Majrul Ansari
Java Technology Trainer

SPRING FRAMEWORK

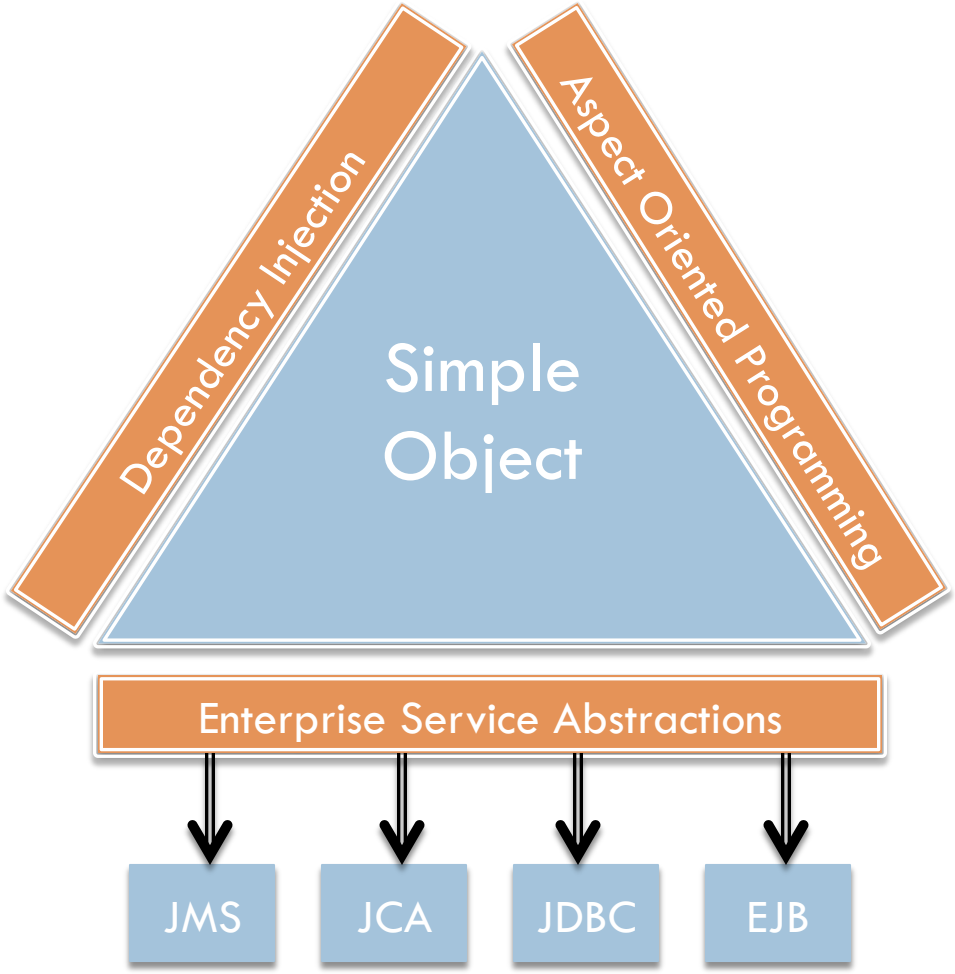
My Introduction

- Working on Java Technology since 1999
- More than 10 years of Corporate Training experience, otherwise a Consultant/Freelancer
- SpringSource **Certified** Spring Professional
- Bean :-) involved with this technology for more than 6 years now

What are we going to learn?

- Core modules of Spring framework
  - IoC/DI, AOP
  - JDBC/Hibernate/JPA Support, Transactions, Messaging
  - Testing support
- New features of Spring 3.x and 4.x
- Spring MVC, Spring Security, Spring JMS

The Spring Triangle



Goal of Spring framework

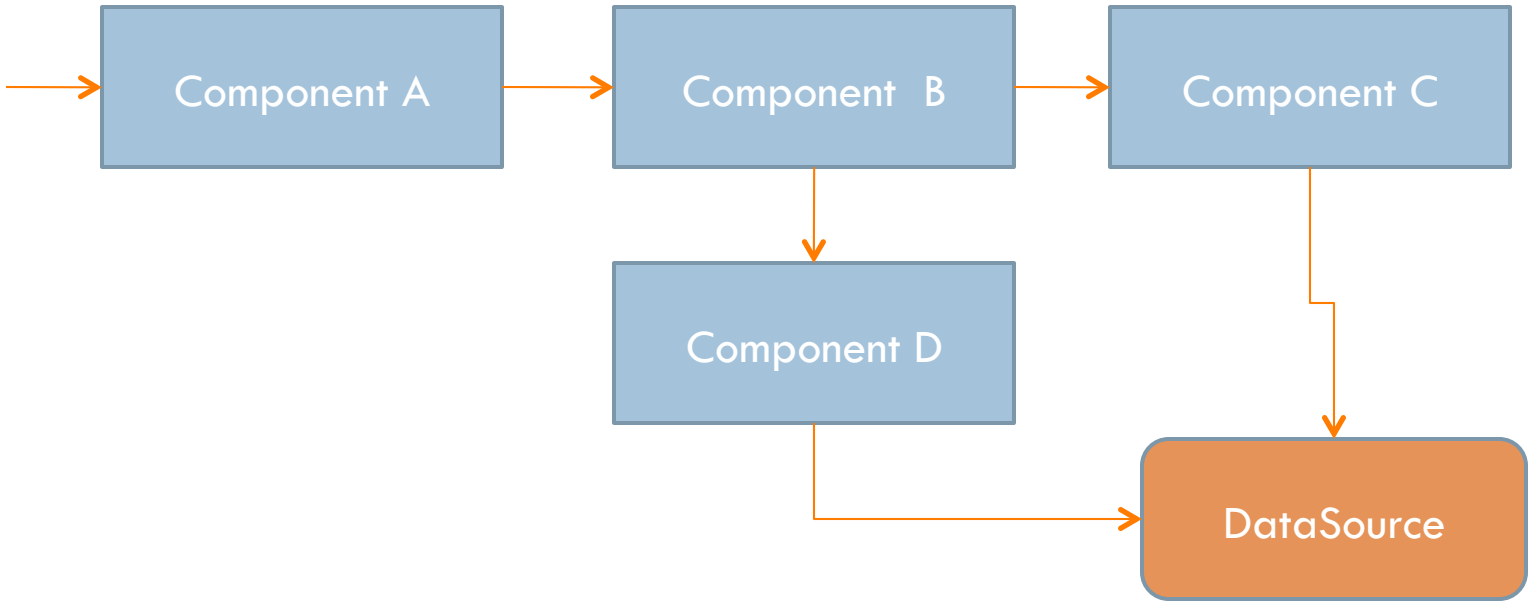
- Provide comprehensive infrastructural support for developing enterprise Java applications
  - Spring deals with the plumbing
  - So you can focus on solving the domain problem

Spring advantage

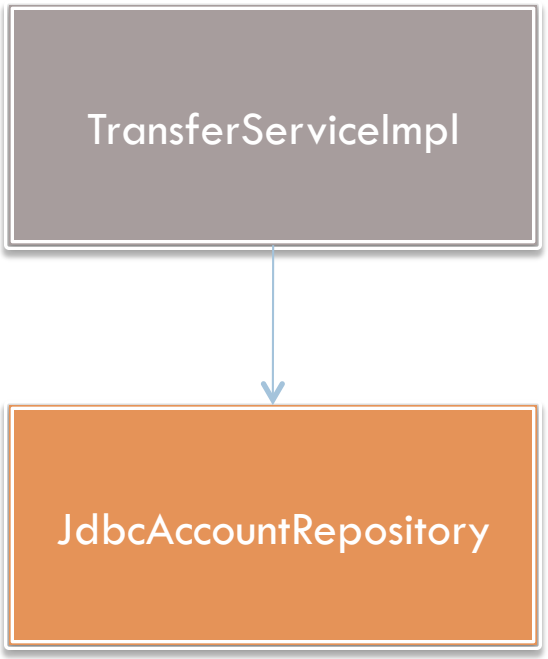
- Make a Java method execute in a database transaction without having to deal with the transaction APIs
- Make a local Java method a remote procedure without having to deal with remote APIs
- Make a local Java method a management operation without having to deal with JMX APIs
- Make a local Java method a message handler without having to deal with JMS APIs

What is Dependency Injection?

- A typical application system consists of several parts working together to carry out a use case

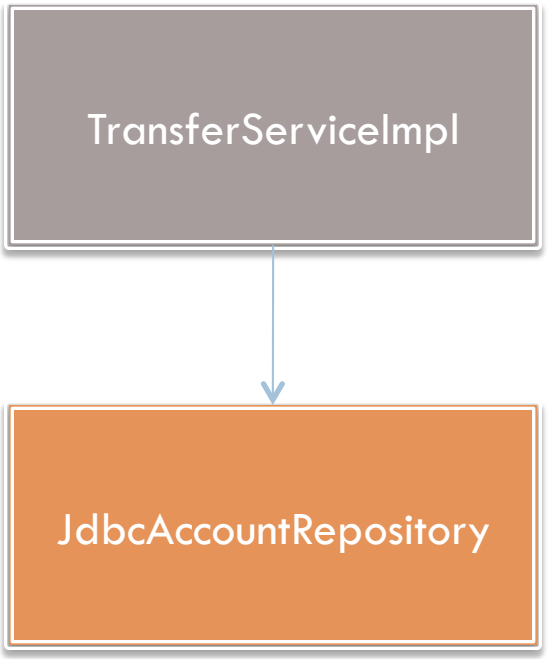


If not using Spring




```
1)  accRepo = new JdbcAccountRepository(…);
2)  tranSer = new TransferServiceImpl(accRepo);
3)  tranSer.tranferMoney("1", "2", 2000.0);
```

What we want

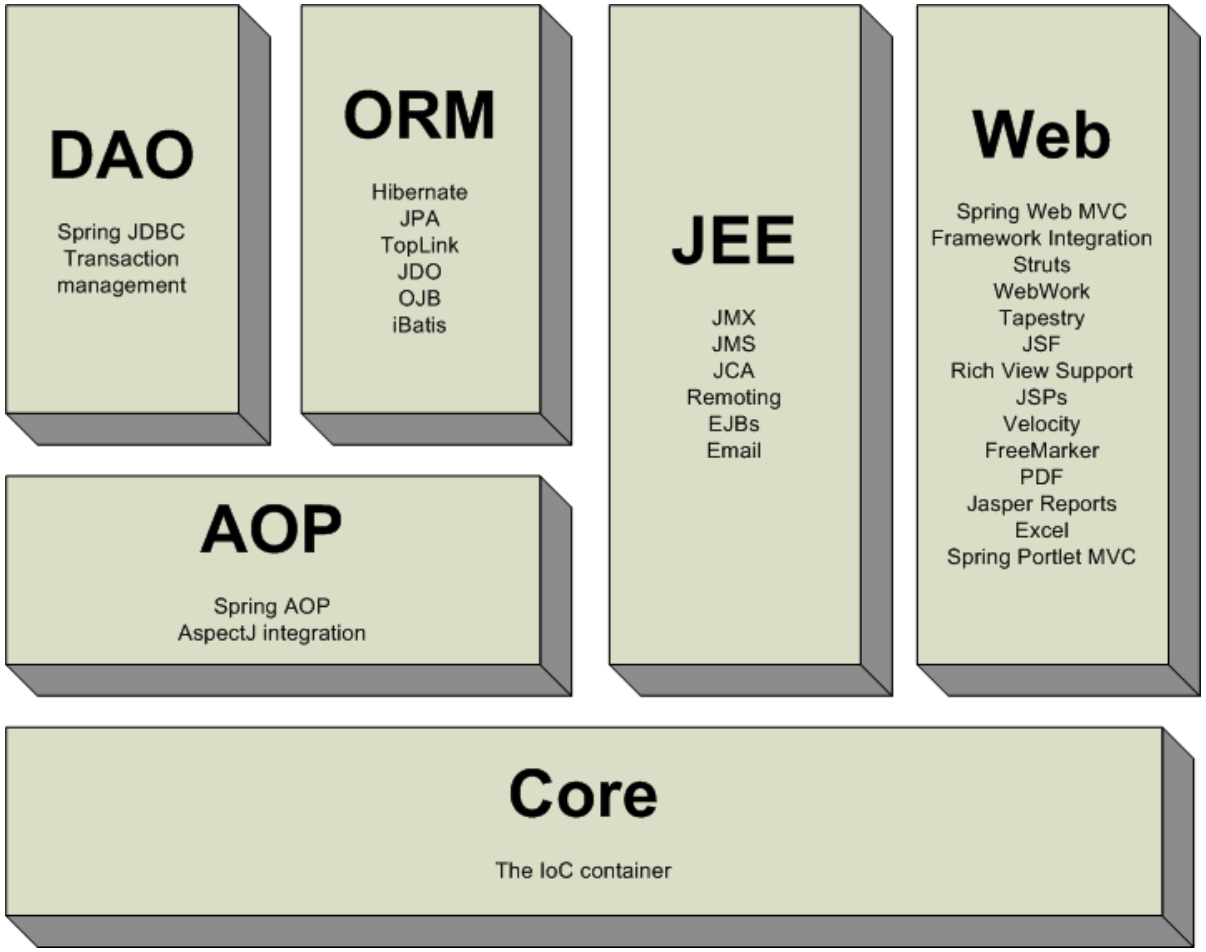


1) tranSer.tranferMoney("1", "2", 2000.0);

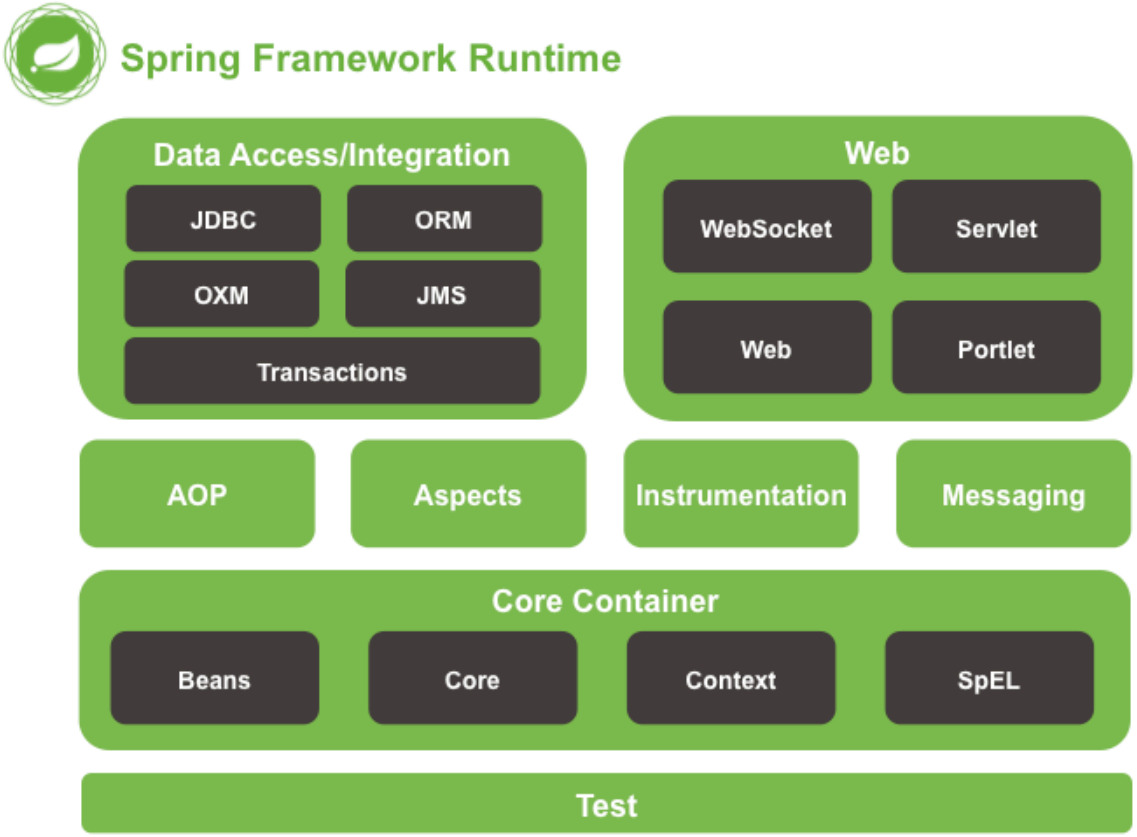
But how?

