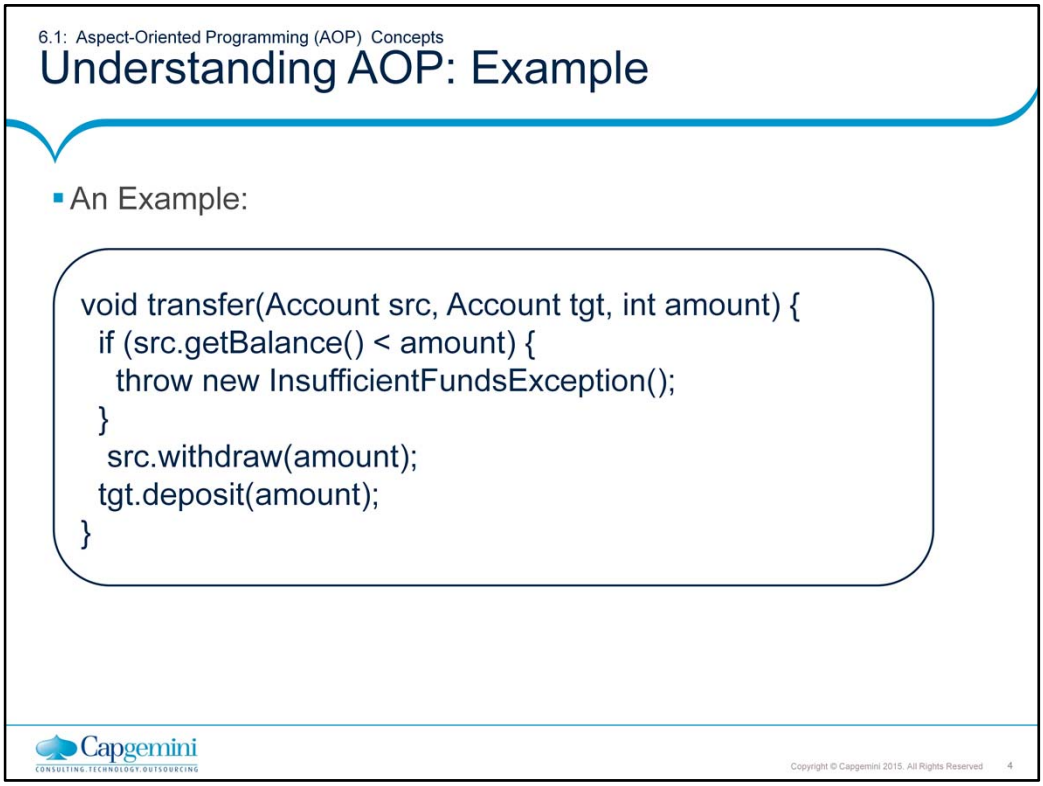


Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. OOP decomposes applications into a hierarchy of objects; AOP decomposes prg's into aspects and concerns. This allows modularization of concerns such as transaction management that would otherwise cut across multiple objects. Such concerns are often called cross-cutting concerns. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

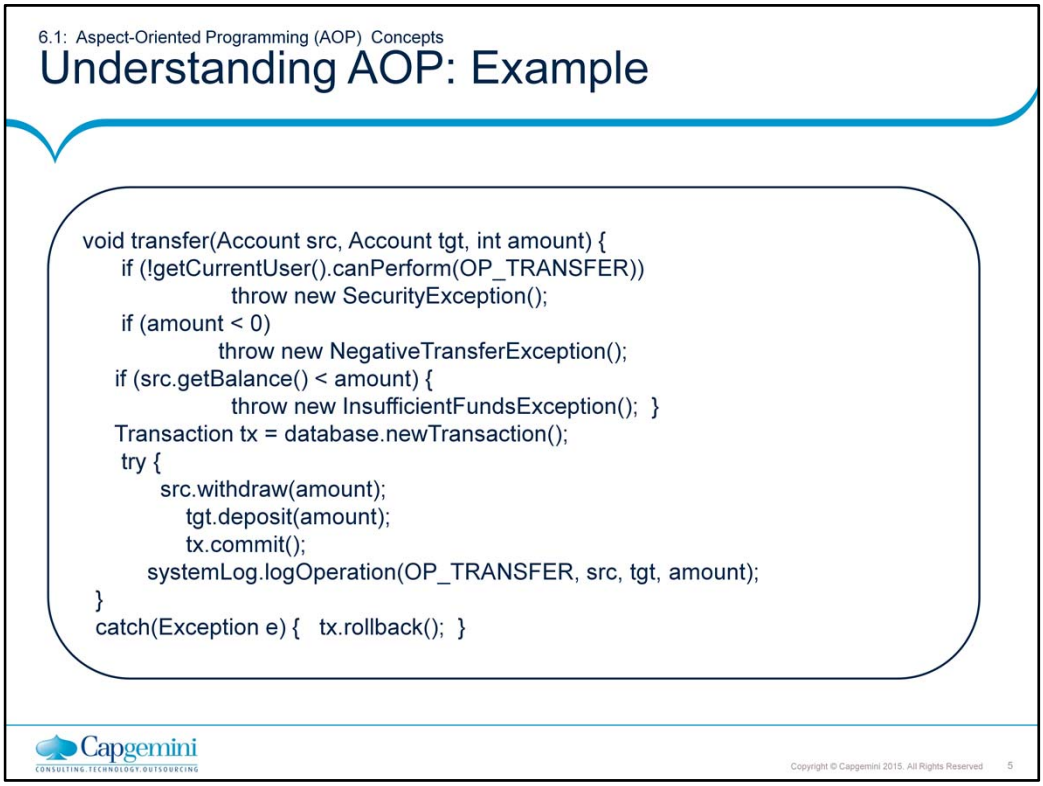
Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP literature.).

There are two distinct types of AOP: static and dynamic. In static AOP, like AspectJ's AOP (http://eclipse.org/aspectj/), the crosscutting logic is applied to your code at compile time and you cannot change it without modifying the code and recompiling. With dynamic AOP like Spring's AOP, crosscutting logic is applied dynamically at run time. This allows you to make changes in the distribution of cross-cutting without recompiling the application.

The AOP framework is one of the key components of Spring. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

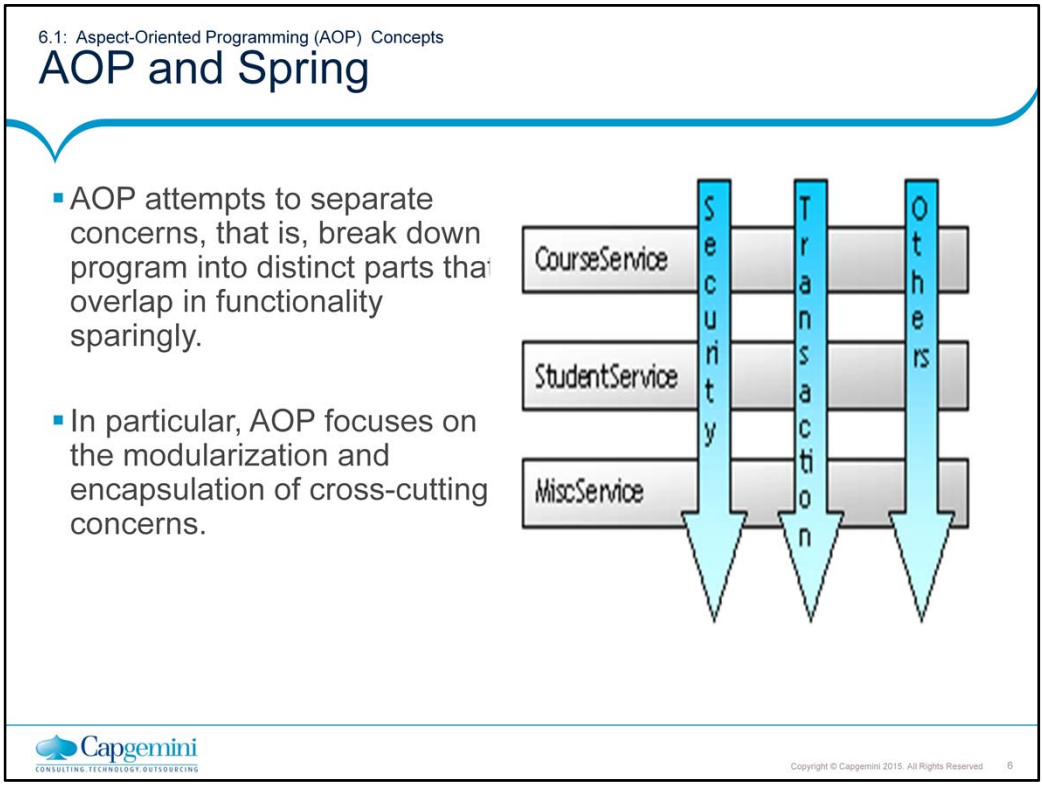


AOP ensures that enterprise service code does not pollute application objects.
For example, consider a banking application with a conceptually simple method for transferring an amount from one account to another as seen above.
The transfer method simply checks if sufficient funds are available in the source account and if ‘yes’, performs the transfer.
However, in a real-world banking application, this transfer method seems far from adequate. We must include security checks to verify that the current user has the authorization to perform this operation. We must enclose the operation in a database transaction to prevent accidental data loss. We must log the operation to the system log, and so on.



A very simplified version with all those new concerns would look somewhat like the code seen above.
We have now included a security check to verify that the current user has the authorization to perform this operation. We have also enclosed the operation in a database transaction in order to prevent accidental data loss.
The code has lost its elegance and simplicity because the various new concerns have become tangled with the basic functionality. The transactions, security, logging, etc. all exemplify cross-cutting concerns.
Therefore, we find that unlike the core concerns of the system, the cross-cutting concerns do not get properly encapsulated in their own modules. This increases the system complexity and makes maintenance considerably more difficult.
AOP attempts to solve this problem by allowing the programmer to develop cross-cutting concerns as full stand-alone modules called aspects.

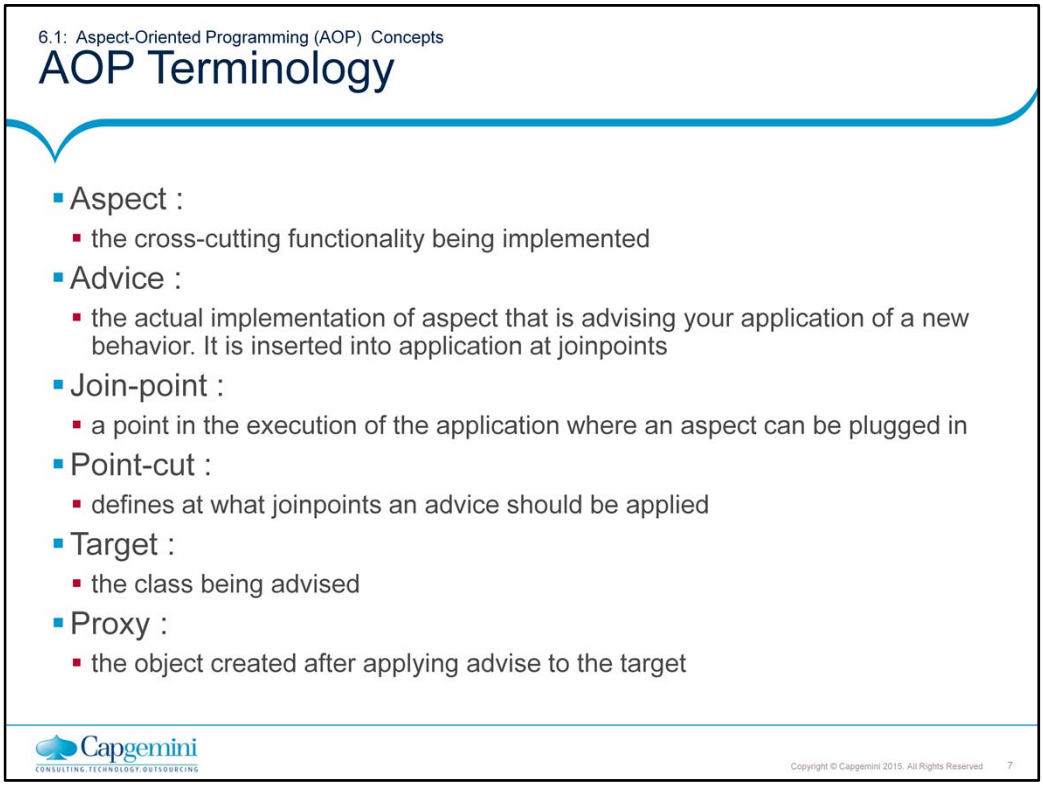
What is Aspect ? It is modularization of a concern. For example, if I have a method to which I want to apply transaction, then applying transaction is a concern. If I modularize then I can use this concern (aspect) to many beans and methods. These services can then be applied declaratively to the components.
Aspect-Oriented Programming has at its core the enabling of a better separation of concerns, by allowing the programmer to create cross-cutting concerns as first-class program modules.