- To achieve this, we need someone who can manage the components and it's dependencies
- The term *Container* which is commonly used in EE, is also used by Spring and is referred to as an IoC (Inversion of Control) Container
- Spring provides a Container/Factory/Context which manages
  - Component instantiation and initialization
  - Component dependencies
  - Services wrapped around those Components

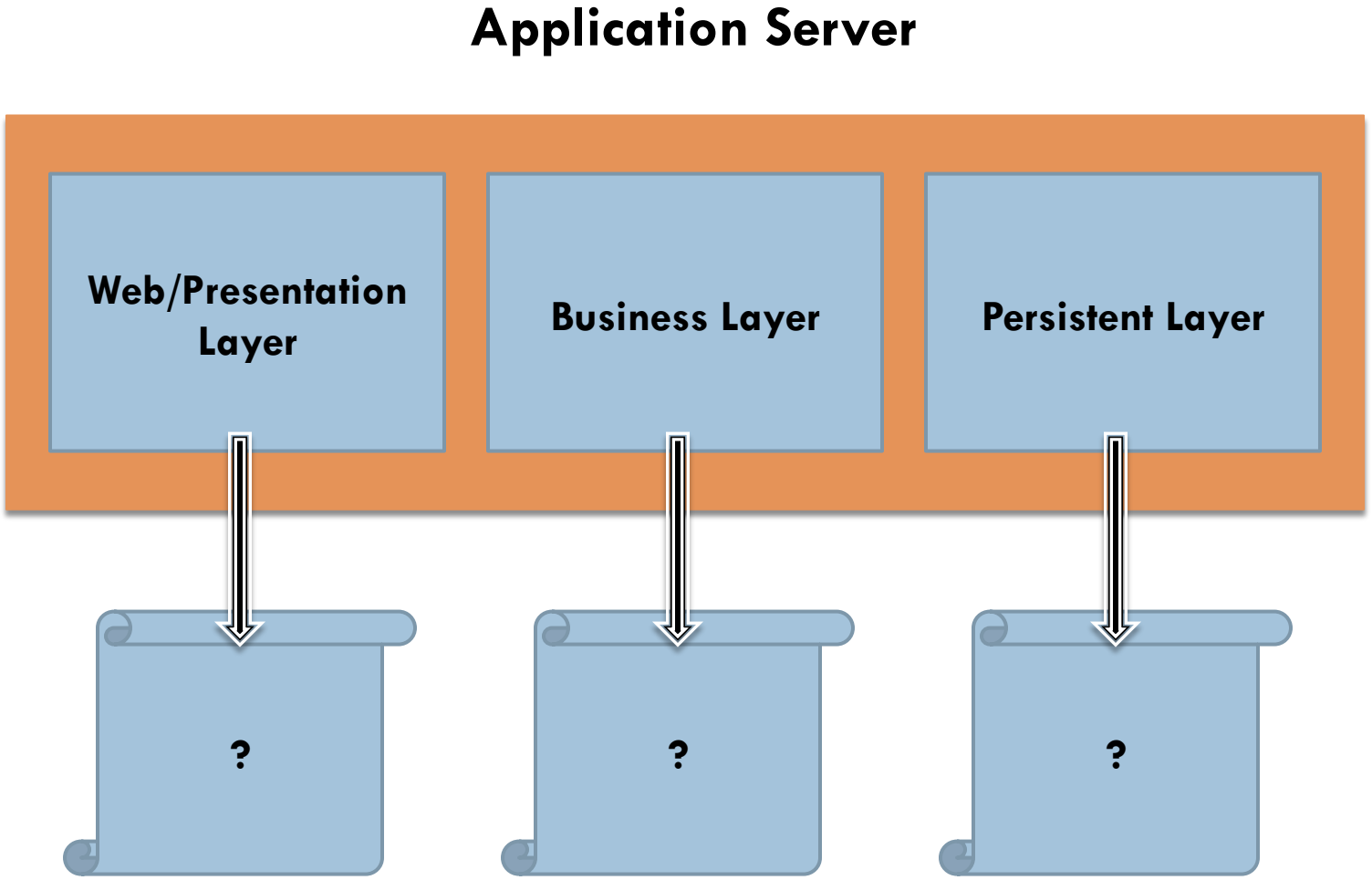
Core modules of Spring Framework



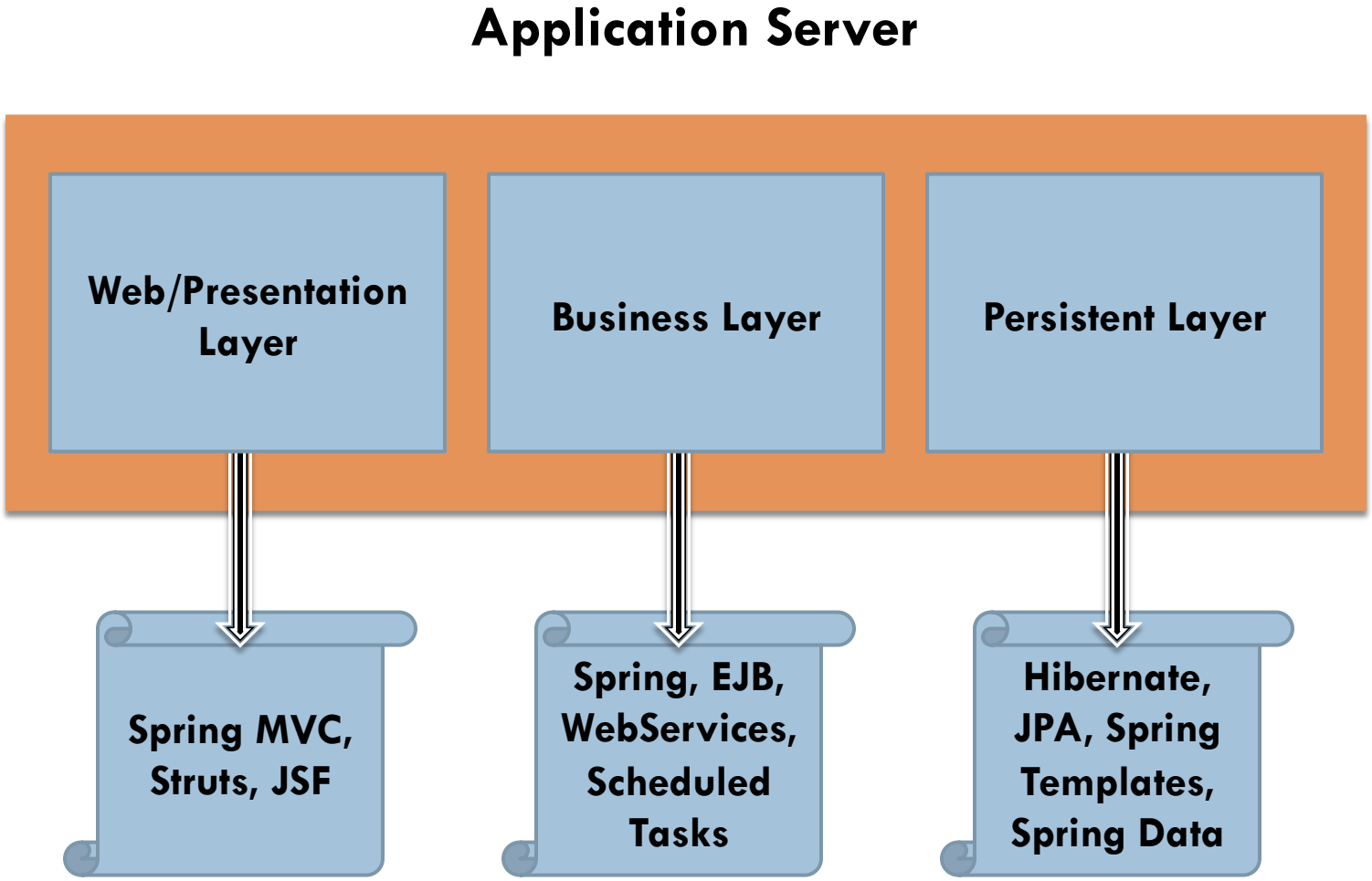
Cont'd…


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Instrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

Typical JEE Applications



Where does Spring fits?



Enter Spring

- A **Lightweight** JEE framework designed to provide louse coupling between components and a **POJO** model recommended for business modeling rather than EJB
- Provides a **Container** which manages business objects and lifecycle
- Dependencies between components are supplied via **XML** configuration. Since Spring 2.x even **annotations** are also supported

The IoC Container

- The IoC container is an implementation of **BeanFactory** interface
- Then we have **ApplicationContext** and **WebApplicationContext** derived from it for adding additional capabilities
- The most commonly used form of IoC container is the **ApplicationContext** implementation

Cont'd…

- When the container is loaded, it gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata can be represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects

What is Dependency Injection but?

- It's a process whereby objects define their dependencies, that is, the other objects they work with. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC)

Different ways of DI

- We will explore different types of DI
  - Setter injection
  - Constructor injection
  - Field injection
    - only possible using annotations approach
  - Method injection
    - Using Spring API

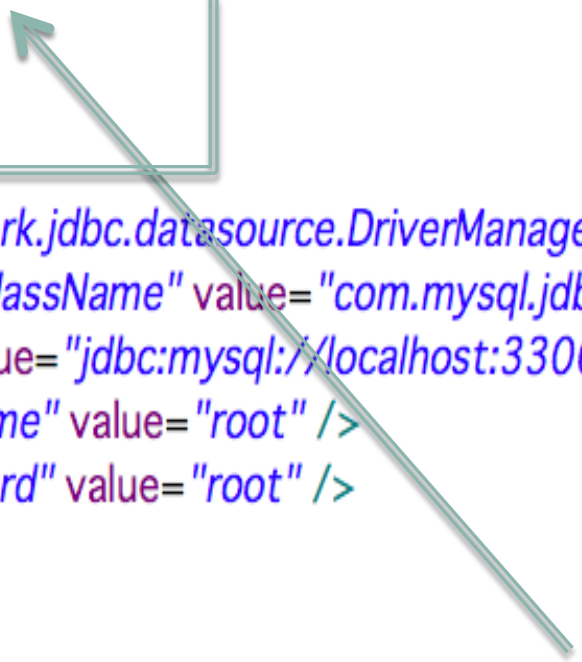
Setter injection


```
public class FlightRepositoryImpl implements FlightRepository {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

We need a DataSource object to be injected

XML Configuration


```
<bean id="flightService" class="xml.FlightRepositoryImpl">
   <property name="dataSource" ref="ds" />
</bean>

<bean id="ds"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
   <property name="url" value="jdbc:mysql://localhost:3306/test" />
   <property name="username" value="root" />
   <property name="password" value="root" />
</bean>
```



That's the setter method of the bean class we are referring to

Annotations style configuration


```
public class FlightRepositoryImpl implements FlightRepository {

    private DataSource dataSource;

    @Resource(name="ds")
    public void setDataSource(DataS
        this.dataSource = dataSource;
    }
```

Here we are injecting a DataSource using annotations. To activate use of annotations for DI, we need to inform the container about the same

```
<context:annotation-config />
```

```
<bean id="flightService" class="xml.FlightRepositoryImpl">
    <property name="dataSource" ref="ds" />
</bean>

<bean id="ds"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/test" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>
```

Cont'd…

- **@Resource** is a JSR standard annotation. Apart from this we can use **@Autowired** annotation provided is by Spring for DI.
- **@Inject** which is part of the **CDI (Context and Dependency Injection)** specification is also supported by default from Spring 3 onwards
- Using the above annotations only enables DI, we still need to add the bean entries in the xml file

Cont'd…

- To avoid using the <bean> tag in the xml file, we need to use **@Component** annotation
- Any bean marked as **@Component** can be scanned by the IoC container on startup and will be loaded dynamically
- To make it more clear as to what type of components are we building, Spring introduced **@Service, @Repository, @Controller** and many other stereotypes to distinguish between the same
- Alternatively **@Named CDI** annotation be used instead of **@Component**
- **@CDI (Context & Dependency Injection)** is a JSR standard for DI in Java. Spring framework supports CDI. Benefit of using CDI will be our code will remain the same if we shift from one DI Container to another in future

@Component type


```
@Repository
public class FlightRepositoryImpl implements FlightRepository {

    private DataSource dataSource;

    @Resource(name="ds")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

Needed in the xml file to tell the container about fully annotated components. To provide an id the bean, it can be done somewhat like this, **@Repository("flightRepo")**

```
<context:component-scan base-package="annotations" />
```

Constructor injection


```
public class FlightRepositoryImpl2 implements FlightRepository {

    private DataSource dataSource;

    public FlightRepositoryImpl2(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

This time the dependencies are bound to a constructor

Informing the container to invoke a parameterized constructor

```
<bean id="flightRepo" class="xml.FlightRepositoryImpl2">
    <constructor-arg ref="ds" />
</bean>
```

Using annotations for Constructor DI


```
@Repository
public class FlightRepositoryImpl2 implements FlightRepository {

    private DataSource dataSource;

    @Autowired
    public FlightRepositoryImpl2(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

Informing the container that we need to inject a DataSource by calling this constructor

A bit more about @Autowired

- Autowiring is a mechanism by which beans dependent on each other can be discovered by the framework while loading them and inject them dynamically based on the autowire type in the XML or annotation reducing duplicate configuration
- Basically there are 2 ways of autowiring dependencies:
  - byName
  - byType

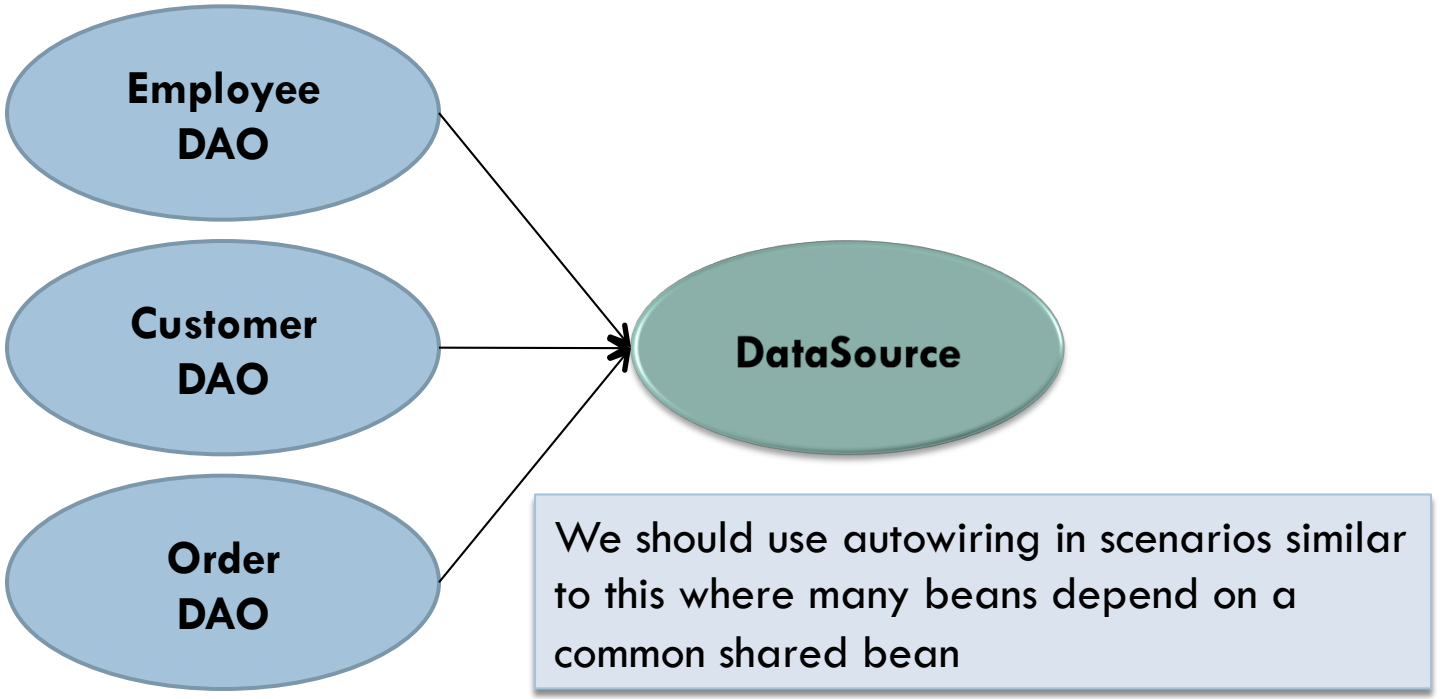
Example


```
@Repository
public class FlightRepositoryImpl3 implements FlightRepository {

    @Autowired @Qualifier("ds")
    private DataSource dataSource;


@Repository
public class FlightRepositoryImpl3 implements FlightRepository {

    private DataSource dataSource;

    @Autowired @Qualifier("ds")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

Possible ways of using autowiring by name explicitly

Autowiring using xml configuration

- Autowiring can be enabled for beans configured in the xml file so that we don't need to use <property /> or <constructor-arg /> tag again and again



Example


```
<bean id="flightRepo" class="xml.FlightRepositoryImpl3" autowire="byType" />

<bean id="ds"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
   <property name="url" value="jdbc:mysql://localhost:3306/test" />
   <property name="username" value="root" />
   <property name="password" value="root" />
</bean>
```

As you can see, we are informing the container that automatically search for bean of type DataSource and inject it in the Repository class

Autowiring byName


```
public class FlightRepositoryImpl3 implements FlightRepository {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/test" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>
```

Revisiting the IoC container

- As we discussed, the IoC container is loaded in the memory as an instance of *BeanFactory*, *ApplicationContext* or a *WebApplicationContext* instance

IoC Container

- **BeanFactory** is the actual container managing all bean instances
- **ApplicationContext** derives from BeanFactory and extra support for:
  - Support for I18N, via MessageSource
  - Access to resources, such as files and URLs
  - Event propagation
  - Loading of multiple contexts
- **WebApplicationContext** is further derived from ApplicationContext. To be used in a web application

Example


```
ApplicationContext container =
    new ClassPathXmlApplicationContext("xml/applicationContext.xml");
FlightRepository flightRepo =
    (FlightRepository) container.getBean("flightRepo");
```

Over here we are loading the IoC container as an ApplicationContext instance

getBean("id") returns an instance of the bean managed by the container

Scope of a bean

- Beans managed by the container can have different scopes:
  - singleton (default)
  - prototype (non singleton)
  - request
  - session
- Apart from these, we can create custom scopes of our own as well

Changing the scope of a bean

We use the scope attribute in the xml to change the scope of a bean

```
<bean id="flightRepo" class="xml.FlightRepositoryImpl" scope="prototype">
```

We use @Scope annotation as an alternative to xml approach

```
@Repository
@Scope(”prototype")
public class FlightRepositoryImpl2 implements FlightRepository {
```

Introduction to IoC and DI

Lab No. 1

Refer to the lab guide provided along with the eclipse project to proceed further

Topics to cover

- Inheritance
- Creating bean instance using static factory method
- Creating bean instance using a custom factory class
- Spring API for DI
- Understanding namespaces and using some of them

Inheritance between beans

- It's obvious that we will use inheritance for reusing common behavior across. From Spring perspective, those beans which share common set of dependencies, will share a common base class
- For example, all repository classes require a DataSource, so instead of declaring the DataSource again and again, we can have a base class containing the DataSource

Example


```
public abstract class BaseRepository {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

```
public class FlightRepositoryImpl extends BaseRepository implements FlightRepository {

    public List<Flight> getAvailableFlights() {
```

```
<bean id="baseRepo" class="xml.BaseRepository" abstract="true">
    <property name="dataSource" ref="ds" />
</bean>

<bean id="flightRepo" class="xml.FlightRepositoryImpl" parent="baseRepo" />
```

That's how we tell the container about the parent/child relationship

Annotation Configuration


```
public abstract class BaseRepositoryAnnotated {

        private DataSource dataSource;

        @Autowired
        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public DataSource getDataSource() {
                return dataSource;
        }

}

@Repository
public class FlightRepositoryImplAnnotated
        extends BaseRepositoryAnnotated implements FlightRepository {

        public List<Flight> getAvailableFlights() {
                Connection conn = null;
                PreparedStatement pst = null;
                ResultSet rs = null;
                try {
                        conn = getDataSource().getConnection();
                        ...
```

A much more natural approach, there is no need to tell who the parent is

Factory method example


```
public class ClientService {
        private static ClientService clientService;

        private ClientService() {
        }

        public static ClientService createInstance() {
                if(clientService == null)
                        clientService = new ClientService();
                return clientService;
        }
}
```

The only addition to the xml is the factory-method attribute so the container knows that this method has to be called to instantiate the bean

```
<bean id="clientService" class="xml.ClientService"
                factory-method="createInstance"/>
```

Custom factory classes

- Spring itself is a readymade factory pattern implementation. That doesn't means we don't need to write one
- In projects we tend to write our own factory classes for many reasons. There are couple of ways to plug our own factory classes in Spring framework:
  - Traditional way
  - Using the *FactoryBean* API

Example


```
public class MyServiceLocator {

    public MyService createMyService() {
        //we assume there is some complex code to initialize
        //MyService bean
        return new MyService();
    }

    public MyService createMyService(DataSource dataSource) {
        //some database specific code to fetch values required for
        // instantiating the bean
        return new MyService();
    }
}
```

In most cases, a factory class will contain multiple methods returning different object everytime or even the same method might return different object based on the parameters passed

The configuration


```
<bean id="serviceLocator" class=”com.package.MyServiceLocator" />

<bean id="myService" factory-bean="serviceLocator"
                              factory-method="createMyService" />
```

As you can see, the container is not informed about the actual bean class, but rather has been provided with the information about a factory bean and the name of the method to call

For each method of the factory class, we need to add those many entries in the xml file. Yes if the same method returns different objects everytime then one entry is sufficient.

Passing parameters to factory methods

- There are 2 ways of passing parameters to factory methods. One is via xml and the other from the the code:
- If the factory method takes a DataSource as a parameter, then this is how the configuration will look like:

```
<bean id="myService" factory-bean="serviceLocator"
                              factory-method="createMyService">
        <constructor-arg ref="ds" />
</bean>
```

Cont'd..

- In most cases, we might want to pass parameters directly from the code, more so dynamically
- To achieve this, we need to call *getBean(id, params)* method. For ex:

```
ApplicationContext container = new
            ClassPathXmlApplicationContext("ex3/ex3-config.xml");
MyService myService =
            (MyService) container.getBean("myService", obj);
```

- "obj" indicates some parameter we need to pass to the factory method. It can be a String or any Java type

FactoryBean interface

- Another way of writing a factory class in Spring is to implement this interface
- If the role of the factory class is precise and clear, for example, we wan't to create a factory class which will return a customized DataSource for our test environment then, this option is a good choice

Example


```
public class MyServiceLocator implements FactoryBean<MyService> {

        @Override
        public MyService getObject() throws Exception {
                return new MyService();
        }
        @Override
        public Class<?> getObjectType() {
                return MyService.class;
        }
        @Override
        public boolean isSingleton() {
                return true;
        }
}
```

The xml configuration


```
<bean id="myService" class="ex4.MyServiceLocator" />
```

If we call getBean("myService"), we will always get the object returned from the factory.

The only problem is there is no way to pass parameters dynamically to the factory from the code. Although we can inject whatever we like as it is configured like any other bean.

Some inbuilt examples of FactoryBean are ProxyFactoryBean, PropertiesFactoryBean, LocalSessionFactoryBean, etc…

Using the Context API for accessing a bean


```
public class CustomerServiceImpl
implements CustomerService, ApplicationContextAware {

        private ApplicationContext ctx;

        @Override
        public void setApplicationContext(ApplicationContext ctx) {
                this.ctx = ctx;
        }

        private BillPaymentService createBillPaymentInstance() {
                return ctx.getBean("billPaymentService", BillPaymentService.class);
        }

        public void payBill(double amt) {
                BillPaymentService billingService = createBillPaymentInstance();
                ...
        }
}
```

Manually calling getBean here

No need to implements *ApplicationContextAware* if we are using annotations

Namespaces in Spring

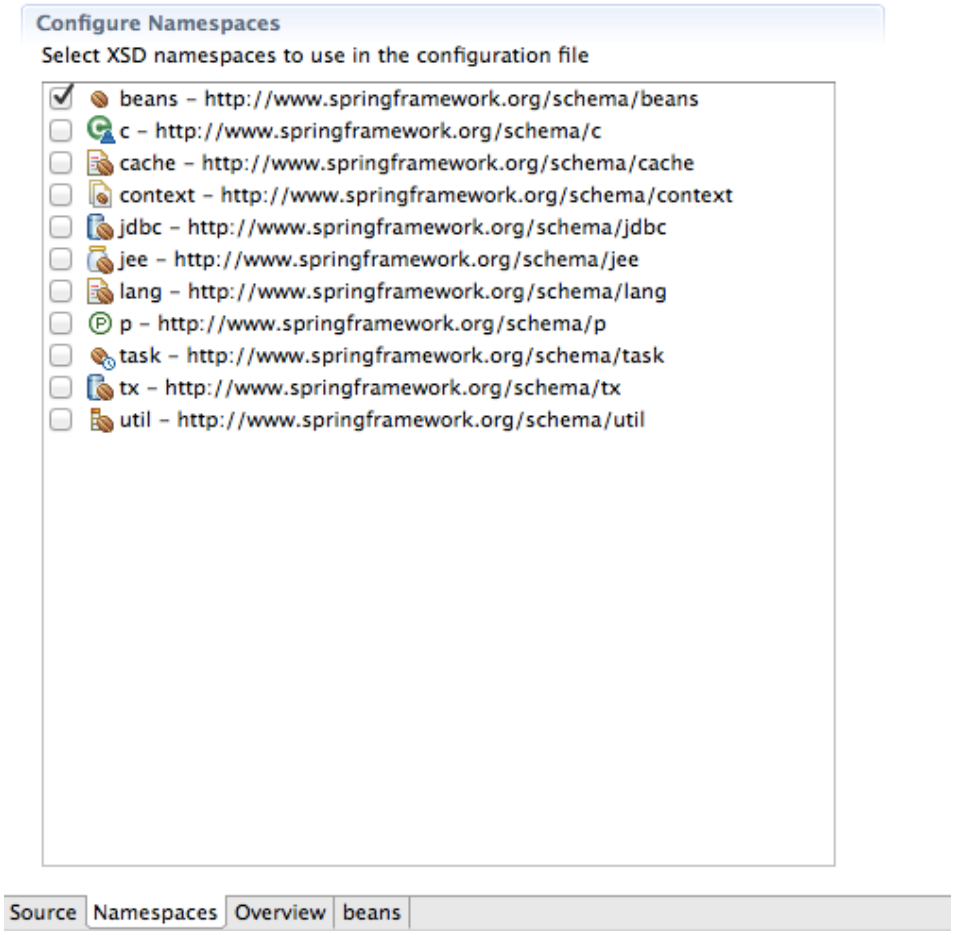
- By now you must be familiar with the xml file which we have been writing every time. Older versions of Spring configuration relied on dtd based validation. From version 2.0, xml schemas (xsd) were introduced making the configuration of different resources in spring much easier
- Also we can create custom schemas of our own and use that in the configuration file of Spring

Default namespace

- We use the beans namespace as the default one. So most of the tags we have seen so far in the xml was defined in this namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

Some of the namespaces



Why so many namespaces?

- Since in Spring any resource is configured as a bean in the xml file, whether it is a DataSource or JNDI resource or user defined beans. Sometimes this leads to confusions since the xml file is less verbose
- With namespaces it becomes easy to identify what type of bean have we configured in the xml file
- We will see some examples to understand the difference and discuss about the best option

Example 1: Reading Properties file


```
public class SomeServiceClass {

    private Properties adminEmails;

    public void setAdminEmails(Properties adminEmails) {
        this.adminEmails = adminEmails;
    }
```

According to code above, we need to inject a Properties object

Old style configuration


```
<bean id="serviceClass" class="xml.SomeServiceClass">
    <property name="adminEmails" ref="adminProperties" />
</bean>

<bean id="adminProperties"
    class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="location" value="classpath:xml/admin.properties" />
</bean>
```

PropertiesFactoryBean will create a Properties object and we are injecting the same in the service class

Schema style configuration


```
<bean id="serviceClass" class="xml.SomeServiceClass">
   <property name="adminEmails" ref="adminProperties2" />
</bean>

<util:properties id="adminProperties2" location="classpath:xml/admin.properties" />
```

In this example, we are using the util namespace. As you can see the name of the tag hides the actual spring class doing the job and makes the configuration a bit more easy to understand. So whenever you see any such tag, the intention is clear, hide the actual bean class doing the job!

Example 2: JNDI lookup

- As in most cases, resources like DataSource, EJB, Queue/Topic, etc… are configured on an Application Server and are bound to a JNDI registry. We need to perform lookups to obtain the necessary resource and proceed accordingly. In Spring there are readymade classes available for the same. Let's see an example on the same!