Aspect Oriented programming is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often, however, these components also carry additional responsibility beyond their core functionality. System services like logging, transaction management and security often find their way into components whose core responsibility is something else. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.

AOP makes it possible to modularize services and then apply them declaratively to the components that they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved.

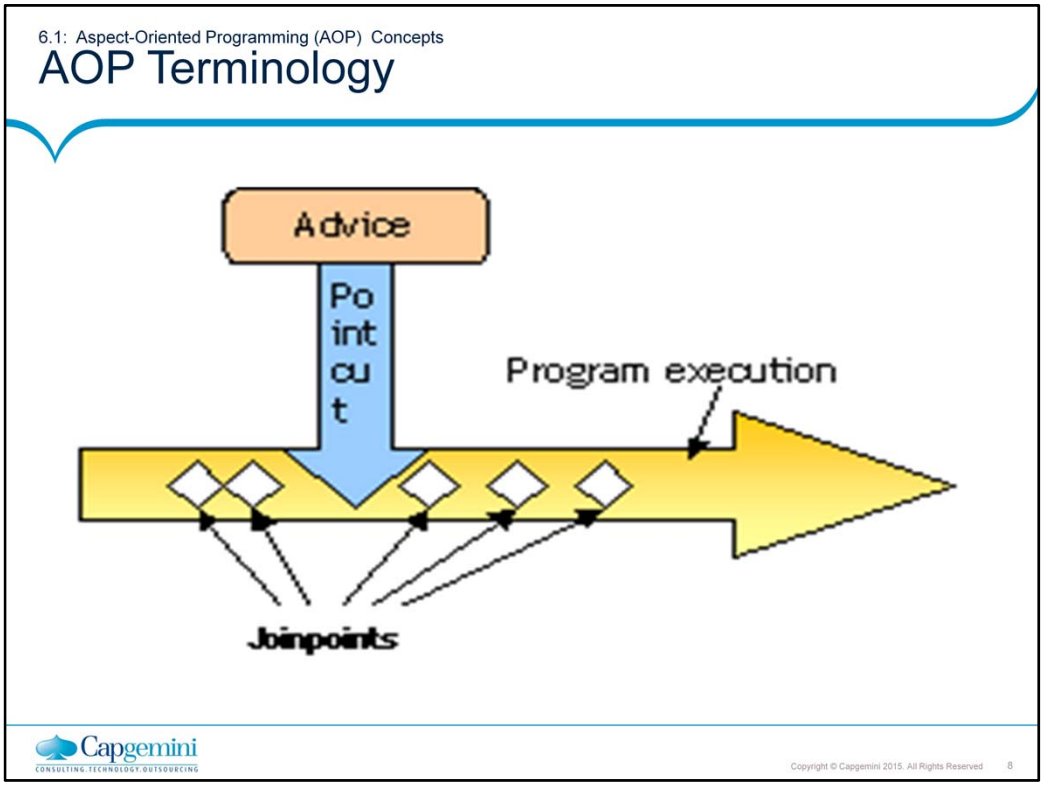
The above figure represents a typical application that is broken down into modules. Each module's main concern is to provide services for its particular domain. However, each of these modules also requires similar support functionalities such as security, logging and transaction management. AOP presents an alternative that can be cleaner in many circumstances. With AOP, you can still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you are applying the new feature. Cross-cutting concerns can now be modularized into special objects called aspects.



AOP Concepts:
Aspect : an aspect is the cross-cutting functionality you are implementing, that is, the area of your application you are modularizing. A simple example is logging. You can create a logging aspect and apply it throughout your application using AOP.
An aspect is also a combination of advice and pointcuts. This combination results in a definition of the logic that should be included in the application and where it should execute.

Joinpoint: this is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown or even a field being modified. These are points where your aspect's code can be inserted into the normal flow of your application to add new behavior.

Advice: this is the actual implementation of aspect, that is, the code that is executed at a particular joinpoint. It is advising your application of new behavior. There are many types of advices that we shall be seeing later.



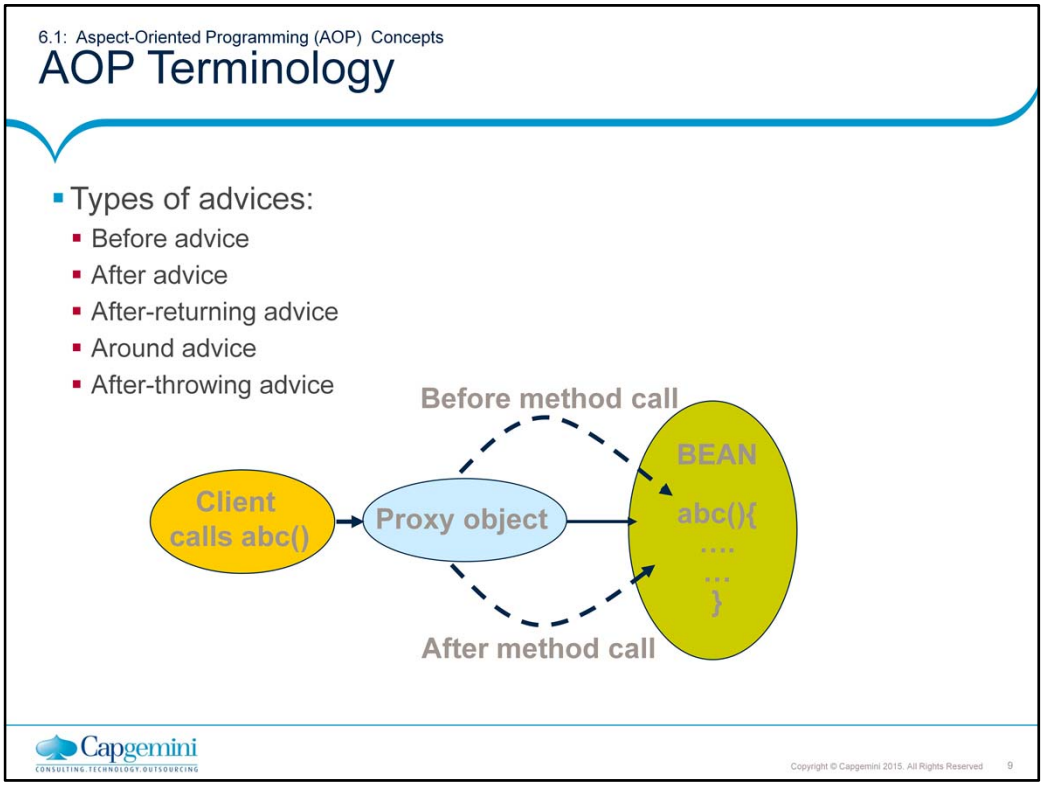
Point-cut: defines at what joinpoints an advice should be applied. Advice can be applied at any joinpoint supported by the AOP framework. By creating pointcuts, you gain fine-grained control over how you apply advice to the components in your application. A typical joinpoint is a method invocation. A typical pointcut is the collection of all method invocations in a particular class. You may not want to apply all aspects at all join-points. Point-cuts allow you to specify where you want advice to be applied.

Target: is the class being advised. Without AOP, this class would have to contain its primary logic plus the logic for any cross-cutting concerns. With AOP, this class is free to focus on its primary concern, oblivious to any advise being applied.

Proxy : is the object created after applying advise to the target. As far as client objects are concerned, the target object (pre-AOP) and proxy object (post-AOP) are the same.

Weaving : is the process of actually inserting aspects into the application code at the appropriate point.

In the above figure, the advice contains the cross-cutting behavior that needs to be applied. The join-points are well-defined points within the execution flow of the application that are candidates to have advice applied. The point-cut defines at what join-points that advice is applied.



More on Advice: In the figure above, the client calls a method abc() on target object. But proxy object intercepts the call. The core goal of a proxy is to intercept method invocations and where necessary, execute chains of advice that apply to a particular method.
Spring's join-point model is built around method interception. This means that the Spring advice will be woven into the application at different points around a method's invocation. Since there are several points during the execution of a method, there are several advice types:

Before—The advice functionality takes place before the advised method is invoked.
After—The advice functionality takes place after the advised method completes, regardless of the outcome.
After-returning—The advice functionality takes place after the advised method successfully completes.
After-throwing—The advice functionality takes place after the advised method throws an exception.
Around—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

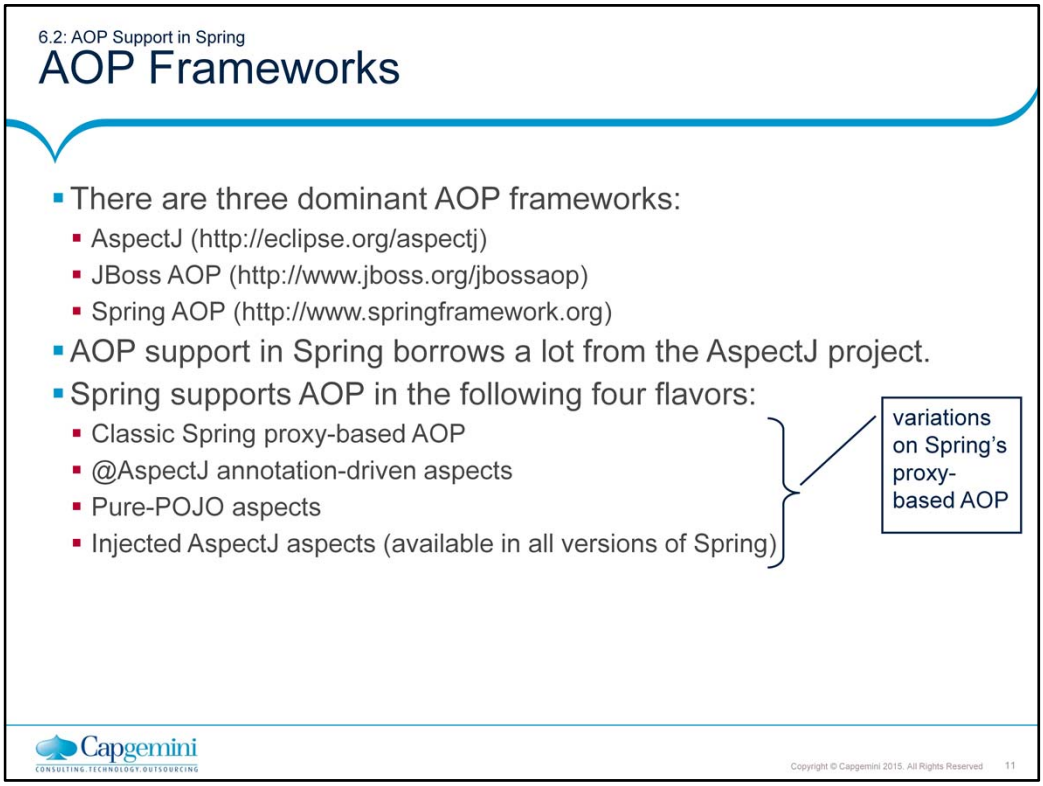
6.1: Aspect-Oriented Programming (AOP) Concepts