Old style configuration


```
public abstract class BaseRepository {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

```
<bean id="baseRepo" class="xml.BaseRepository" abstract="true">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/trainingdb" />
</bean>
```

Schema style configuration


```
<bean id="baseRepo" class="xml.BaseRepository" abstract="true">
    <property name="dataSource" ref="dataSource2" />
</bean>

<jee:jndi-lookup id="dataSource2" jndi-name="jdbc/trainingdb" />
```

This time we are using the jee namespace. And once again as you can see, the jndi-lookup tag hides the actual java class from the configuration

Example 3: Using p namespace


```
<bean id="myDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/test" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>
```

The p namespace simplifies setting multiple properties conveniently

```
<bean id="myDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/mydb"
    p:username="root"
    p:password="masterkaoli"/>
```

Cont'd…


```
<bean id="baseRepo" class="xml.BaseRepository" abstract="true">
    <property name="dataSource" ref="myDataSource" />
</bean>
```

```
<bean id="baseRepo" class="xml.BaseRepository" abstract="true"
    p:dataSource-ref="myDataSource" />
```

Example 4: Again Properties file

- We will now discuss about some more commonly used helper class provided by Spring dealing with properties file and then we will see the schema style configuration for the same

PropertyPlaceholderConfigurer

- With the help of this utility class, one can avoid hardcoding the bean properties in the xml and use ant-style *${propertyname}* styntax in the xml file
- As of Spring 3.1, **PropertySourcesPlaceholderConfigurer** should be used when not using namespaces

Old style configuration


```
<bean id="myDataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource"
   p:driverClassName="${driver}"
   p:url="${url}"
   p:username="${user}"
   p:password="${pass}"/>
```

```
#db.properties
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/test
user=root
pass=root
```

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
   <property name="location" value="classpath:xml/db.properties" />
</bean>
```

PropertiesPlaceHolder will replace all the placeholders, ${} with the value from properties file. Even for reading system properties, we can use this helper class, but we will see the new approach in 3.0

Schema style configuration


```
<bean id="myDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="${driver}"
    p:url="${url}"
    p:username="${user}"
    p:password="${pass}"/>
```

```
#db.properties
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/test
user=root
pass=root
```

```
<context:property-placeholder location="classpath:xml/db.properties"/>
```

Here we are using the property-placeholder tag from the context namespace to achieve the same as we saw in the previous slide

PropertyOverrideConfigurer

- This one is another smart little helper class to play with Properties file. With the help of this class, you can set the bean properties in a properties file without even using placeholders in the xml file

Old style configuration


```
<bean id="myDataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource" />
```

```
#db2.properties
myDataSource.driverClassName=com.mysql.jdbc.Driver
myDataSource.url=jdbc:mysql://localhost:3306/test
myDataSource.username=root
myDataSource.password=root
```

```
<bean class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
   <property name="location" value="classpath:xml/db2.properties" />
</bean>
```

PropertyOverrideConfigurer will search for the bean properties in the properties file and set the matching ones

Schema style configuration


```
<bean id="myDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource" />
```

```
#db2.properties
myDataSource.driverClassName=com.mysql.jdbc.Driver
myDataSource.url=jdbc:mysql://localhost:3306/test
myDataSource.username=root
myDataSource.password=root
```

```
<context:property-override location="classpath:xml/db.properties"/>
```

With the help of all these examples, I hope you have a good understanding as to why Spring has introduced so many different namespaces. Explore as many as you can!

Spring IoC

Lab No. 2

Refer to the lab guide provided along with the eclipse project to proceed further

Topics to be covered

- Some new features of Spring 3.x & 4.x
- A lot has changed in recent versions of Spring
- New approaches have been introduced to simplify development as much as possible
- As of now, Spring 4.1 is the latest version

New features of Spring 3.x

- Java 5 based
  - Support for Generics and other features of Java
- Spring Expression Language (EL)
- Java based container metadata (Annotations)
- Improved Spring MVC and REST support
- Support for Java EE 6
  - Which means supports for lots of standard JSR annotations like @Inject, @Async and others
- Caching, environment abstraction, bean profiles and others introduced in Spring 3.1

New features of Spring 4.x

- Java 8 ready
- Fully compliant with latest Java EE 7 enhancements like JMS 2.0, JPA 2.1, Bean Validation 1.1 and others
- Core container improvements in autowiring, lazy on injection points and others
- Improvements over existing features in Spring 4.1
  - JMS
  - Caching

Spring 3.0 EL (SpEL)

- Spring 3.0 introduces support for expression language which is similar to the Unified EL support in JSP. The syntax of EL is **# { *some expression* }**

Example


```
public class ErrorHandler {

    private String defaultLocale;

    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public void handleError() {
        //some error handling here which is locale specific
        System.out.println(defaultLocale);
    }
}
```

```
<bean id="errorHandler" class="ex1.ErrorHandler">
        <property name="defaultLocale"
                value="#{ systemProperties['user.country'] }" />
</bean>
```

Over here we are reading a system property and setting it in the bean using the EL notation introduced in 3.0

Annotation style configuration


```
public class ErrorHandler {

    @Value("# { systemProperties['user.region'] }")
    private String defaultLocale;

    public void handleError() {
        //some error handling here which is locale specific
        System.out.println(defaultLocale);
    }
}
```

Correction: it's user.country

@Value annotation can be used at field level also so need of a setter method for injecting the value. @Value can be used at setter and constructor level also

Elvis & Ternary Operator in SpEL

- With the help of elvis & ternary operator, we can set a default value when reading any system property using SpEL

```
@Value("#{systemProperties[pop3.port] ?: 25}")
```

This will inject a system property **pop3.port** if it is defined or 25 if not

```
@Value("#{itemBean.qtyOnHand < 100 ? true : false}")
private boolean warning;
```

Example using ternary operator in SpEL

Reading other bean properties


```
<bean id=”bean1" class=”com.package.ExampleBean1”>
        <property name="randomNumber”
                  value="#{ T(java.lang.Math).random() * 100.0 }" />
        <property name="someValue" value="#{ bean2.value }" />
</bean>

<bean id=”bean2" class=”com.package.ExampleBean2 ">
        <property name="value" value="100" />
</bean>
```

Accessing the properties of other beans was something not so directly possible in Spring before

Collection Selection

- Selection is a powerful expression language feature that allows you to transform some source collection into another by selecting from its entries. Selection uses the syntax ?[selectionExpression]. This will filter the collection and return a new collection containing a subset of the original elements

```
ExpressionParser parser = new SpelExpressionParser();
List<Product> products =
        (List<Product>) parser.parseExpression(
        "products.?[name.startsWith('Nokia') and price >= 9500]”)
        .getValue(objOfClassContainingProductsList);
System.out.println(products);
```

Collection Projection

- Projection allows a collection to drive the evaluation of a sub-expression and the result is a new collection
- The syntax for projection is ![projectionExpression]

```
List<String> productNames = (List<String>)
            parser.parseExpression("products.![name]")
                        .getValue(objOfClassContainingProductsList);
System.out.println(productNames);
```

Bean Profiles

- This is one of the new features of Spring 3.1. In every application the resources made available during development, testing and production will vary. For example, during development we might be using some database, for testing we might use a different one and then for production it will be different
- Traditionally people used properties file and separate xml config sets
- Spring now supports profiles which allows us to achieve this abstraction so very well

An example


```
<bean id="flightRepo" class="ex1.FlightRepositoryImpl">
            <property name="dataSource" ref="ds" />
</bean>

<beans profile="dev">
            <bean id="ds"
                        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                        <property name="driverClassName" value="org.hsqldb.jdbc.JDBCDriver" />
                        <property name="url"
                                    value="jdbc:hsqldb:file:../Lab00-Database/trainingdb;shutdown=true"/>
                        <property name="username" value="sa" />
                        <property name="password" value="" />
            </bean>
</beans>

<beans profile="prod">
            <bean id="ds"
                        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                        <property name="driverClassName" value=”com.mysql.jdbc.Driver" />
                        <property name="url"
                                    value=”jdbc:mysql://localhost:3306/test />
                        <property name="username" value=”root" />
                        <property name="password" value="" />
            </bean>
</beans>
```

We need to set a property by the name *spring.profiles.active* either as a VM argument using –D or can be set as a context param in web.xml

In case if we don't specify the profile name, then beans from the "**default**" profile will be loaded by default

Caching

- Added in 3.1, Spring support for caching introduces a transparent layer into the application with minimal impact on the code
- From Spring 4.1 onwards, Spring also supports JSR-107 standard for caching, JCache
- For caching we need a separate provider and the ones supported are:
  - EhCache
  - Gemfire
  - Gauva
  - JCache implementations

Example


```
public class FlightRepositoryImplCacheable {

        private DataSource dataSource;

        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        @Cacheable("allflights")
        public List<Flight> getAvailableFlights() {
                System.out.println("getAvailableFlights called..");
                Connection conn = null;
                PreparedStatement pst = null;
                ResultSet rs = null;


        @CacheEvict(value = "allflights", allEntries = true)
        public Flight addNewFlight(Flight flight) {
                Connection conn = null;
                PreparedStatement pst = null;
```

Cont'd…

- Another interesting feature of Spring caching is support for expressions as you can see below:

```
@Cacheable(value = "carrierflights", condition = "#carrier.startsWith('JET')")
public List<Flight> getAvailableFlights(String carrier) {
          System.out.println("getAvailableFlights called..Accessing the DB");
          Connection conn = null;
          PreparedStatement pst = null;
          ResultSet rs = null;
          conn = dataSource.getConnection();
          String sql = "select * from flights_test where carrier like ?";
          pst = conn.prepareStatement(sql);
          …
}
```

Caching Configuration


```
<cache:annotation-driven />

<bean id="cacheManager"
        class="org.springframework.cache.ehcache.EhCacheCacheManager">
        <property name="cacheManager" ref="ehcache" />
</bean>

<bean id="ehcache”
        class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean”>
        <property name="configLocation" value="classpath:ex1/ehcache.xml" />

</bean>
```

Shifting from EhCache to Gemfire or Guava is a matter of just changing the above configuration with no change at all in the code

Java based container configuration

- Spring 3.0 provides XMLess approach for the first time. New set of annotations have been introduced to eliminate xml completely
- With the introduction of @Component family of annotations in 2.5, we were able to remove bean definitions from the xml and using @Resource and @Autowired, even the dependencies could be managed from the code itself. But we still needed an XML file for registering non-annotated beans
- In 2.0, annotation support was introduced primarily for Transaction management and AOP
- Prior to 2.0, everthing had to be in XML files

@Configuration


```
@Configuration
@ComponentScan("ex2")
@Import(DBConfig.class)
public class AppConfig {

        @Bean
        public FlightRepository flightRepo() {
                FlightRepositoryImpl flightRepository = new FlightRepositoryImpl();
                flightRepository.setDataSource(dataSource());
                return flightRepository;
        }

       @Bean
       public DataSource dataSource() {
                DriverManagerDataSource dataSource = new DriverManagerDataSource();
                dataSource.setDriverClassName(env.getProperty("driver"));
                dataSource.setUrl(env.getProperty("url"));
                dataSource.setUsername(env.getProperty("username"));
                dataSource.setPassword(env.getProperty("password"));
                return dataSource;
       }
```

```
<beans …>
  <import resource="db-config.xml" />

  <bean id="flightRepo" class="xml.FlightRepositoryImpl”>
    ...
  </bean>
```

So what we were doing in xml files for all these years, now the same configuration will be provided through Java code

Loading the container


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

This time we are launching the Spring IoC container with the help of a Java based configuration class. Alternatively to avoid changing the bootstrap process, you can still have an xml file containing
<context:component-scan base-package=”com.package.beans" /> ,
so that the container will automatically load the Java based configuration class like any other bean but with those additional configurational capabilities

Best of both the worlds!

Spring 4.x core improvements

- Spring 4 introduces new features to improve the core IoC container. Some of these new features are:
  - Condition API for conditionally loading a bean
  - Better support for ordering when injecting multiple beans of the same type
  - Improved support for generics when injecting dependencies
  - Lazy support for injecting dependencies (I like this feature)

Condition API

- With the help of this feature one can now define conditions based on which beans will be loaded into the context. Prior to Spring 4, the only way to achieve this was through EL or Bean Profiles. When compared to EL and Bean Profiles, the difference is that it's far more dynamic because of the presence of proper Java code to decide the condition
- Finally, which option to use will depend upon the project specific needs. When compared, EL and Bean Profiles can simply be modified by changing the configuration, but changing the condition written using this new API will require proper Java code modification

Example


```
@Configuration
@ComponentScan("example1")
public class AppConfig {

        @Bean
        @Conditional(Condition1.class)
        public PropertySourcesPlaceholderConfigurer placeholderConfigurer1() {
                PropertySourcesPlaceholderConfigurer props =
                                        newPropertySourcesPlaceholderConfigurer();
                props.setLocation(
                                newClassPathResource("example1/messages1.properties"));
                return props;
        }

        @Bean
        @Conditional(Condition2.class)
        public PropertySourcesPlaceholderConfigurer placeholderConfigurer2() {
                PropertySourcesPlaceholderConfigurer props =
                                        newPropertySourcesPlaceholderConfigurer();
                props.setLocation(
                                newClassPathResource("example1/messages2.properties"));
                return props;
        }
}
```

Cont'd…


```
public class Condition1 implements Condition {

        @Override
        public boolean matches(ConditionContext condition,
                               AnnotatedTypeMetadata metadata) {

                return condition.getEnvironment()
                        .getProperty("user.country").equals("US");
        }

}
public class Condition2 implements Condition {

        @Override
        public boolean matches(ConditionContext condition,
                               AnnotatedTypeMetadata metadata) {

                return condition.getEnvironment()
                        .getProperty("user.country").equals(”GB");
        }

}
```

Ordering of injected dependencies


```
@Component
@Order(1)
public class ProductDao implements Dao {
        …
}
@Component
@Order(2)
public class EmployeeDao implements Dao {
        …
}
@Component
public class SomeUtility {

        @Autowired
        private List<Dao> listOfDaos;

        @PostConstruct
        public void init() {
                System.out.println(listOfDaos);
                for(Dao dao : listOfDaos) {
                        //some initializing logic for the daos
                }
        }

}
```

Support for generics in DI


```
public interface Dao<T> {

        public void add(T t);
        public void remove(T t);
}

@Repository
public class EmployeeDao implements Dao<Employee> {

        …

}

@Repository
public class ProductDao implements Dao<Product> {

        …

}
```

Cont'd…


```
@Service
public class ProductInventoryService {

        @Autowired
        public Dao<Product> productDao;

        public void addProduct(Product product) {
                productDao.add(product);
        }
        …
}
```

From Spring 4 onwards, Spring support for Generic based dependencies has improved. So that means in the above example, Spring will search for an Object of type *Dao<Product>* rather than searching an Object of type *Dao* when compared to the old version

Lazily injecting a dependency


```
@Component
public class ClassA {

        @Autowired
        @Lazy
        private ClassB classB;

        …
}
```

New features of Spring 3.x and 4.x

Lab No. 3

Refer to the lab guide provided along with the eclipse project to proceed further

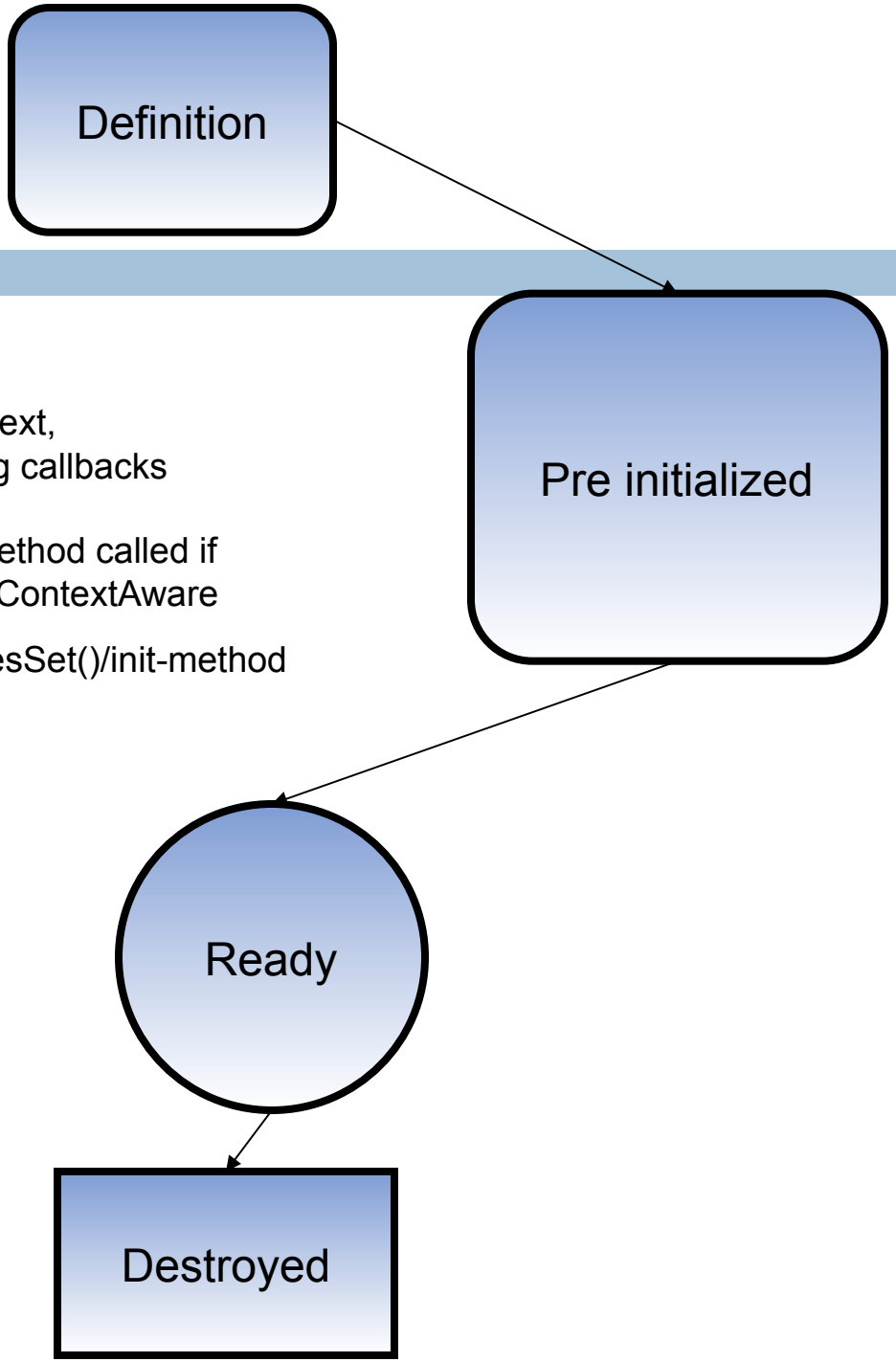
Topics to be covered

- Lifecycle of beans managed by the container
- Extensions for managing container's lifecycle
- Managing events in a Spring environment