AOP Vs OOP

AOP	OOP
Aspect – code unit that encapsulates pointcuts, advice, and attributes	**Class** – code unit that encapsulates methods and attributes
Pointcut – define the set of entry points (triggers) in which advice is executed	**Method signature** – define the entry points for the execution of method bodies
Advice – implementation of cross cutting concern	**Method bodies** –implementation of the business logic concerns
Weaver – construct code (source or object) with advice	**Compiler** – convert source code to object code

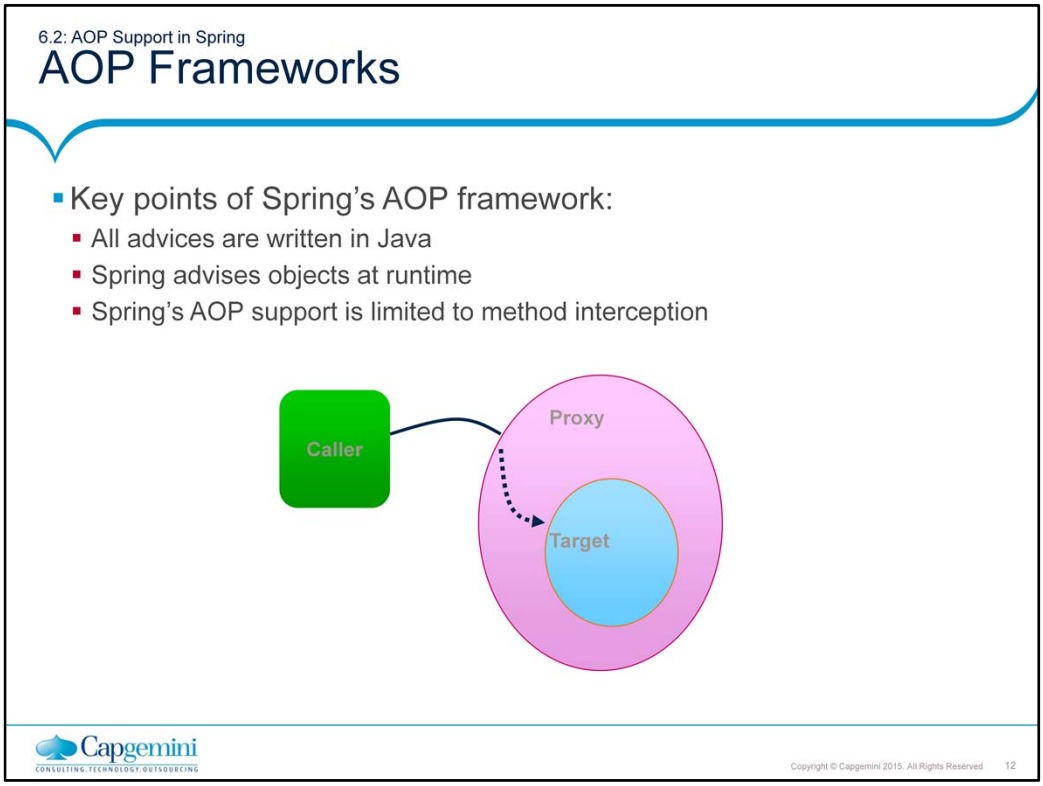
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 10

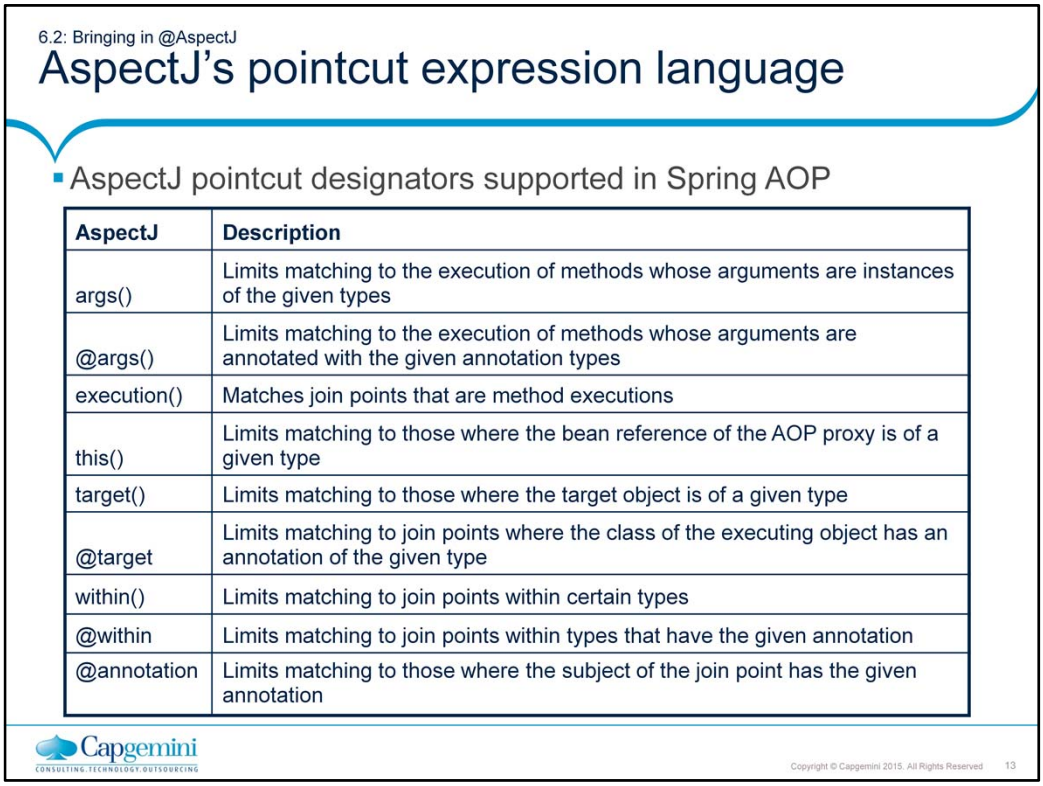


AOP frameworks differ from one another. Advice can be applied at the field modification level in some frameworks, whereas others only expose the join points related to method invocations. They may also differ in how and when they weave the aspects. In any case, ability to create pointcuts that define the join points at which aspects should be woven is what makes it an AOP framework.
The AOP frameworks have undergone some housecleaning in the last few years, resulting in some frameworks merging and others going extinct. Today, we have three dominant AOP frameworks (listed above)
We shall focus on Spring AOP, this being a Spring course. But, the AOP support in Spring borrows a lot from the AspectJ project. Thus, Spring's support for AOP comes in four flavors (listed above). The first three are limited to method invocation. If you want more, consider implementing aspects in AspectJ. You can take advantage of Spring DI to inject Spring beans into AspectJ aspects.

Today, Springs Classic Spring proxy-based AOP is no longer very popular, since Spring supports much cleaner and easier ways to work with aspects. Compared to simple declarative AOP and annotation-based AOP, Spring's classic AOP seems bulky and complex. Working directly with ProxyFactory- Bean can be wearying. Hence we will not cover this method in this course. However, you may refer to Appendix-D to understand the classis AOP API's better.



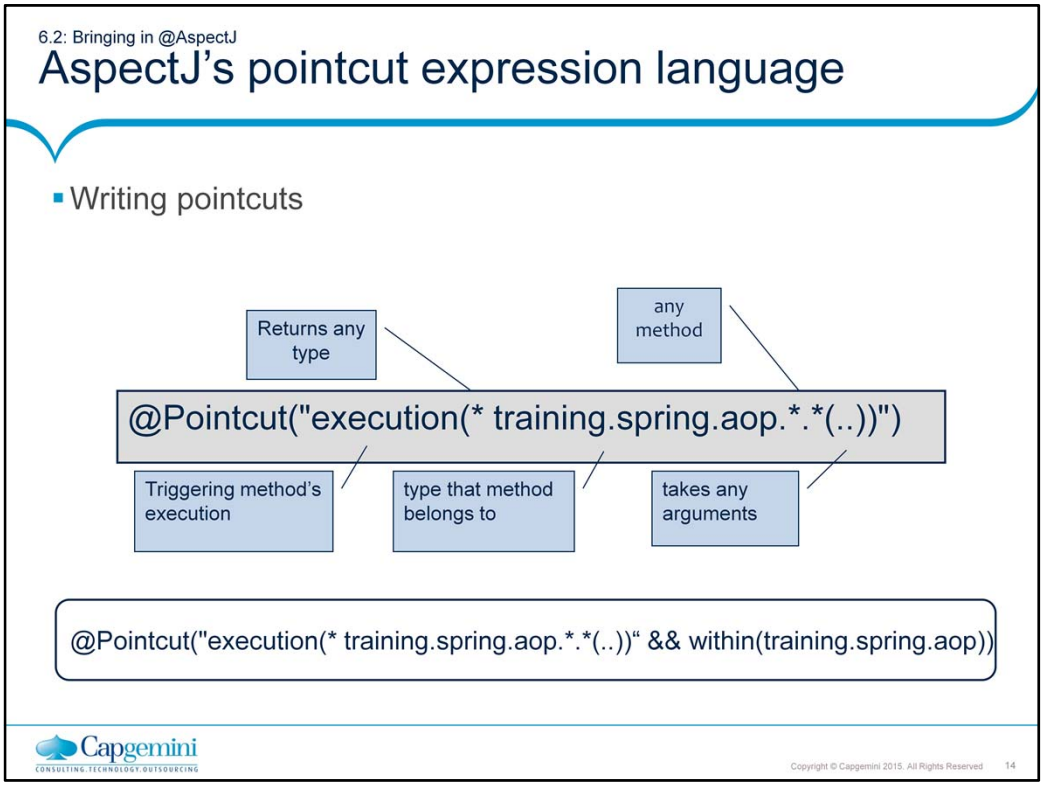
All advices are written in Java
Spring advises objects at runtime: Aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. See the figure above. The proxy class wraps around the target bean, intercepting advised method calls and forwarding those calls to the target bean. Spring doesn't create a proxied object until that proxied bean is needed by the application.
Spring's AOP support is limited to method interception : Some other AOP frameworks, such as AspectJ and JBoss, provide field and constructor join points in addition to method pointcuts. Since Spring does not support field pointcuts, you cannot create very fine-grained advice, such as intercepting updates to an object's field. And without constructor pointcuts, there is no way to apply advice when a bean is instantiated. Method interception is suitable for most needs.



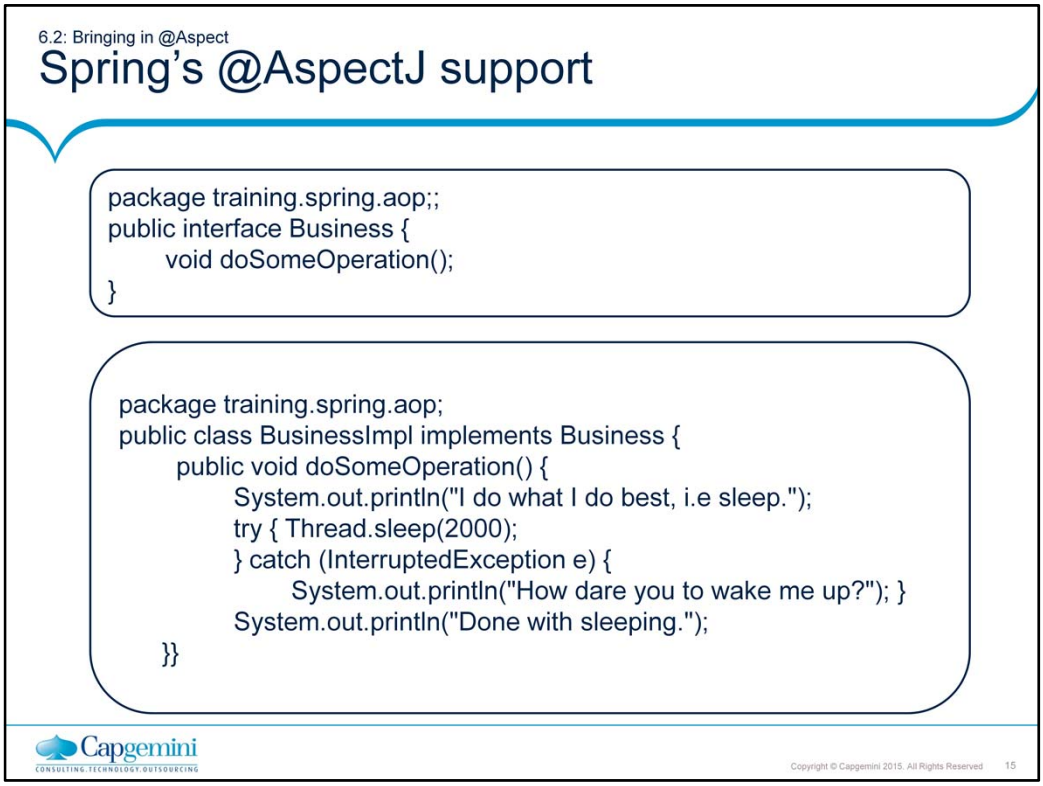
AspectJ	Description
args()	Limits matching to the execution of methods whose arguments are instances of the given types
@args()	Limits matching to the execution of methods whose arguments are annotated with the given annotation types
execution()	Matches join points that are method executions
this()	Limits matching to those where the bean reference of the AOP proxy is of a given type
target()	Limits matching to those where the target object is of a given type
@target	Limits matching to join points where the class of the executing object has an annotation of the given type
within()	Limits matching to join points within certain types
@within	Limits matching to join points within types that have the given annotation
@annotation	Limits matching to those where the subject of the join point has the given annotation

Pointcuts are used to pinpoint where an aspect's advice should be applied. Pointcuts are one of the most fundamental elements of an aspect. In Spring AOP, pointcuts are defined using AspectJ's pointcut expression language.

Note : Spring only supports a subset of the pointcut designators available in AspectJ. The table above lists out the AspectJ pointcut designators that are supported in Spring AOP. All of these limit Join point matches to a particular scenario. Only the execution designator actually performs matches. The other designators are used to limit those matches.

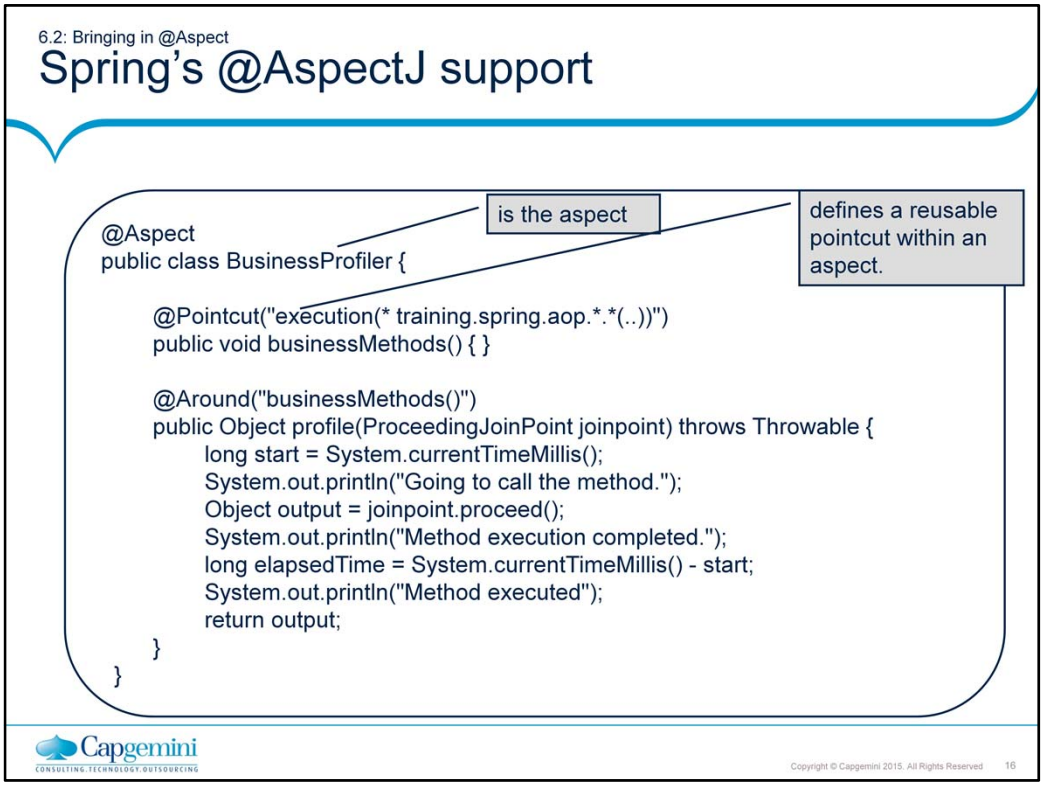


Writing Pointcuts: The pointcut expression shown above is used to apply advice whenever any method of any class in the training.spring.aop package is executed. The method specification starts with an asterisk, which indicates that we do not care what type the method returns. We need to specify the fully qualified class name and the name of the method we want to select. In the method's parameter list, the double-dot (..) indicates that the pointcut should select any method, no matter what the argument list is.
To confine the reach of that pointcut to only the training.spring.aop package, use the within() designator as seen in example above.

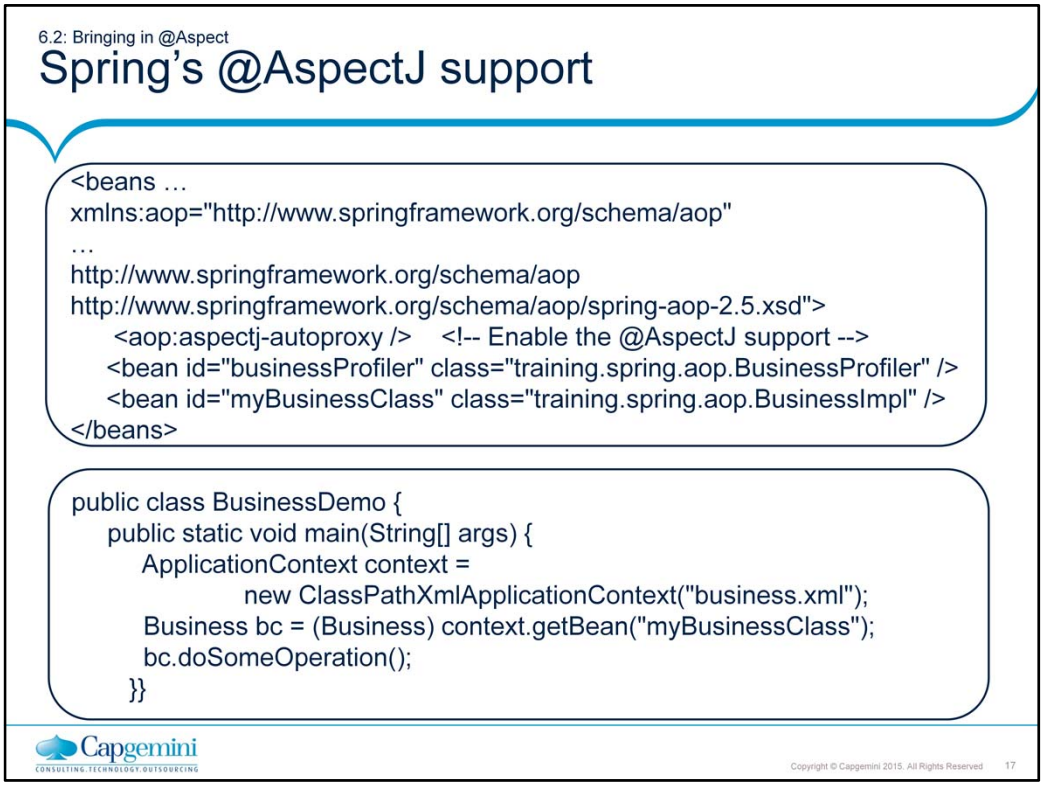


@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. When the @AspectJ support is enabled, any bean in your container that has the @Aspect annotation will be automatically detected by Spring and used to configure Spring AOP.
To use @AspectJ aspects in a Spring configuration you need to enable Spring support, and autoproxying beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.
The @AspectJ support is enabled by including the following element inside your spring configuration:
<aop:aspectj-autoproxy/>
For this, you will need to bring in the necessary schema support. This is seen in the XML that is part of the subsequent demo.
Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut and advice.

The code above shows the BusinessImpl class that contains business logic. We shall add Spring AOP to profile our business methods.

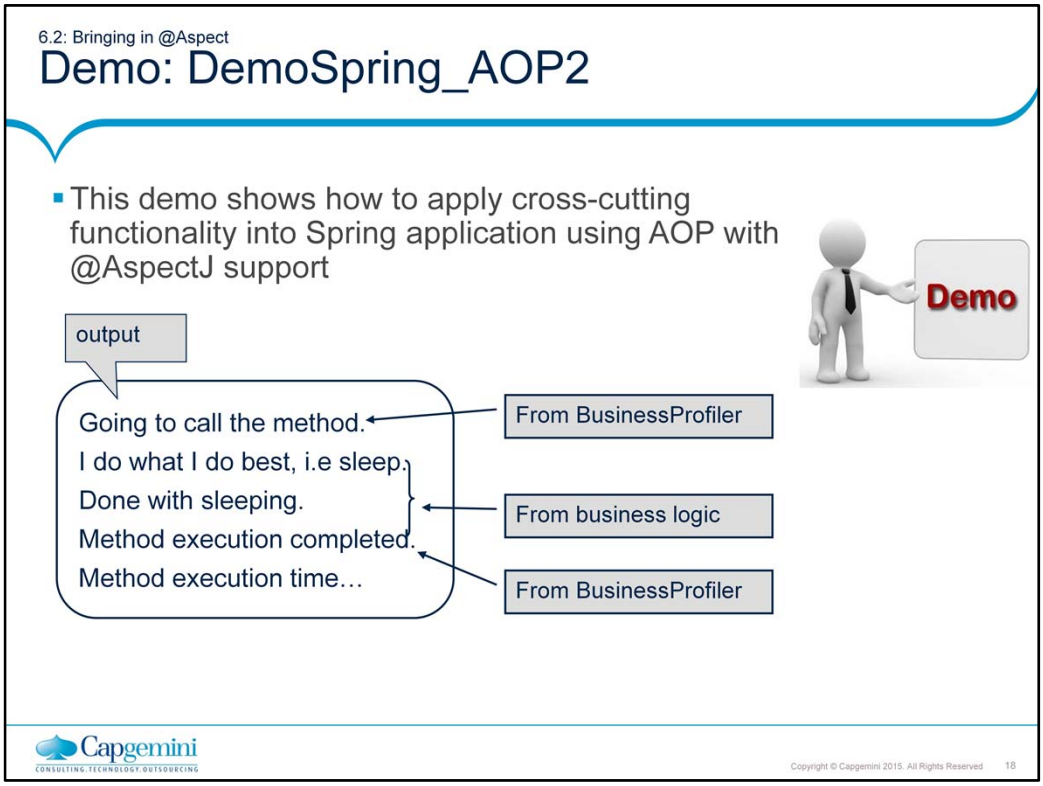


See the code above.
Using @AspectJ annotation, we have declared that this class is an Aspect. This will profile our business method
Using @Pointcut annotation, we have defined a pointcut that will match the execution of all public method inside training.spring.aop package.
Using @Around annotation, we have defined a Around advice which will be invoked before and after our business method. The pjp.proceed() method calls our business method from @Around advice.