Bean life cycle

- As like any other component, beans managed by the IoC container have a life cycle associated with it. The life cycle methods can be implemented optionally and there are multiple possible ways of managing the same
  - Callback API implementation
  - XML configuration
  - Annotations

1. Default Constructor called



2. All the depencies are injected

3. References to ApplicationContext, MessageSource and other Spring callbacks executed
for ex: setApplicationContext() method called if the class implements ApplicationContextAware

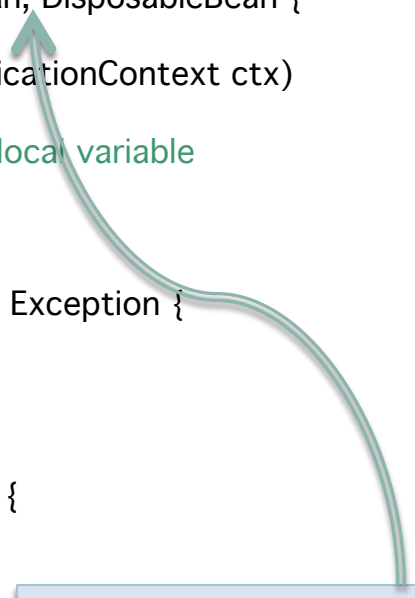
4. @PostConstruct/afterPropertiesSet()/init-method called

5. Application running, beans ready to work

6. @PreDestroy/destroy()/ destroy-method called

API Example


```
public class LifecycleBean implements
        ApplicationContextAware, InitializingBean, DisposableBean {
        @Override
        public void setApplicationContext(ApplicationContext ctx)
                throws BeansException {
           //storing the context reference in a local variable
        }

        @Override
        public void afterPropertiesSet() throws Exception {
           //some custom initialization
        }

        @Override
        public void destroy() throws Exception {
           //some custom cleanup
        }
}
```

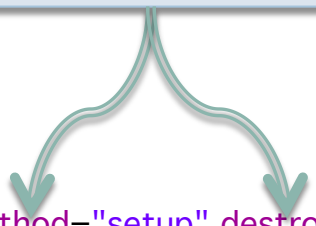


Here we are implementing the lifecycle interfaces

Using some of the xml options


```
public class LifecycleBean2 implements ApplicationContextAware {

    @Override
    public void setApplicationContext(ApplicationContext ctx)
                        throws BeansException {
        //storing the context reference in a local variable
    }

    public void setup() {
        //some custom initialization
    }

    public void cleanup() {
        //some custom cleanup
    }
}
```

Using init-method and destroy-method attributes provided in the xml file



```
<bean id="mybean" class="xml.LifecycleBean2" init-method="setup" destroy-method="cleanup" />
```

Annotations based life cycle callback


```
public class LifecycleBean3 {

	@Resource
	private ApplicationContext ctx;

	@PostConstruct
	public void setup() {
		//some custom initialization
	}

	@PreDestroy
	public void cleanup() {
		//some custom cleanup
	}
}
```

Pure annotations based code. Much simpler if you agree. As you can see Spring supports JSR standard annotations wherever possible

Context level lifecycle callbacks

- After discussing the primary lifecycle of a bean in the container, let's now discuss about the lifecycle callbacks that can be implemented at a more broader level
- *BeanFactoryPostProcessor* and *BeanPostProcessor* implementations are the most commonly used extension points in Spring

BeanPostProcessor interface

- The BeanPostProcessor interface defines *callback methods* that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and otherwise initializing a bean, you can plug in one or more BeanPostProcessor implementations
- You can control the order in which these BeanPostProcessor interfaces execute by setting the order property. You can set this property only if the BeanPostProcessor implements the Ordered interface
  - @Order annotation can also be used instead of the above approach also

Cont'd…

- Classes that implement the BeanPostProcessor interface are *special*, and so they are treated differently by the container. All BeanPostProcessors *and their directly referenced beans* are instantiated on startup, as part of the special startup phase of the ApplicationContext

Example


```
public class BeanInitializationLogger implements BeanPostProcessor {

	@Override
	public Object postProcessAfterInitialization(Object bean, String beanName)
	throws BeansException {
		System.out.println("after Initializing Bean '"+beanName+"'");
		return bean;
	}

	@Override
	public Object postProcessBeforeInitialization(Object bean, String beanName)
	throws BeansException {
		System.out.println("before initializing Bean '"+beanName+"'");
		return bean;
	}
}
```

A BeanPostProcessor implementation will be given a chance to perform custom processing before and after any bean is initialized

Another example

- BeanPostProcessor is used by the framework heavily. One example is *RequiredAnnotationBeanPostProcessor*
- In Spring we have an annotation called as @Required to make it mandatory that the dependency has to be injected
- Just by using this annotation in the code will not work, since someone has to check whether the required dependencies has been set or not and report an error accordingly. That's the role of this library class

Example


```
public class BankService {

        private CustomerService customerService;
        private BillPaymentService billPaymentService;

        @Required
        public void setCustomerService(CustomerService customerService) {
                this.customerService = customerService;
        }

        @Required
        public void setBillPaymentService(BillPaymentService billPaymentService) {
                this.billPaymentService = billPaymentService;
        }

}
```

We need to use RequiredAnnotationPostProcessor which checks whether @Required dependencies have been injected or not

The configuration


```
<bean id="custService" class="xml.CustomerServiceImpl" />

<bean id="billingService" class="xml.BillPaymentServiceImpl" />

<bean id="bankService" class="xml.BankServiceImpl">
        <property name="customerService" ref="custService" />
        <property name="billPaymentService" ref="billingService" />
</bean>

<bean class=”o.s.beans.factory.annotation.RequiredAnnotationBeanPostProcessor" />
```

If we don’t set both the properties of BankService class, we will get an error

BeanFactoryPostProcessor

- The semantics of this interface are similar to the *BeanPostProcessor*, with one major difference: *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*
- Please remember, If you want to change the actual bean *instances* (the objects that are created from the configuration metadata), then use *BeanPostProcessor*

Example


```
public class AllBeansLister implements BeanFactoryPostProcessor {

    public void postProcessBeanFactory(ConfigurableListableBeanFactory factory)
    throws BeansException {
            System.out.println("the factory contains the following beans:");
            String[] beanNames = factory.getBeanDefinitionNames();
            for(int i=0;i<beanNames.length;i++)
                    System.out.println(beanNames[i]);

            //Finding out how many repositories we have
            int count = factory.getBeansOfType(BaseRepository.class).size();

            //Registering a new bean dynamically
            factory.registerSingleton("clientService", ClientService.createInstance());
    }
}
```

BeanFactoryPostProcessors are the very first set of components loaded by the IoC container. None of the beans have yet got instantiated so that's why it's possible to change the bean metadata

Example

- Again, one of the commonly used *BeanFactoryPostProcessor* is PropertyPlaceHolderConfigurer class. We have already seen the usage of this class before. This class replaces the configuration of any bean containing ${} with the actual value from a properties class, so by the time the bean is instantiated, the correct values are already known to the container

Revisiting


```
<bean id="myDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="${driver}"
    p:url="${url}"
    p:username="${user}"
    p:password="${pass}"/>
```

```
#db.properties
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/test
user=root
pass=root
```

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:xml/db.properties" />
</bean>
```

Event model

- Spring framework provides and event based IoC container
- Another example of an event based container is a Web/Servlet Container. We write listeners over there also to deal with different server side events
- In Spring core, following are in the inbuilt events which can occur:
  - ContextRefreshedEvent, ContextStartedEvent, ContextStoppedEvent, ContextClosedEvent and RequestHandledEvent(MVC)

Example on handling an inbuilt event


```
public class ContextClosedListener
                    implements ApplicationListener<ContextClosedEvent> {

    @Override
    public void onApplicationEvent(ContextClosedEvent evt) {
        //some cleanup code here
        System.out.println("Context Closed Event handled at : “
                                                + evt.getTimestamp());
    }
}
```

In this case we are handling the ContextClosedEvent. Whenever the container wil raise this event, our listener will get chance to perform some cleanup

Raising a custom event

- The steps for raising a custom event are simple:
  - Write a custom event class
  - Write the business logic to raise an event
  - Write a listener for dealing with the event

Writing a custom event class


```
public class BlackListEvent extends ApplicationEvent {

        private String address;

        public BlackListEvent(String address, Object source) {
                super(source);
                this.address = address;
        }
}
```

An event class is just an extension of ApplicationEvent abstract class. Whenever an event is raised, the listener can track the source of the event from the event object.

Business logic to raise an event


```
public class EmailBean implements ApplicationContextAware {

    private List<String> blackList;

    private ApplicationContext ctx;

    public void setBlackList(List<String> blackList) {
        this.blackList = blackList;
    }

    public void setApplicationContext(ApplicationContext ctx) {
        this.ctx = ctx;
    }

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
            BlackListEvent event = new BlackListEvent(address, this);
            ctx.publishEvent(event);
            return;
        }
        // send email...
    }
}
```

We are injecting the ApplicationContext so that we can tell the container to raise an event so all the listeners listening for this event can take necessary action

Finally the Listener class

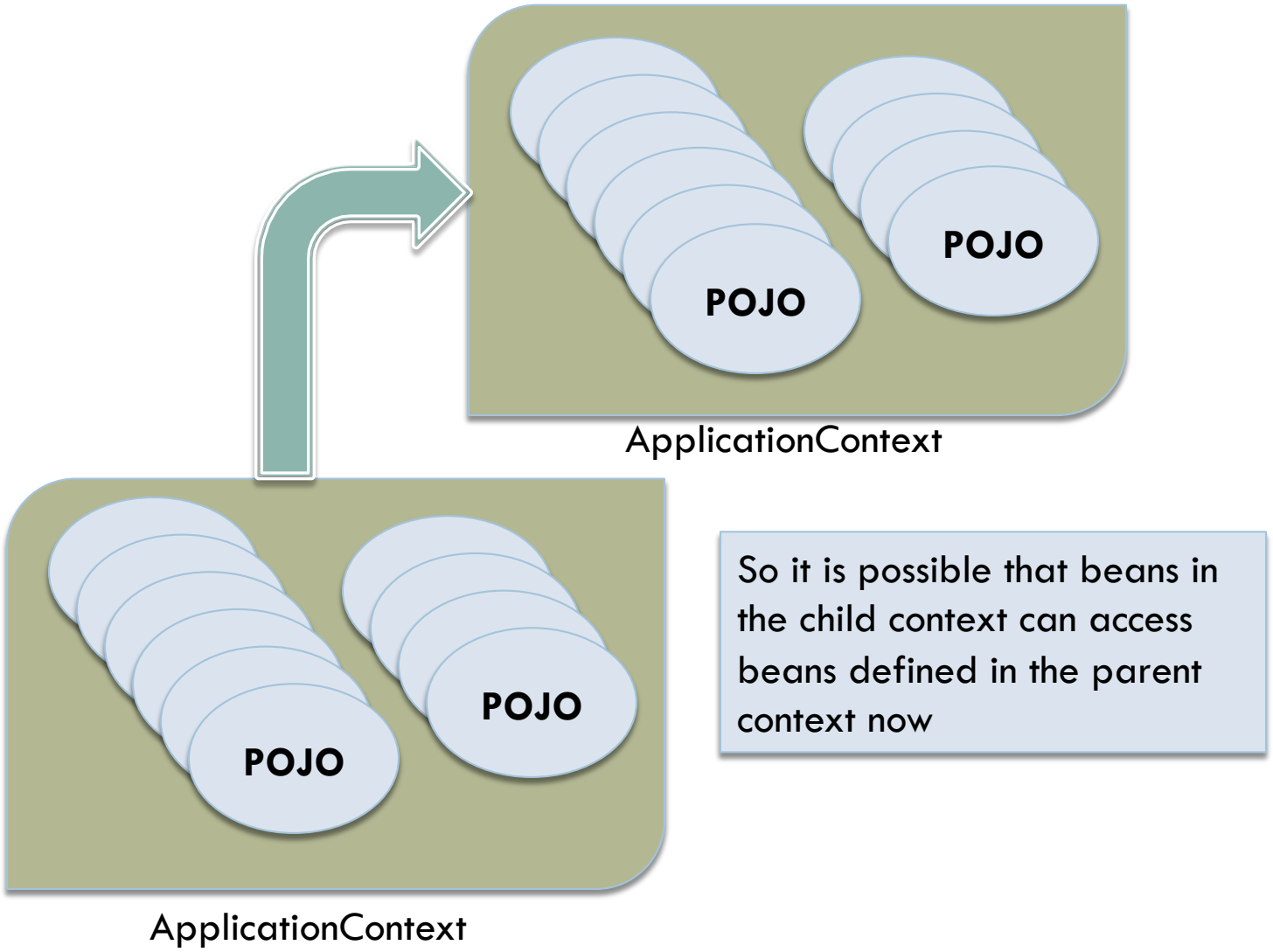