In the Spring configuration file above, we have:
Added the necessary AOP schemas.
Enabled the @AspectJ support to our application using <aop:aspectj-autoproxy />
Defined two normal Spring beans – one for our Business class and the other for Business Profiler (i.e. our aspect).

<aop:aspectj-autoproxy/> will automatically generate proxy beans whose methods match the pointcuts defined with @Pointcut annotations in @Aspect-annotated beans. To use the <aop:aspectj-autoproxy> configuration element, you'll need to remember to include the aop namespace in your Spring configuration file.



Please refer to demo, DemoSpring_AOP2.
Package -> com.igate.aop
Execute the BusinessDemo.java class

6.3: Schema-based AOP support

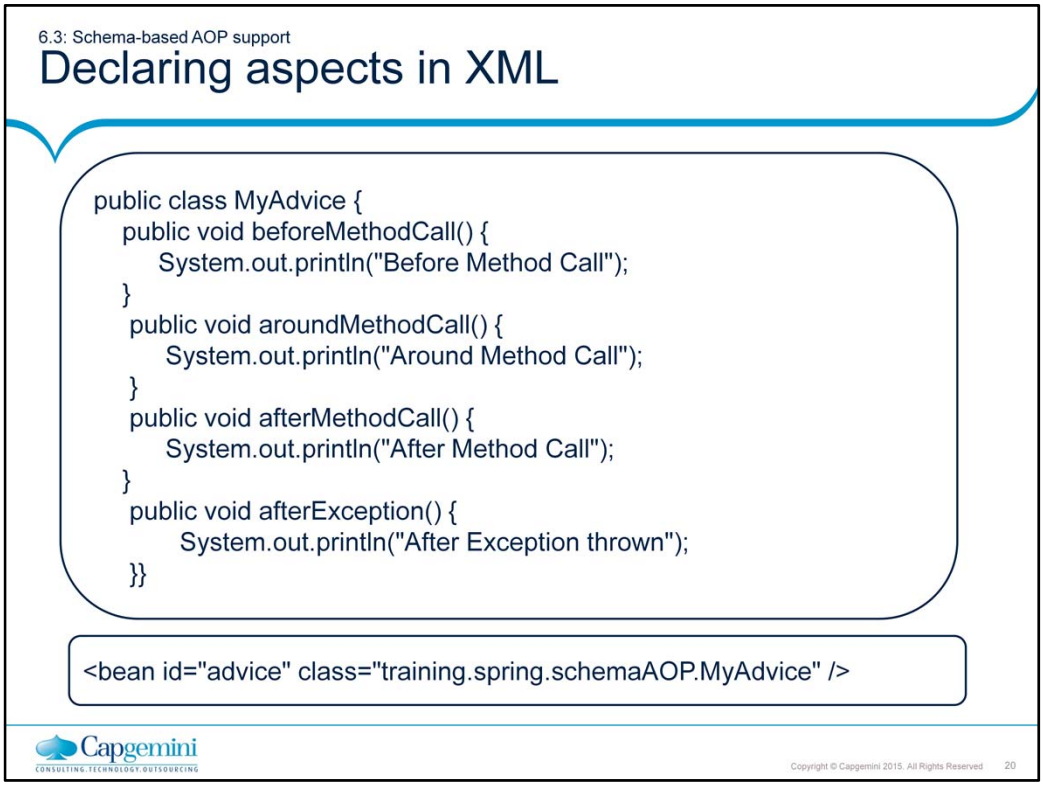
Declaring aspects in XML

AOP config element	Purpose
<aop:advisor>	Defines an AOP advisor.
<aop:after>	Defines an AOP after advice (regardless of whether the advised method returns successfully).
<aop:after-returning>	Defines an AOP after-returning advice.
<aop:after-throwing>	Defines an AOP after-throwing advice.
<aop:around>	Defines an AOP around advice.
<aop:aspect>	Defines an aspect.
<aop:aspectj-autoproxy>	Enables annotation-driven aspects using @AspectJ.
<aop:before>	Defines an AOP before advice.
<aop:config>	The top-level AOP element. Most <aop:*> elements must be contained within <aop:config>.
<aop:declare-parents>	Introduces additional interfaces to advised objects that are transparently implemented.
<aop:pointcut>	Defines a pointcut.

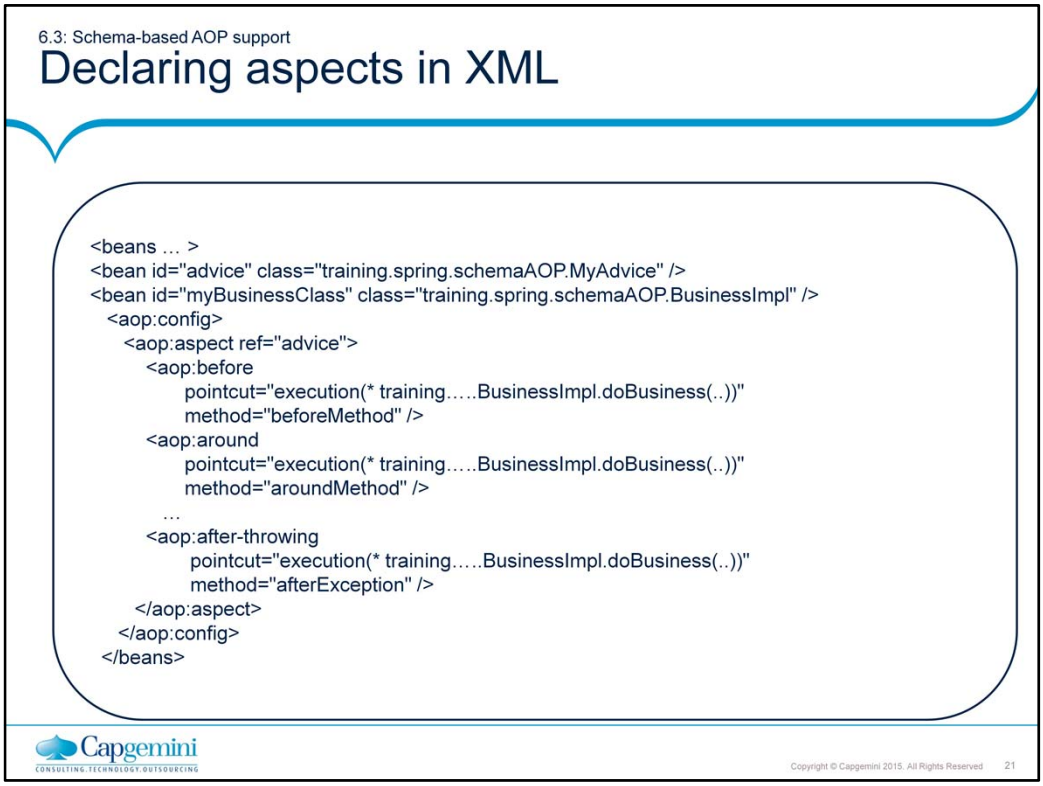
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 19

From version 2.0 onwards, Spring offers support for defining aspects using the new "aop" namespace tags. The pointcut expressions and advice kinds supported are similar to @AspectJ style.
To use the aop namespace tags, you need to import the spring-aop schema. This is seen in the XML that is part of the subsequent demo.
Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (multiple <aop:config> element can exist). An <aop:config> element can contain pointcut, advisor, and aspect elements, in that order!

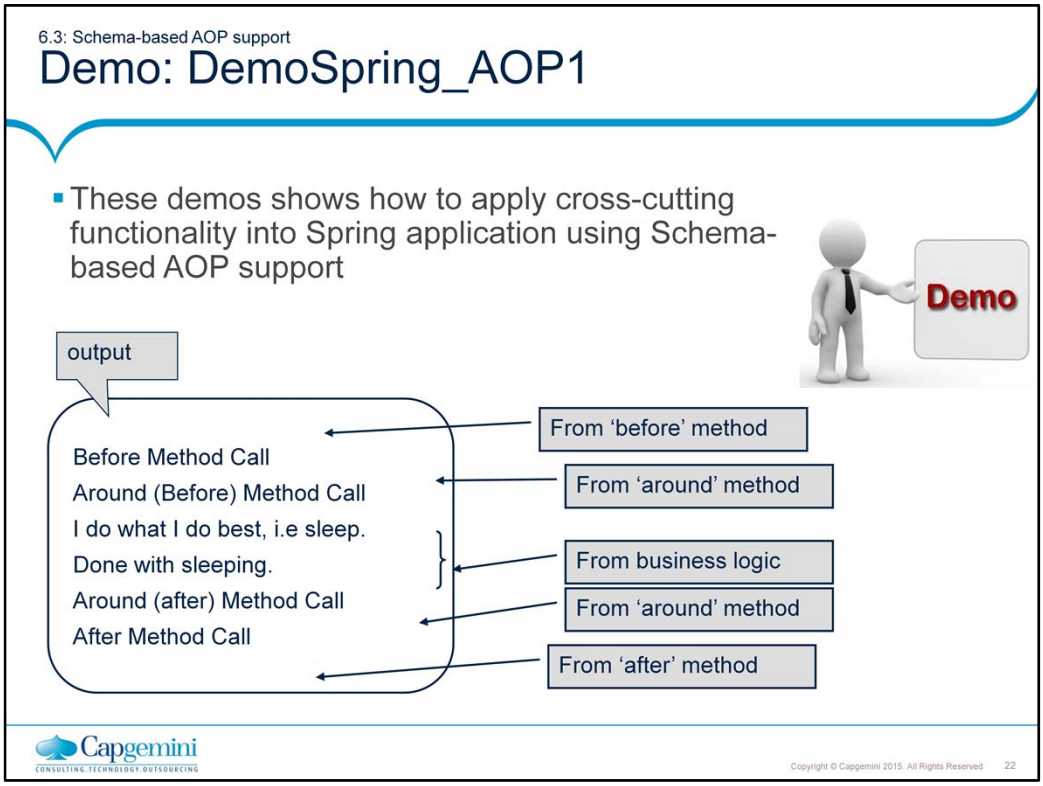


Let us see an example of using Spring's aop configuration namespace. The code listing above is very straightforward and simply creates functions of an advice. It's a POJO with a handful of methods. Thus we can register it as a bean in the Spring application context like any other class. See above for how.
The MyAdvice class seems an Aspect but needs Spring's AOP magic to become one.