```
public class BlackListNotifier implements
ApplicationListener<BlackListEvent> {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(BlackListEvent event) {
            String blackListedEmailAddr = event.getAddress();
        // notify appropriate person...
    }
}
```

Rest of the logic goes in the listener class now as to what to do with this blacklisted email address. As you can see the main benefit is, EmailBean is lousely coupled with the Listener class. EmailBean doesn't need to know anything about who will handle blacklisted emails

Creating Multiple Contexts

- As we know by know, Spring provides all the services of an IoC container in the form of an ApplicationContext instance
- Each ApplicationContext instance can contain 'n' number of beans defined in xml files as well as annotations
- Each context can be linked with multiple xml files
- Spring framework supports hierarchical context creation. Which means you can have something like 'parent' and 'child' context

Cont'd…



So it is possible that beans in the child context can access beans defined in the parent context now

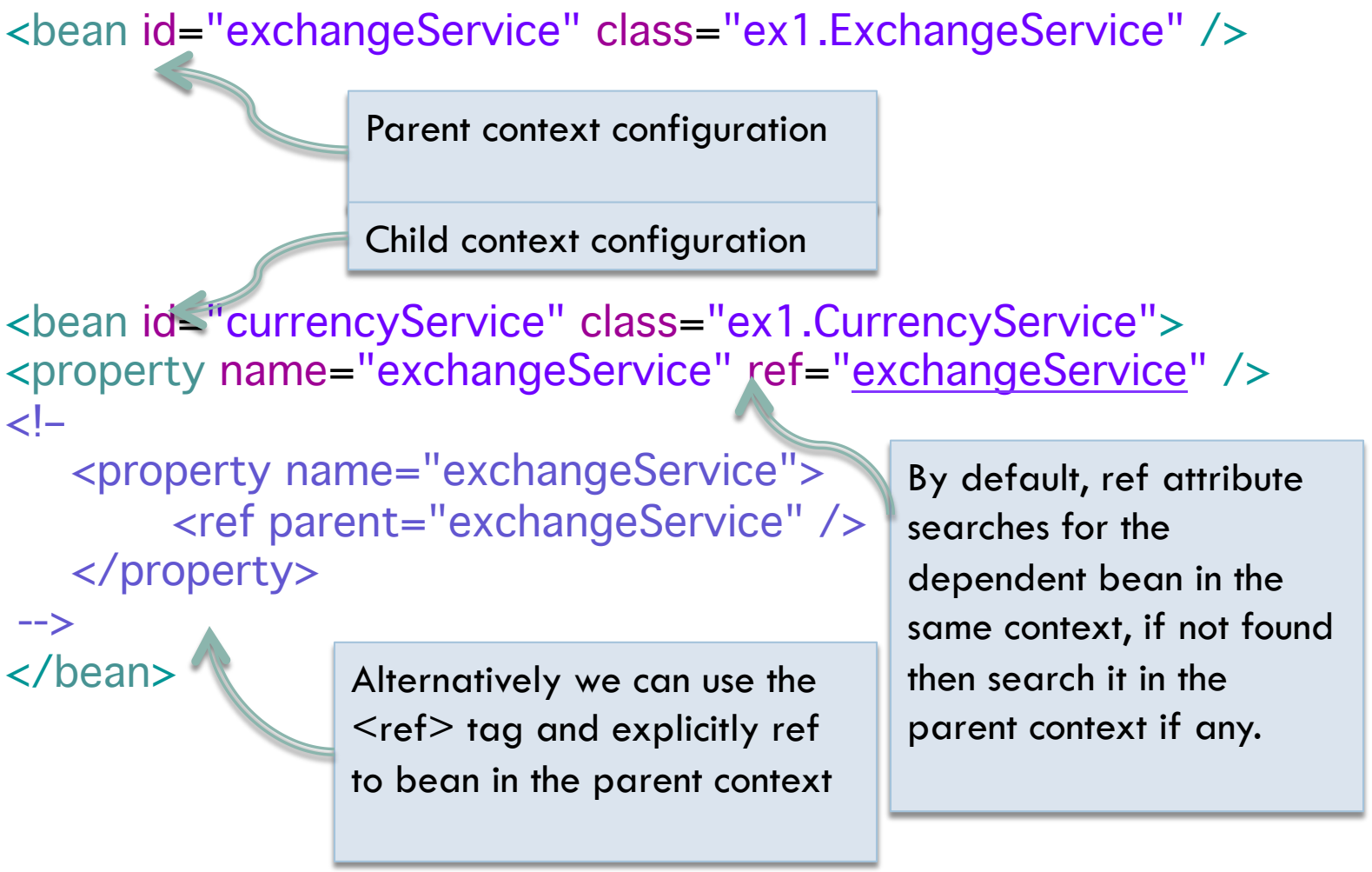
Example


```
//creating the parent context
ConfigurableApplicationContext pContext = new
    ClassPathXmlApplicationContext("ex5/context-1.xml");

//creating the child context
ConfigurableApplicationContext cContext = new
    ClassPathXmlApplicationContext(
                    {"ex5/context-2.xml"}, pContext);
```

One ApplicationContext instance can be associated with another ApplicationContext instance whereby creating parent/child context hierarchy. Broadly speaking, this is inheritance at the container level.

Which means, beans defined in the child context can access the beans defined in the parent context, but not the other way round.

The xml configuration



I18N support

- At the core level, we are provided with a *MessageSource* interface for supporting i18n
- MessageSource implementations provide access to resource bundle conveniently in a spring environment
- According the the bean lifecycle, any bean can get access to the MessageSource object by implementing the MessageSourceAware callback interface
  - Now we can simply autowire the MessageSource object, so need to implement any callback API

Example


```
public class OrderProcessingService implements MessageSourceAware {

    private MessageSource messageSource;
    @Override
    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
    public void placeOrder(int quantity) {
        int availableQuantity = 0; //just an assumption
        if(quantity > availableQuantity) {
          String errorMessage =
                  messageSource.getMessage(
                                  "insufficient.stock",
                                  new Object[]{availableQuantity},
                                  Locale.getDefault());
          throw new RuntimeException(errorMessage);
        }
    }
}
```

The configuration


```
<bean id="messageSource”
class=”o.s.context.support.ResourceBundleMessageSource">
    <property name="basename" value="ex6/mymessages" />
</bean>

<bean id="orderService” class="ex6.OrderProcessingService" />
```

```
#mymessages_en_US.properties
insufficient.stock=Sorry, not enough stock. Only {0} copies left.
```

No need to inject the MessageSource in OrderProcessingService as it implements the MessageSourceAware interface

Unit Testing support

- Spring is one of the frameworks which realized the importance unit testing and provided early support for the same.
- The intention is to make the core container capabilities accessible to test cases, i.e. Dependency Injection
- Unfortunately all the facilities provided by Spring are available only for Spring managed objects.
- JUnit test cases aren't managed by Spring so it's not directly possible, but it can be achieved!

Traditional Junit 4 style test case


```
//Writing a test class the way we have been doing it till now
public class SampleBeanTest {

    @Test
    public void testSampleBean() {
        //everytime we need to take care of creating an ApplicationContext
        ApplicationContext ctx = new
        ClassPathXmlApplicationContext("ex3/ex3-config.xml");
        //then we need to call getBean() to access a particular
        //bean which we need to test
        SampleBean sampleBean = ctx.getBean(SampleBean.class);
        //and then finally we get a change to test the bean method
        sampleBean.sampleMethod();
    }
}
```

This is how we have been testing the code so far!

Spring's support for JUnit


```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="ex3/ex3-config.xml")
public class SampleBeanTest2 {
    //Inject the bean which we need to test.
    //Why to do it manually. So DI in JUnit also!!
    @Autowired
    private SampleBean sampleBean;

    @Test
    public void testSampleBean() {
        //just need to now concentrate o
        sampleBean.sampleMethod();
    }
}
```

Basically the RunWith class will automatically create an ApplicationContext and inject the beans required

Additionally for testing transactional code, default support for rollback is provided in test cases. If the test succeeds, Spring will initiate a rollback rather than a commit which is generally what we do manually outside of Spring

Lot more about Spring IoC

Lab No. 4

Refer to the lab guide provided along with the eclipse project to proceed further

Topics in this Session

- Introduction to AOP
- What is an Aspect?
- Spring's support for AOP
  - Proxy based
  - Weaving
- Spring and AspectJ relationship
- Pointcuts, Joinpoints and other terminologies
- Different types of Aspects
  - Before, After, AfterReturning, AfterThrowing, Around

The problem


```
public String doSomething(String input) {
    // Logging
    System.out.println("entering business method with:" + input);
    // Security check for authorization of action (business-method)
    // transactionality
    try {
        // Start new session and transaction
        // Some business logic
        // Commit transaction
    } catch (Exception e) {
        // Rollback transaction
    } finally {
        // Close session
    }
    // Logging
    System.out.println("exiting business method with:" + input);
    return input;
}
```

IOC Example, where is AOP then!


```
public class BankAccount {
        public void transfer(BigDecimal amount, BankAccount recipient) {
                SecurityManager.hasPermission(this, Permission.TRANSFER,
                SecurityContext.getCurrentUser());
                recipient.deposit(this.withdraw(amount));
        }
        public void closeOut() {
                SecurityManager.hasPermission(this, Permission.CLOSE_OUT,
                SecurityContext.getCurrentUser());
                this.open = false;
        }
        public void changeRates(BigDecimal newRate) {
                SecurityManager.hasPermission(this, Permission.CHANGE_RATES,
                SecurityContext.getCurrentUser());
                this.rate = newRate;
        }
}
```

Identifying repetitive code


```
public class BankAccount {
        public void transfer(BigDecimal amount, BankAccount recipient) {
                recipient.deposit(this.withdraw(amount));
        }

        public void closeOut() {
                this.open = false;
        }

        public void changeRates(BigDecimal newRate) {
                this.rate = newRate;
        }
}
```

This is how the BankAccount class will look like after applying AOP

As you can see AOP has freed the object from the cross-cutting constraint of security authorization. The end result is a removal of duplicate code and a simplified class that is focused on its core business logic.

Cont'd…

- So how do we get the security checks back into the system? You can add the authorization mechanism into the execution path called aspect-oriented programming (**AOP**)
- **Aspects** are concerns of the application that apply themselves across the entire system. The SecurityManager is one example of a system-wide aspect, as its hasPermission methods are used by many methods. Other typical aspects include **logging, auditing, exception handling, performance monitoring** and **transaction management**
- These types of concerns are best left to the framework hosting the application, allowing developers to focus more on business logic

Cont'd…

- An AOP framework, such as Spring AOP, will **interject** (also called **weaving**) aspect code transparently into your domain model at runtime or compile time. This means that while we may have removed calls to the *SecurityManager* from the *BankAccount* class, the deleted code will still be executed in the AOP framework
- The beauty of this technique is that both the domain model (the BankAccount) and any client of the code are unaware of this enhancement to the code

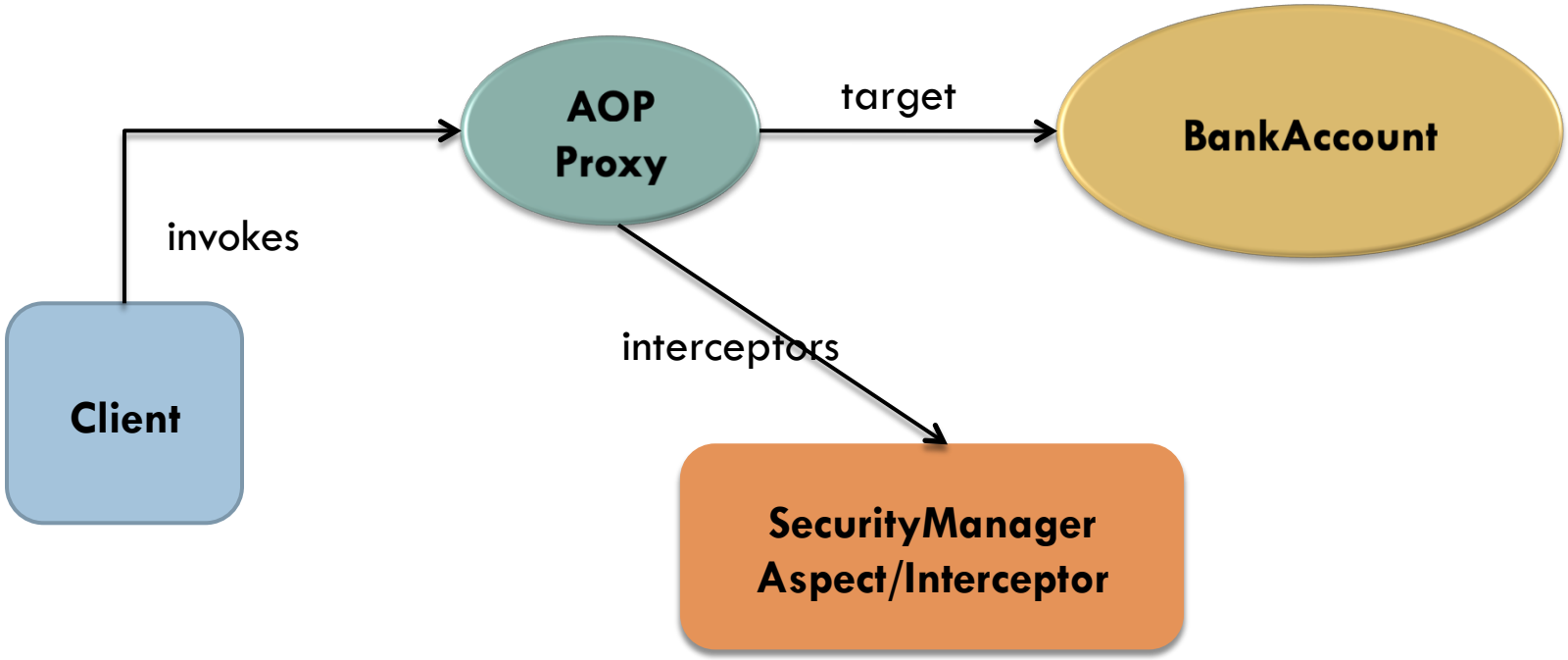
Different AOP implementations

- AOPAlliance API (Interceptor)
  - Supported in Spring by default
- Spring AOP (Advice)
  - Spring's own API
- AspectJ (Aspect)
  - A very powerful AOP framework
  - Supported by Spring since 2.0

Spring's dynamic proxies for AOP

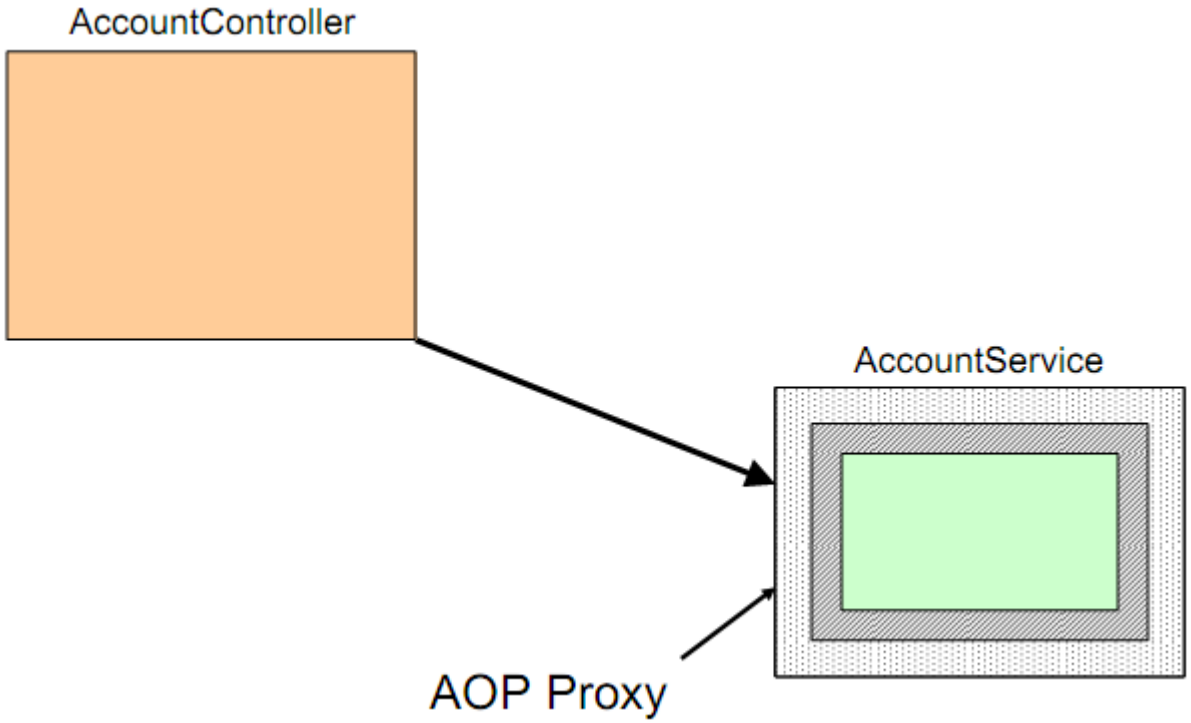
- Till Spring 2.0, Spring's implementation is what is called as **proxy-based** AOP implementation irrespective of which AOP library you are using
- These proxies essentially wrap a target object (the BankAccount instance) in order to apply aspects (SecurityManager calls) before and after delegation to the target object
- The proxies appear as the class of the target object to any client, making the proxies simple drop-in replacements anywhere the original target is used

Proxy approach



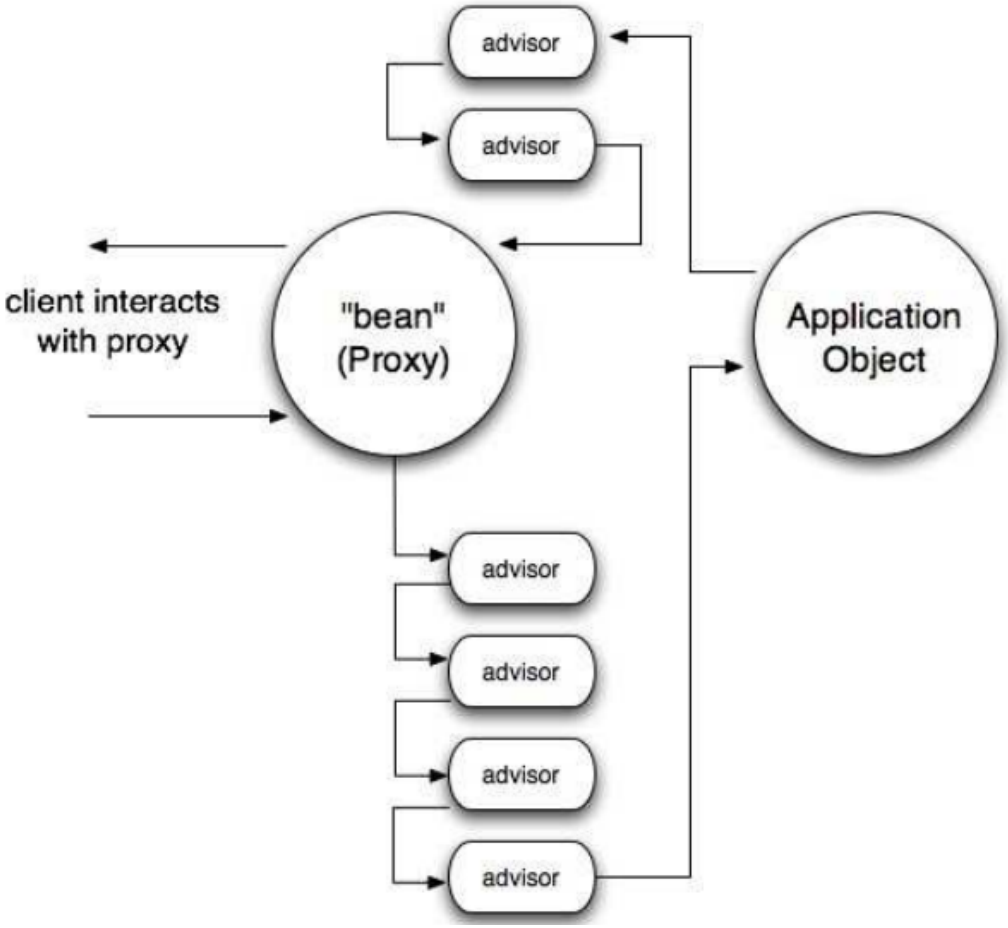
Since the client code should be unaware that he is talking to different class, the Proxy generated by Spring would either be a subclass of BankAccount or implement interfaces exposed by BankAccount class so that the client code remains transparent to the changes in the configuration.
Client invokes methods exposed by the proxy, which in turn will execute interceptors configured for the target bean

Cont'd…

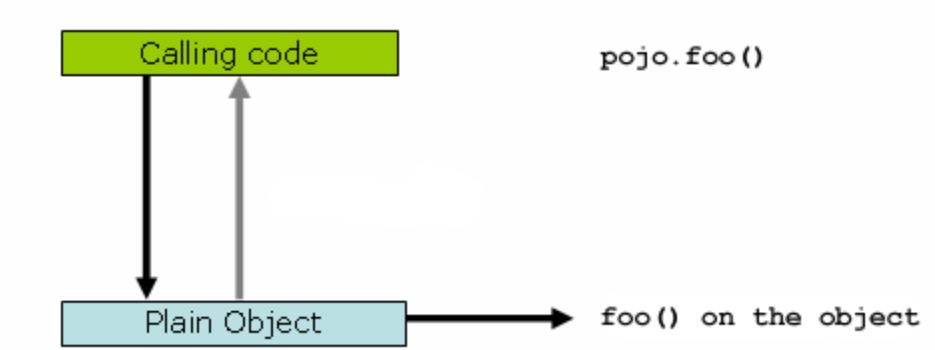


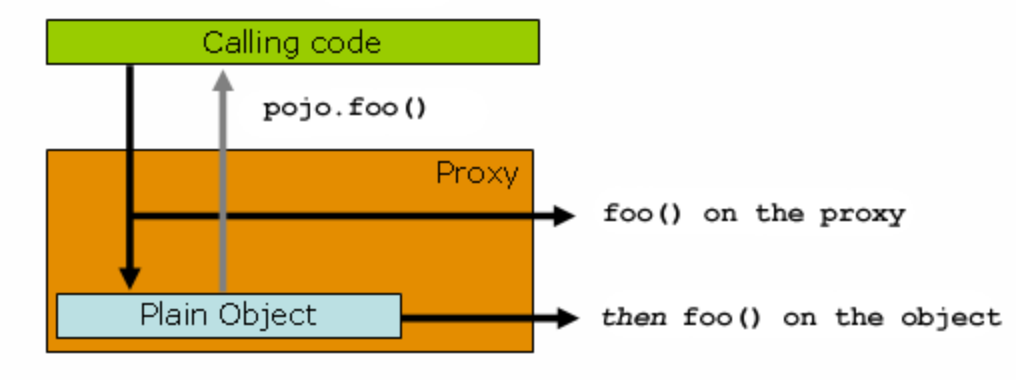
Client will never know that he is calling the method of a proxy class. Proxy classes are wrappers around our actual components and contain the same signature methods making it 100% transparent to the system!
In Spring, proxy classes are used in many places apart from AOP as well!

Cont'd…


advisor
advisor
client interacts
with proxy
"bean"
(Proxy)
Application
Object
advisor
advisor
advisor
advisor

One more diagram

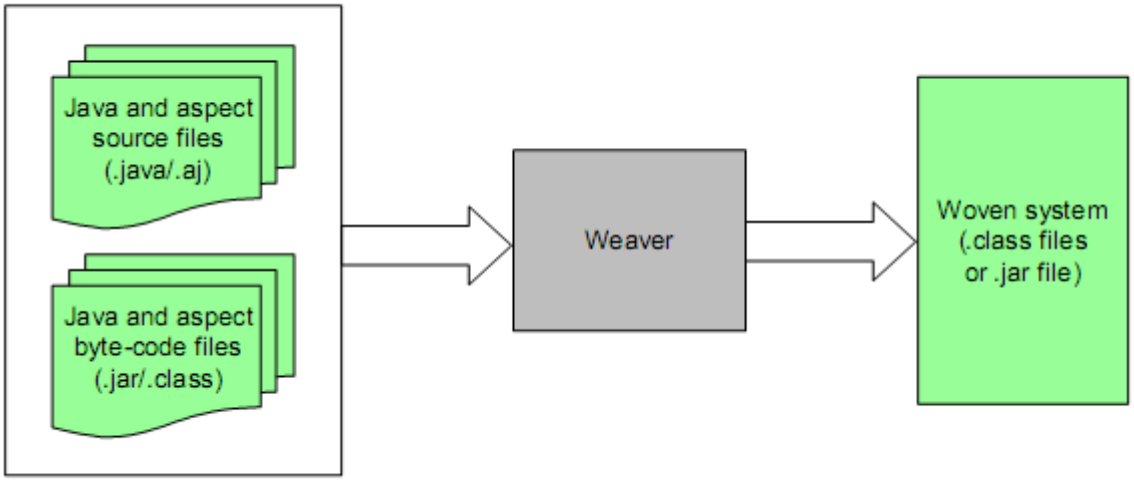




Weaving style of working with AOP

- In Spring 2.5, support for LTW (load time weaving) was also introduced. Weaving also means bytecode instrumentation
- There are possible ways of weaving the bytecode in AspectJ:
  - Compile/Binary weaving
    - Aspect + Source/Byte code = class files
  - Load time weaving (LTW)
    - Aspect + class files
    - Weave class files when being loaded in VM

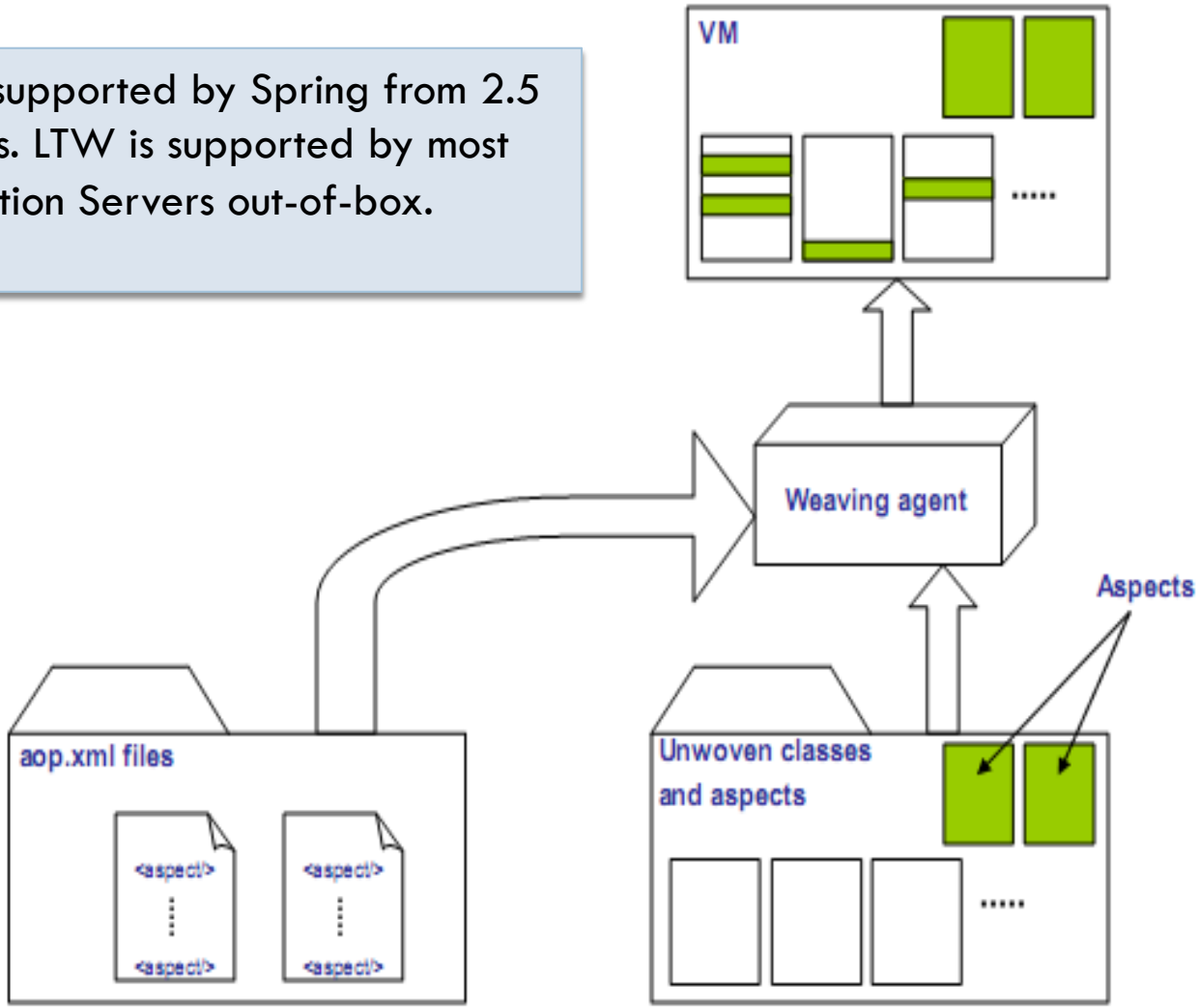
AspectJ weaver



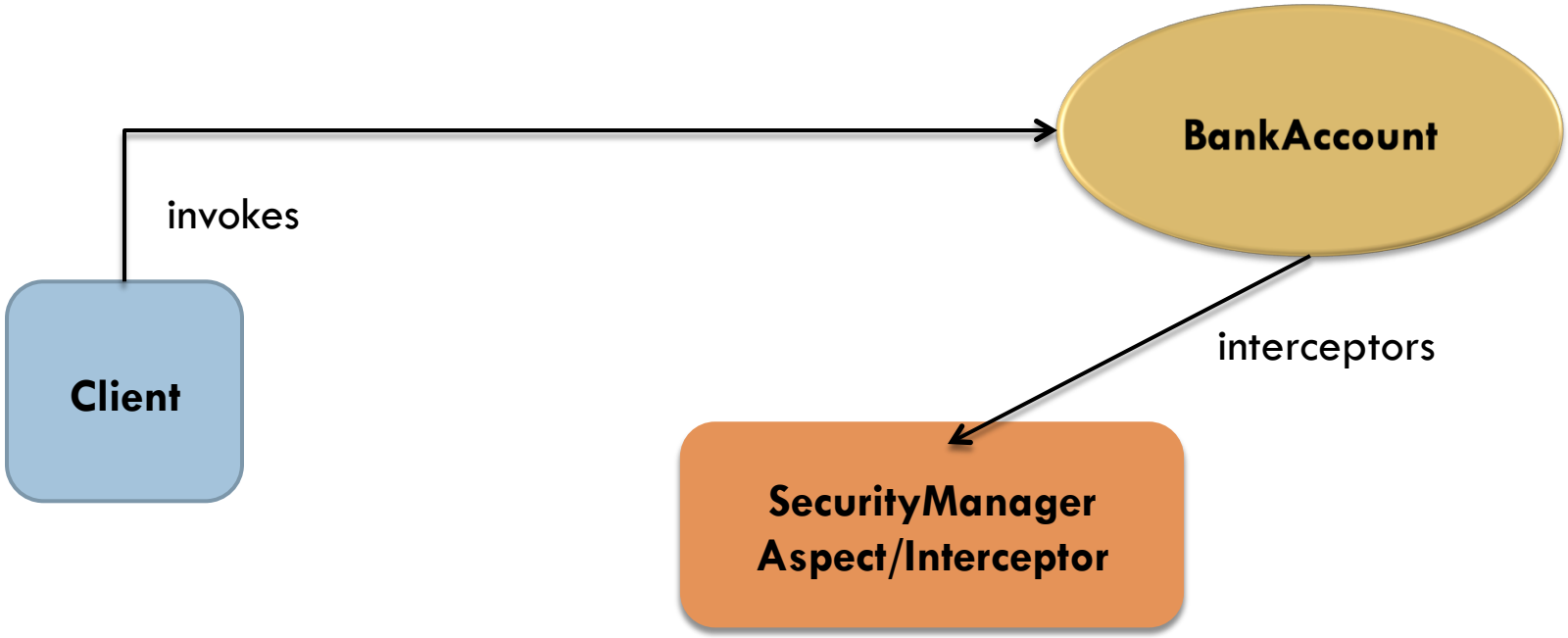
Compile-time weaving and binary weaving requires a separate AspectJ compiler, which has nothing to do with Spring. You will have to separate learn AspectJ to try it out. Nowadays, Eclipse plugins are also available for AspectJ making it much more easier to achieve the same.

Load time weaver

LTW is supported by Spring from 2.5 onwards. LTW is supported by most Application Servers out-of-box.



Weaving approach



With weaving as the approach used, the byte code of the BankAccount would be modified to invoke the Aspect/Interceptor directly

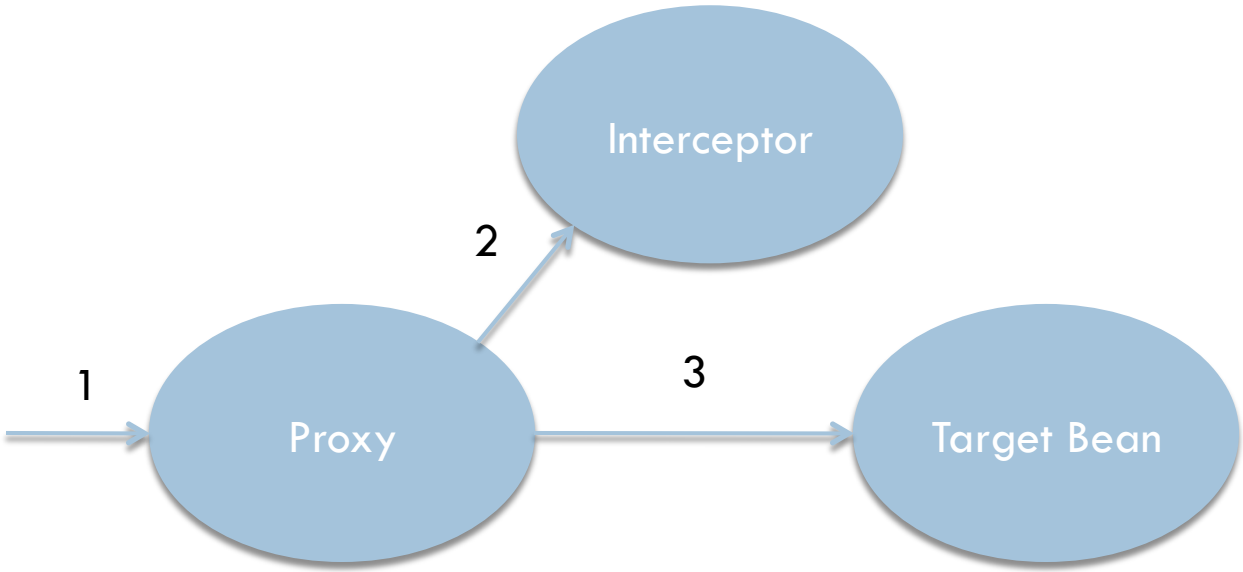
So when using AspectJ with Spring

- We need to decide which approach to use for working with AOP.
- **<aop:aspectj-autoproxy** /> to enable proxy based AOP implementation
- **<context:load-time-weaver** /> to enable to LTW (load time weaving) of classes
  - Works with Tomcat 5.5+, dm Server 1.0+, WebLogic 10, OC4J 10.x, Glassfish.
  - For Standalone/test applications we need to attach the weaver using –javaagent option to the VM

Types of Advice

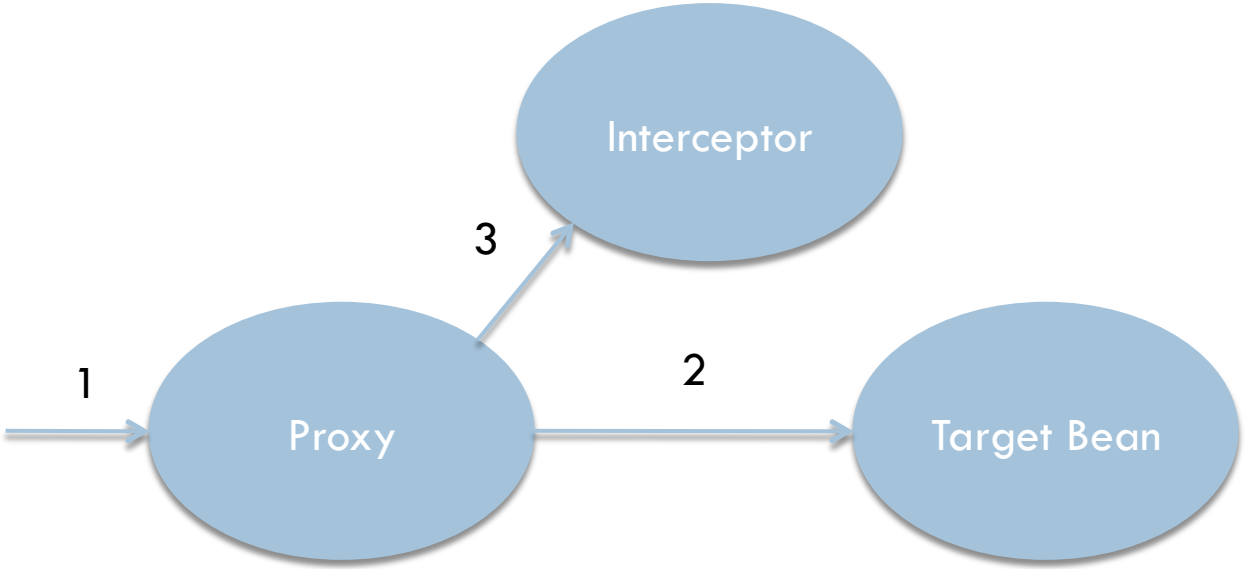
- While writing an aspect, we need to also decide when will this aspect/advice run:
  - Before
  - After
  - AfterReturning
  - AfterThrowing
  - Around

Before Advice



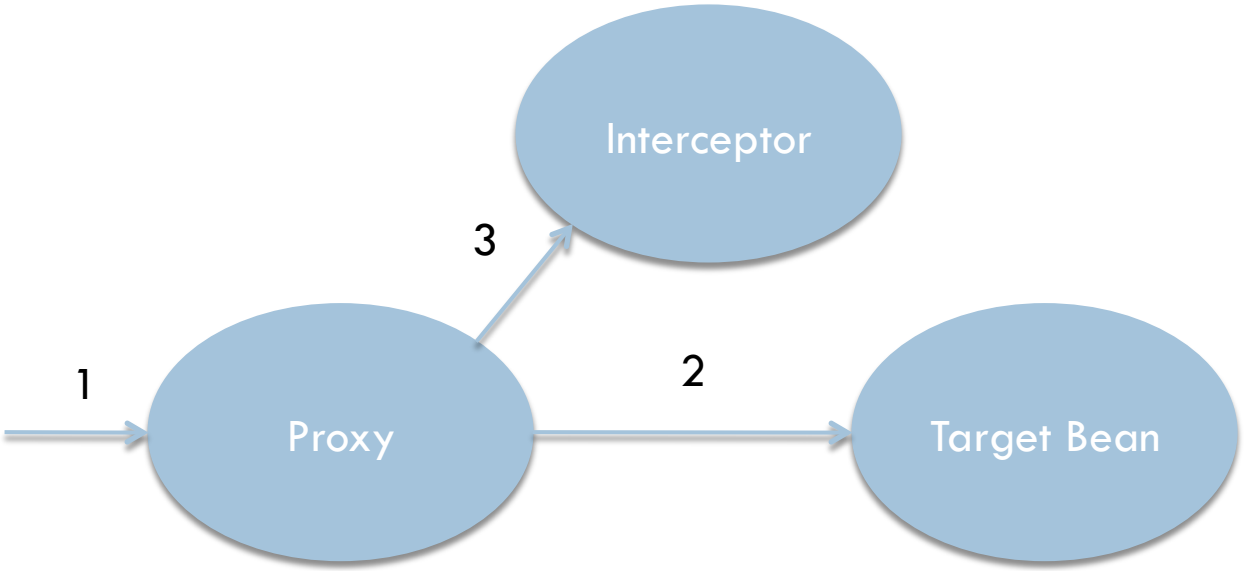
1. Client calls some method of the proxy class
2. The proxy executes the interceptor first
3. If the interceptor executes without any error, then finally invokes the real method of the bean class

After Advice



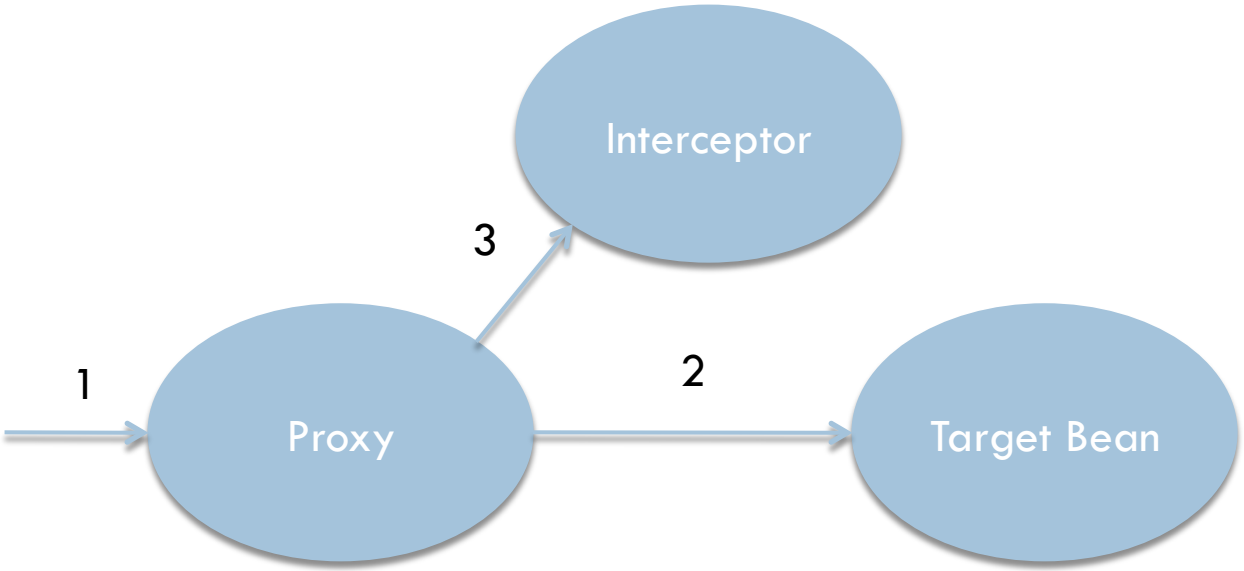
1. Client calls some method of the proxy class
2. The proxy executes real method first.
3. The interceptor is executed irrespective of the target bean method fails or succeeds. Logically acts like the finally block!

AfterReturning Advice



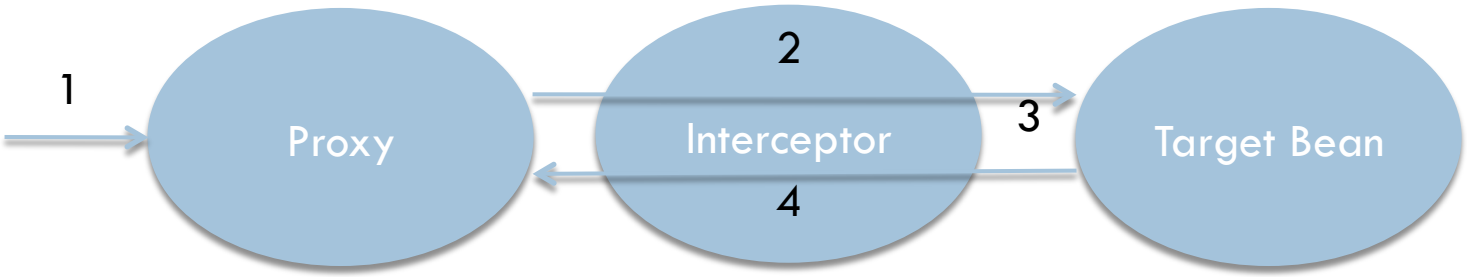
1. Client calls some method of the proxy class
2. The proxy executes real method first.
3. The interceptor is executed only if the target method returns successfully.

AfterThrowing Advice



1. Client calls some method of the proxy class
2. The proxy executes real method first.
3. The interceptor is executed only if the target method fails by throwing an exception.

Around Advice



1. Client calls some method of the proxy class
2. The proxy executes the interceptor first.
3. The interceptor is now responsible for proceeding to the target method of the bean class.
4. After the target method completes, the interceptor is again executed. So basically Around advice is like perfoming pre and post processing both.

AOP terminologies

- *Aspect* – A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is an example of crosscutting concern in J2EE applications. Aspects are implemented using Spring as Advisors or interceptors
- *Joinpoint* – Point during the execution of a program, such as a method invocation or a particular exception being thrown. In Spring AOP, a joinpoint is always a method invocation

Cont'd…

- *Advice* – Advice specifies what to do at a join point. In Spring, this is the additional behavior that Spring will inject around a method invocation
- *Pointcuts* – Pointcuts are expressions which identify where an advice should apply

Example

Log a message everytime, **before** a method is called..

Steps to be followed:
1. Write a class which will contain the common logging code. This called will be called as *Aspect*.
2. Either in xml or annotations, we will have to configure a **pointcut**, to specify when(*before*) and where(for all the beans or only a few or only for a particular method pattern) this aspect will apply.

The Sample bean class