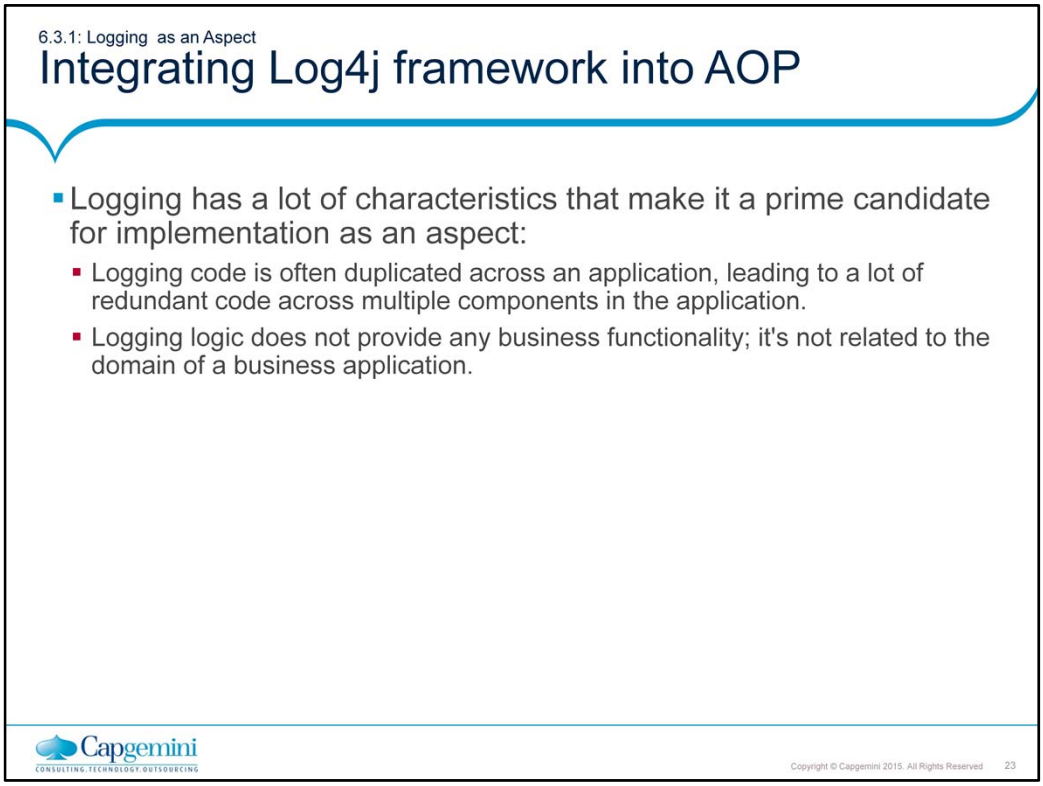
Using the Spring's AOP configuration elements, as shown in the listing above, we have converted the MyAdvice POJO into an aspect!
Notice that most of the configuration elements are used within the context of the <aop:config> element. Within <aop:config> you may declare one or more advisors, aspects, or pointcuts. In our example, we have declared a single aspect using the <aop:aspect> element. The ref attribute references the POJO bean (MyAdvice in this case) that will be used to supply the functionality of the aspect ie methods called by any advice in the aspect.
The aspect has several bits of advice. The <aop:before> element calls aspect method (beforeMethod()) before the methods matching the pointcut are executed. The <aop:after-throwing> element defines an after-throwing advice to call the afterException() if any exceptions are thrown.
In all advice elements, the pointcut attribute defines the pointcut where the advice will be applied. Notice that the value of the pointcut attribute is the same for all of the advice elements. That's because all of the advice is being applied to the same pointcut.
The pointcut expression is repeated throughout the code. To avoid this, you can use <aop:pointcut> element to define a named pointcut that can be used by all of the advice elements.

```
<aop:aspect ref="advice">
    <aop:pointcut id="pt" expression=
        "execution(* training.spring.schemaAOP.BusinessImpl.doBusiness())" />
<aop:before pointcut-ref="pt" method="beforeMethod" />
```



Please refer to demo, DemoSpring_AOP1 -> com.igate.aop
Execute the BusinessDemo.java class

We have modified the aroundMethod() method in MyAdvice.java a bit to add the proceed() method. The ProceedingJoinPoint object allows us to invoke the advised method from within our advice. The advice method will do everything it needs to do and, when it's ready to pass control to the advised method, it'll call ProceedingJoinPoint's proceed() method. If you don't include the call to proceed() method, your advice will block access to the advised method.



Logging has a lot of characteristics that make it a prime candidate for implementation as an aspect. The following are two of the notable characteristics:
Logging code is often duplicated across an application, leading to a lot of redundant code across multiple components in the application. Even if the logging logic is abstracted to a separate module so that different components have a single method call, that single method call is duplicated in multiple places.
Logging logic does not provide any business functionality; it's not related to the domain of a business application.
Spreading the logging across multiple components introduces some complexity to the code. Business objects not only perform business tasks, they also take care of logging functionality (and other such crosscutting concerns, such security, transaction, caching, etc.).
AOP makes it possible to modularize these crosscutting services and then apply them declaratively to the components that they should affect.

Let us see a simple example. We shall use a around advice AOP advice to intercept a method in the business class and log the operation at DEBUG level.

6.3.1: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

```
public interface SampleInterface {
   public void process();
   public String getName();
   public int getAge();
   public void setAge(int age);
   public void setName(String str);
}
```

Business interface and its implementing class

```
public class SampleBean implements SampleInterface {
   private String name;
   private int age;
   //getter/setter methods for these properties

   public void process() {
         System.out.println("checking with the process() method-1");
    }
}
```

Business method

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 24

6.3.1: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

The interceptor

```
public class LoggingInterceptor {
   Logger myLog;
   public Object logs(ProceedingJoinPoint call) throws Throwable {
      Object point = null;
      myLog = Logger.getLogger(LoggingInterceptor.class);
      PropertyConfigurator.configure("log4j.properties");
      try {
         System.out.println("from logging aspect: entering method "
                                          + call.getSignature().getName());
         myLog.info("Hello : It is " + new java.util.Date().toString());
         point = call.proceed();
         System.out.println("from logging aspect: exiting method ");
      } catch (Exception e) {
         System.out.println("Logging the exception with date " + new Date());
       }
       return point;
   }
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 25

6.3.1: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

The configuration file

```
<beans  …..>
   <bean id="sampleBean" class="training.spring.aop.logger.SampleBean"/>
   <bean id="loggingInterceptor"
            class="training.spring.aop.logger.LoggingInterceptor" />
   <aop:config>
      <aop:aspect ref="loggingInterceptor">
         <aop:pointcut id="myCutLogging" expression="execution(* *.p*(..))"/>
            <!-- - when you want to do? before method ,after method,..... -->
          <aop:around pointcut-ref="myCutLogging" method="logs" />
       </aop:aspect>
    </aop:config>
</beans>
```

log4j.properties

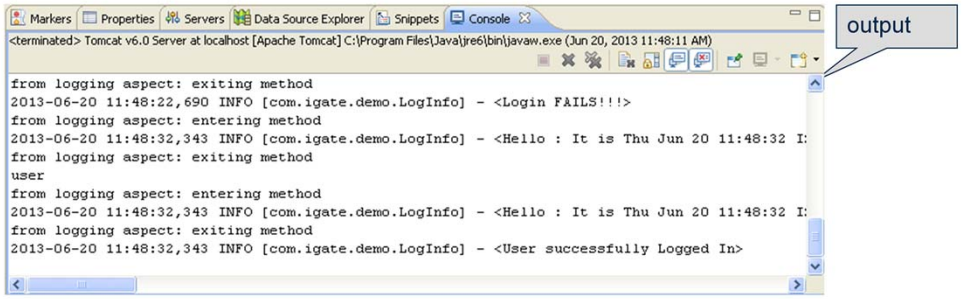
```
log4j.rootLogger=debug, myAppender
log4j.appender.myAppender=org.apache.log4j.ConsoleAppender
log4j.appender.myAppender.layout=org.apache.log4j.SimpleLayout
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 26

Demo: DemoMVC_AOP

- This demo shows how to integrate the Log4j logging framework with AOP using MVC based an application



Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 27

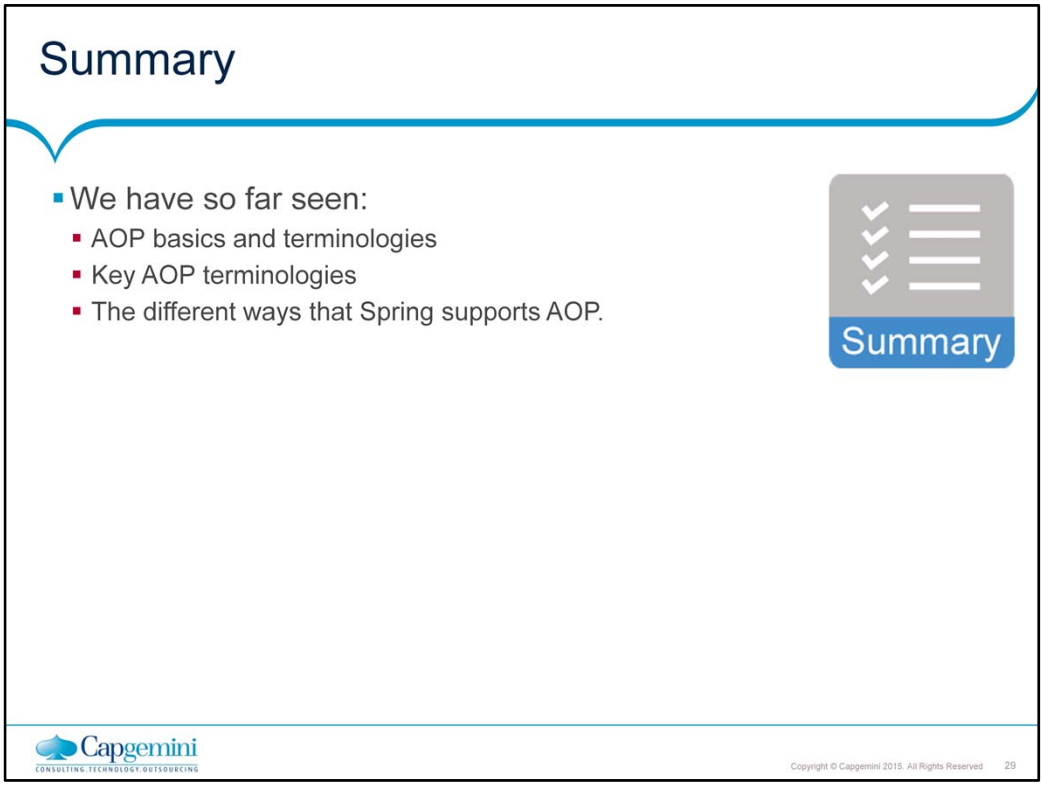
Please refer to demo, DemoMVC_AOP

Lab

- Lab-3 from the lab guide

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 28



AOP is a powerful complement to object-oriented programming. With aspects, you can now group application behavior that was once spread throughout your applications into reusable modules. You can then declaratively or programmatically define exactly where and how this behavior is applied. This reduces code duplication and lets your classes focus on their main functionality. Thus, the AOP technique helps you add design and run-time behavior to an object model in a non-obtrusive manner by using static and dynamic crosscutting.

The first thing that comes to mind when someone mentions AOP is logging, followed by transaction management. However, these are just special applications of AOP. AOP can be used for performance monitoring, call auditing, caching and error recovery too. More advanced uses of AOP include compile-time checks of architecture standards. For example, you can write an aspect that will enforce you to call only certain methods from certain classes.

Today, Spring AOP offers fine grained object advising capabilities using AspectJ support.

Review Questions

- Question 1: In Spring's aop configuration namespace, how is an aspect defined?
  - Option 1 : <aop:advisor>
  - Option 2 : <aop:aspect>
  - Option 3 : <aop:declare-aspect>
  - Option 4 : <aop:config>

- Question 2 : In addition to method join points, Spring also supports field and constructor joinpoints.
  - Option 1 : True
  - Option 1 : False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 30

Review Questions

- Question 3 : ProceedingJoinPoint's _____ method must be used to provide access to the advised method, so that it can execute.
  - Option 1 : invoke()
  - Option 2 : continue()
  - Option 3 : proceed()
  - Option 4 : next()

- Question 4 : <aop:aspectj-autoproxy/> automatically proxies beans whose methods match the pointcuts defined with @Pointcut annotations in @Aspect-annotated beans.
  - Option 1 : True
  - Option 1 : False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 31

Review Questions

S	T	A	R	G	E	T	P
J	P	D	B	C	A	A	R
N	R	V	I	U	T	S	O
P	O	I	N	T	C	U	T
R	C	C	V	W	E	Y	E
O	E	E	O	Q	P	F	E
X	E	T	K	G	S	M	G
Y	D	E	E	G	A	H	E

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 32

There are some AOP terms hidden in the above table. The words may be across, down or even upside down.
the cross-cutting functionality being implemented
the actual implementation of aspect that's advising your application of new behavior
defines at what joinpoints, an advice should be applied
the class being advised
the object created after applying advise to the target
A method of the MethodInterceptor interface
Helps to go to the next interceptor in the chain

Basic Spring 4.0

Document Revision History

Date	Revision No.	Author	Summary of Changes
Aug-2012	2.0	Mohan C	Lab book exercises are revamped
June-2013	3.0	Mohan C	Lab book exercises are revamped
June-2015	4.0	Rathnajothi.P	Upgraded from spring version 3 to 4
May-2016	5.0	Vinod Satpute	Revamped as per the integrated ELT TOC
June- 2016	6.0	Vinod Satpute Yukti Valecha Tanmaya Acharya	Modified as per Toc for ELTP

Table of Contents

Getting Started

Overview

This lab book is a guided tour for learning Basic Spring 4.0. It comprises solved examples and 'To Do' assignments. Follow the steps provided in the solved examples and work out the 'To Do' assignments given.

Setup Checklist for Spring Framework

Here is what is expected on your machine in order for the lab assignments to work.

Minimum System Requirements

- Intel Pentium IV or higher
- Microsoft Windows (NT 4.0/XP/2K)
- Memory: 256MB of RAM (512 recommended)
- 500MB hard disk space
- JDK version 1.8 + with help, Netscape or IE
- MS-Access/Connectivity to Oracle database
- Wildfly
- Eclipse Luna
- Spring4.0 API from https://spring.io/docs. Download spring-framework-4.0.3.RELEASE-with-docs.zip, which contains the documentation alsoand unzip it.

Creating the first Spring application:

- ✓ Ensure that Java 8 is installed and Eclipse Luna is available.
- ✓ You will need Wildfly server to work with.
- ✓ Unzip the spring-framework-4.0.3.RELEASE-with-docs.zip into any folder.
- ✓ Create a new project in eclipse and name it.

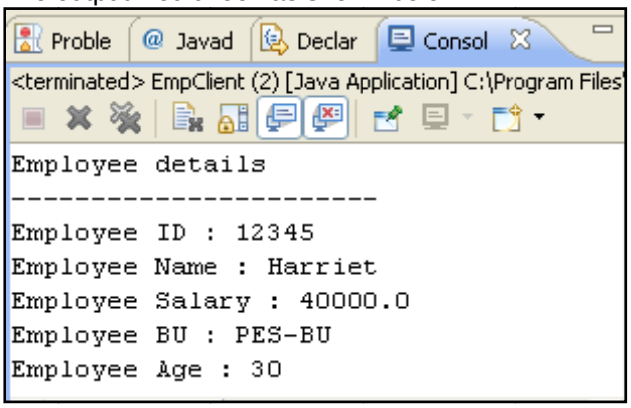
Lab 1. Injecting dependencies into a Spring application

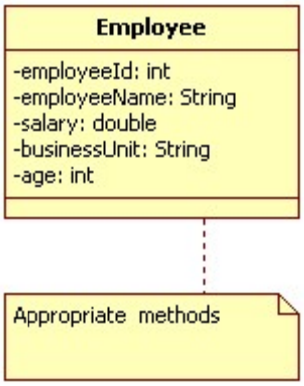
Goals	• Using IoC to integrate disparate systems in a loosely coupled manner.
Time	180 minutes

Problem statement-1.1: Injecting dependencies

Write an Employee bean. Inject values into bean using DI and display all values. Refer the class diagram below

The output would look as shown below:



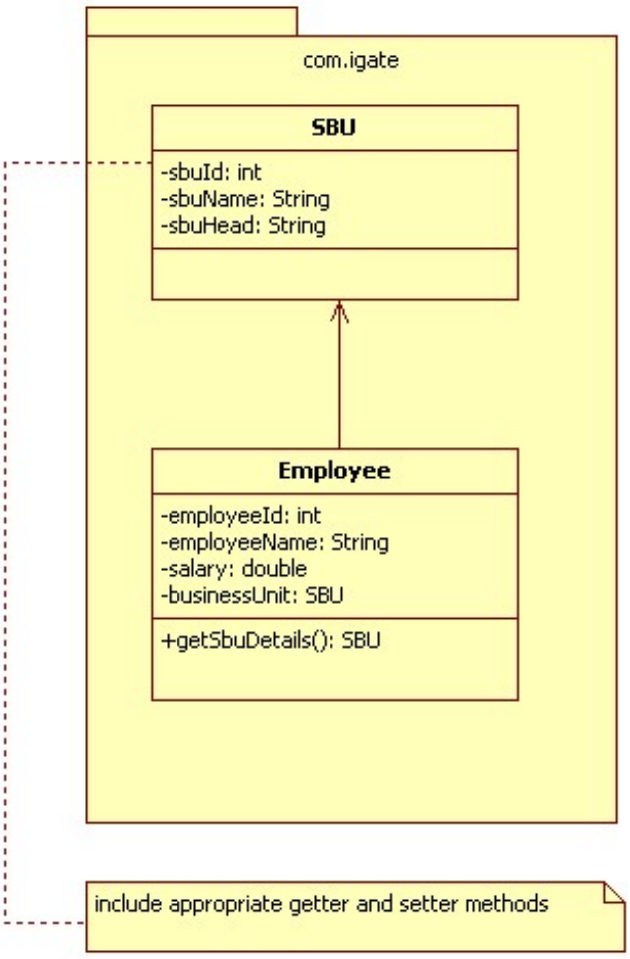


Class Diagram 1: Employee

Keep each of the lab solutions separate, preferably in different packages/source folders

Problem statement-1.2: Injecting dependencies

Code SBUbean.Revisit the Employee bean and provide a method to retrieve SBU details (getSBUDetails()) for the employee. You will need to inject the SBU bean to the Employee bean as shown in the Class diagram below:



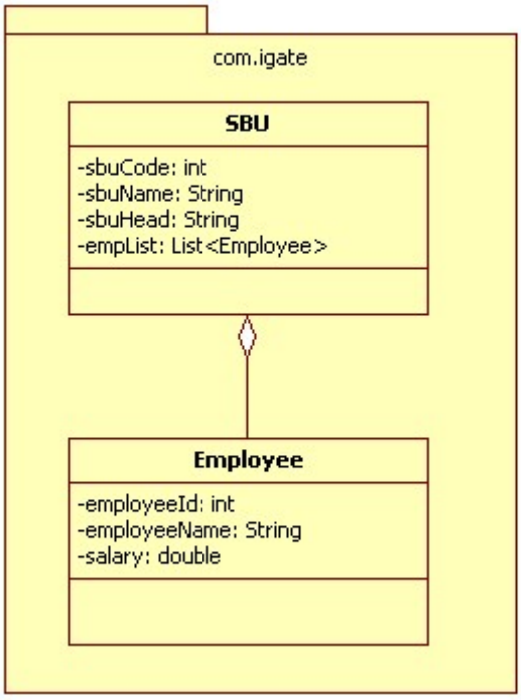
Class Diagram 2: **SBU and Employee**

The output would look as shown below:

```
Problems  Javadoc  Declaration  Console
<terminated> EmpClient (3) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 20, 2012 4:30:57 PM)
Employee details
------------------------
Employee [empAge=40, empId=12345, empName=Harriet, empSalary=40000.0
sbu details=SBU [sbuCode=PES-BU, sbuHead=Kiran Rao, sbuName=Product Engineering Services]]
```

Problem statement-1.3: Injecting dependencies

Revisit the SBU bean. Create a new property called empList which will contain a list of all employees in the PES BU. Display the SBU details, followed by a list of all employees in that BU. To inject employee objects into the SBU bean, use "List" collection. Allocate two employees to PES. Refer Class diagram below



Class Diagram 3: **SBU and Employee (Ver -2)**

The output would look as shown below:

```
Problems  Javadoc  Declaration  Console
<terminated> EmpClient (4) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 20, 2012 4:53:43 PM)
SBU details
-----------------------
sbuCode=PES-BU, sbuHead=Kiran Rao, sbuName=Product Engineering Services
 Employee details:---------------
[Employee [empAge=30, empId=12345, empName=Harriet, empSalary=40000.0], Employee [empAge=30, empId=12346
```

Problem statement-1.4: Injecting dependencies

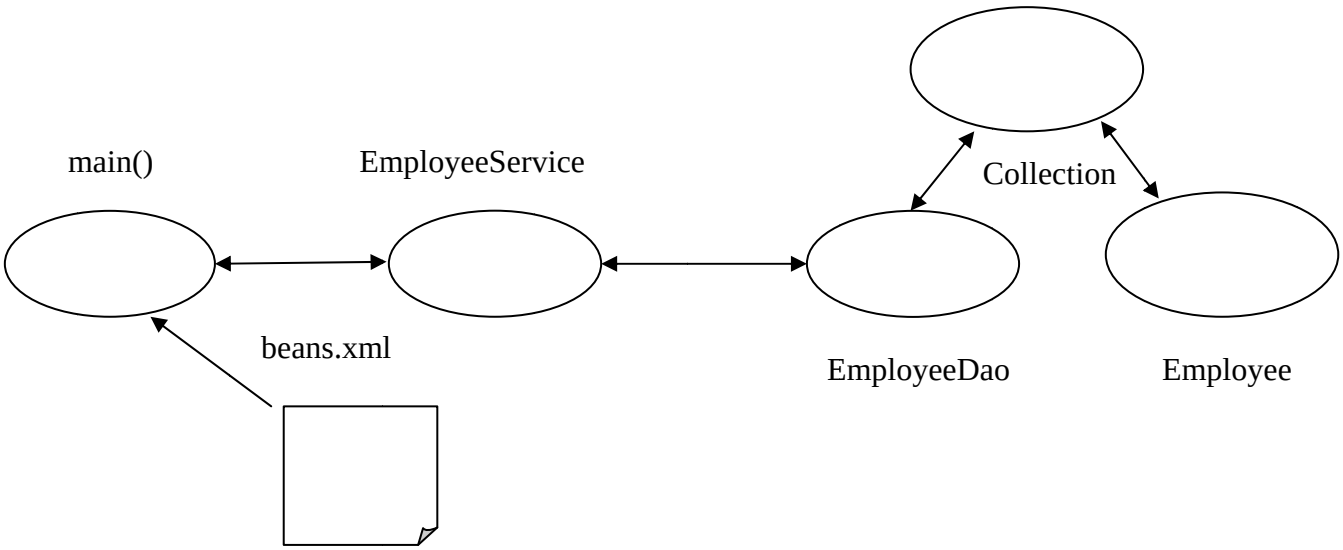
Develop a console based spring application where main method of client class will retrieve employee information from Employee collection and displays info in the console as shown:

Input:
Employee ID : 100

Output:
Employee Info:
Employee ID :100
Emplpoyee NAME :Rama

Employee SALARY :12345.67

Refer diagram below for implementation details:



Flow Diagram - 1

Note: implement above application using

- Setter Injection
- Constructor Injection (use index and type attribute with constructor arg tag)
- Use different bean wiring mechanism like..
  - By Name
  - By Type
  - Auto Wiring

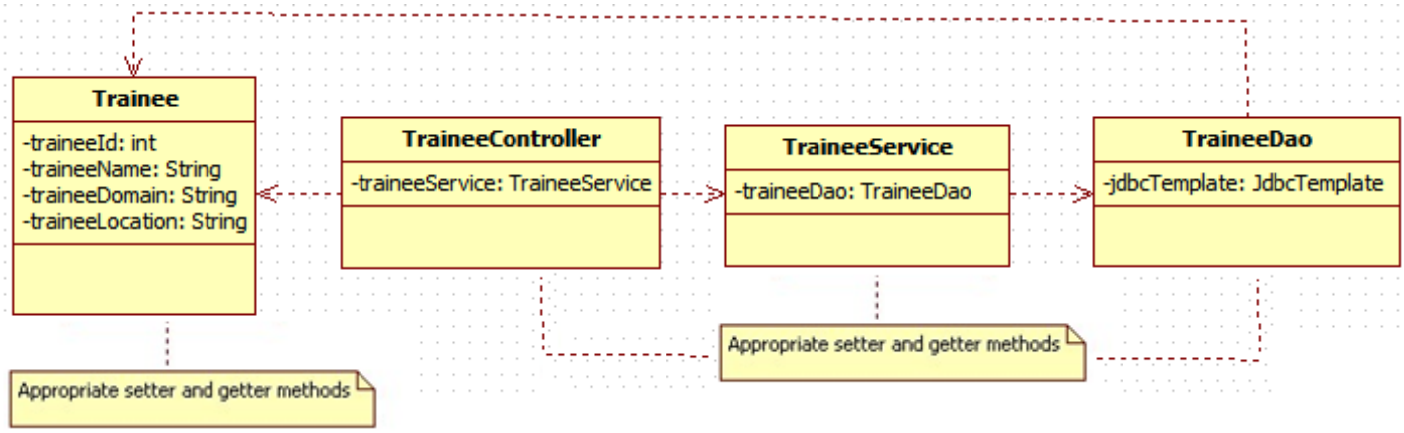
Change Request: Now Change the application so that all components are autowired and components are automatically scanned.Use Spring boot API.

Lab 2. Spring MVC and JPA

Goals	• Demonstrate Spring's MVC framework • Integrate Spring and JPA
Time	180 minutes

Version -1:

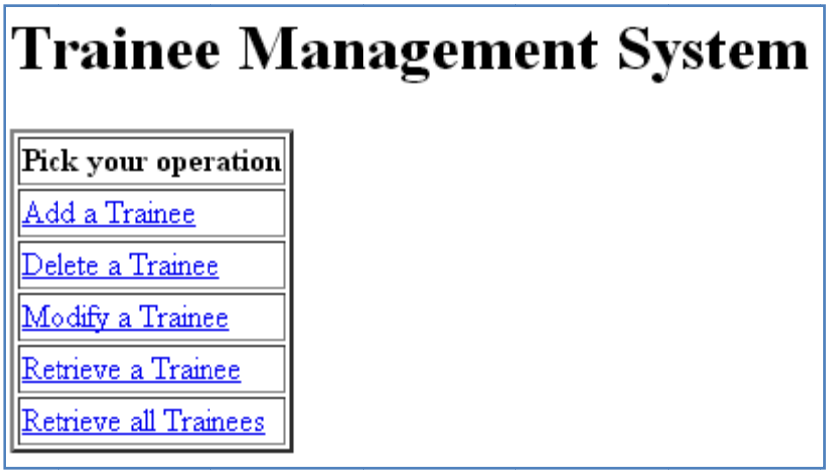
Develop a Spring MVC based application to manage list of trainees by an admin. Refer the class diagram below to develop required classes.



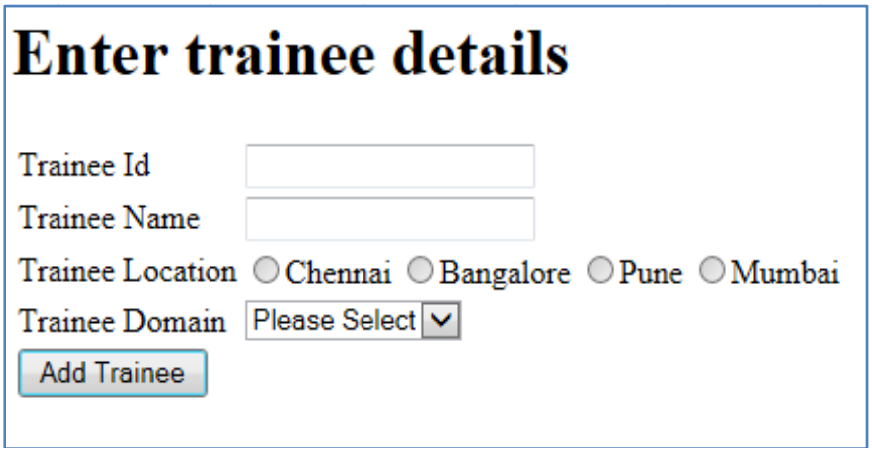
Class Diagram 4**: Trainee related Classes**

Initially when the application is deployed login page should come up as shown below.

Administrator will enter valid credentials to go to the page shown below.

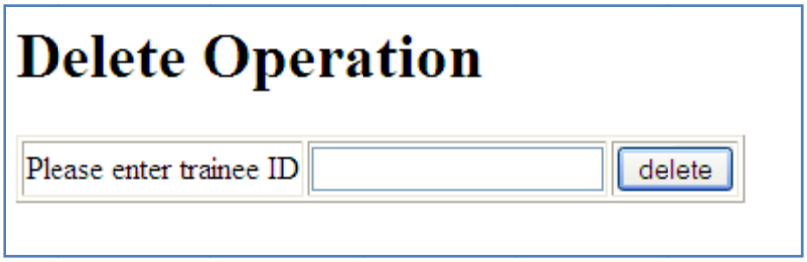


Admin select add hyperlink to add a new trainee. The following page shows up.

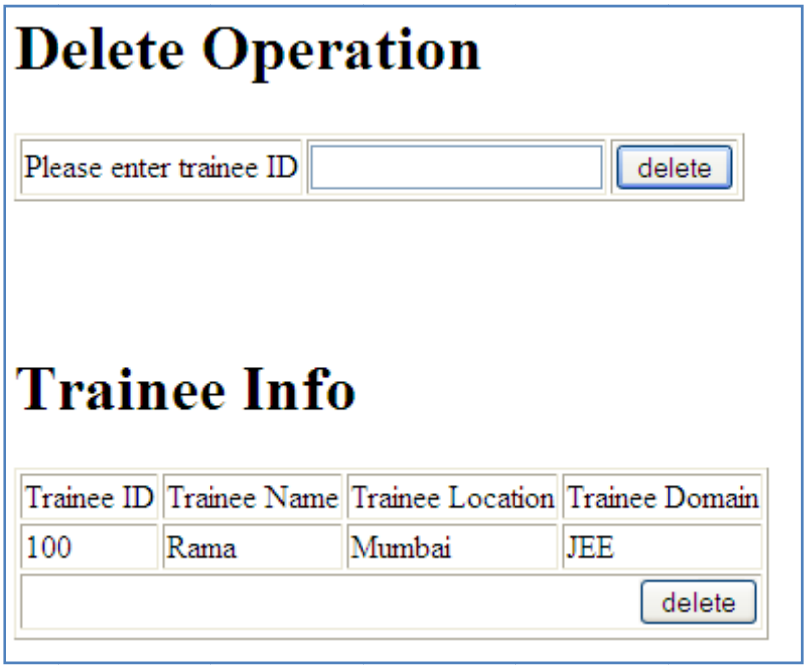


After entering trainee details admin will submit, to insert trainee info into database.

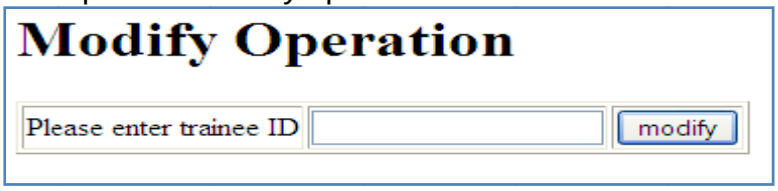
If admin selects delete operation refer the following screen shots to design your application.



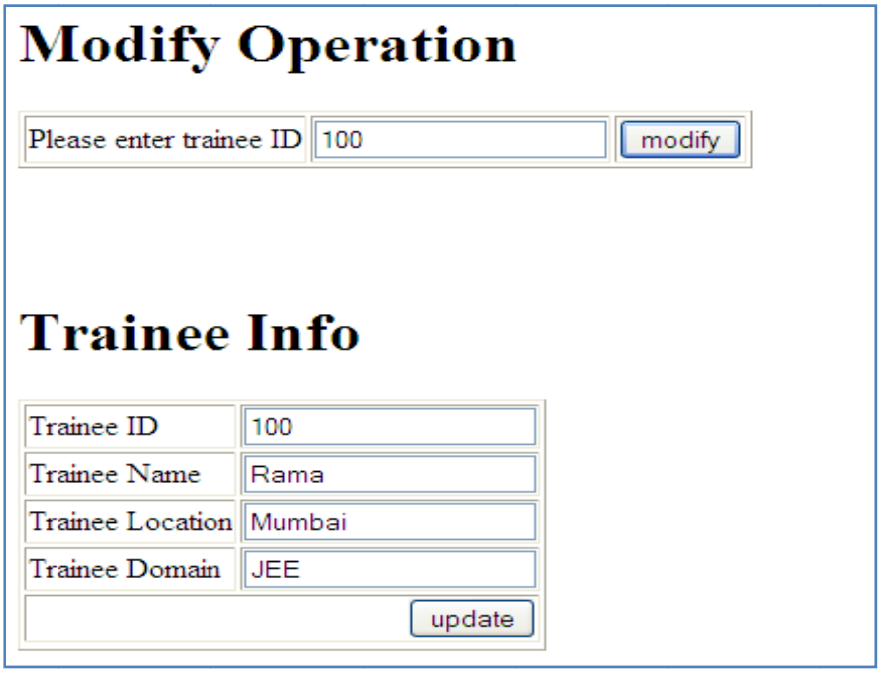
After entering trainee ID in the same page trainee info will be displayed as shown below



To implement modify operation on trainee info refer the following list of screen shots.

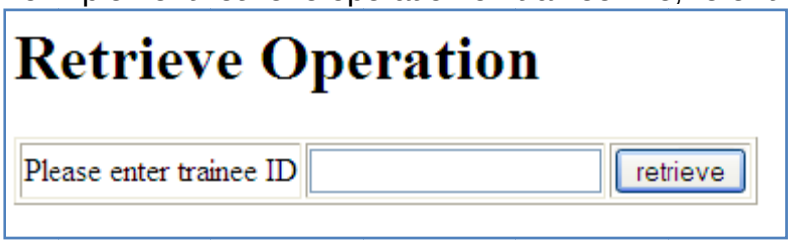


When admin enters trainee id, in the same page trainee info will be retrieved for modification as shown below in the screen shot



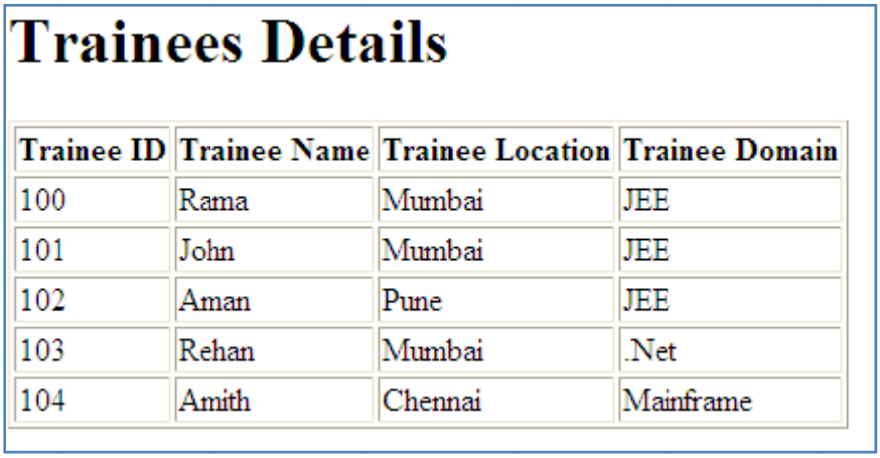
After making appropriate entry admin will click update button to reflect changes in Trainee table.

To implement **retrieve** operation on trainee info, refer the following list of screen shots.



When admin enters trainee id, trainee info will be displayed in the same page as shown below in the screen shot.

Admin selects *'retrieve all'* link to retrieve all trainee info as shown below in the screen shot



Note:

1. Perform appropriate validations on each field for all admin operations including login page
2. If any operation fails admin should be redirected to appropriate error page

Lab 3. Injecting cross-cutting concerns

Goals	• Using AOP
Time	30 minutes

Using spring and AOP

Refer to Trainee Management System created in the previous lab, implement logging using spring AOP. Logging should take care recording all the admin operations with the timestamp.

Ex: If admin executes inserting new trainee record into database, the logging should log that info in a file called adminoperations.log as shown below:

2013-05-28 09:04:39 INFO Admin executed Add Trainee operation.

Appendices

Appendix A: Class Diagrams