```
public class CustomerServiceImpl implements CustomerService {

        public void applyForChequeBook(long acno) {
                System.out.println("applyForChequeBook method called..");
        }

        public void stopCheque(long acno) {
                System.out.println("stopCheque method called..");
        }

        public void applyForCreditCard(String name, double salary) {
                System.out.println("applyForCreditCard method called..");
        }
        ...
}
```

This is the bean class for which we would like to apply our aspect

The Aspect class


```
//This is an Aspect class
public class LoggingAspect {

    //This is an advice.
    //Pointcut means where will this advice be applied.
    public void log(JoinPoint joinPoint) {
            System.out.println("common logging code executed for : "+joinPoint);
    }
}
```

Context about the intercepted point

Methods of the Aspect class can be logically be called as advices. Since we are not using annotations, the pointcut configuration will be done in an xml file

The configuration


```
<aop:aspectj-autoproxy />

<bean id="customerService" class="service.CustomerServiceImpl" />

<bean id="loggingAspect" class="ex1.LoggingAspect" />

<aop:config>
    <aop:pointcut id="pointcut1"  expression="execution(public * apply*(..))" />
        <aop:aspect ref="loggingAspect">
            <aop:before method="log" pointcut-ref="pointcut1" />
        </aop:aspect>
</aop:config>
```

We need to use the AOP namespace in the xml to configure aspects and pointcuts

Annotation style configuration


```
@Aspect
public class LoggingAspect {

    @Before("execution(public * apply*(..))")
    public void log(JoinPoint joinPoint) {
        System.out.println("common logging code executed for : "+joinPoint);
    }
}
```

This is the annotated version of the same aspect as before. As you can see the instruction about **when** and **where** will the log method execute is configured using relevant annotations. You decide which approach is better!

About JoinPoint object


```
@Aspect
public class LoggingAspect2 {

    @Before("execution(public * apply*(..))")
    public void log(JoinPoint joinPoint) {
            Object proxyObject = joinPoint.getThis();
            Object targetBean = joinPoint.getTarget();
            Object[] args = joinPoint.getArgs();
            Signature signature = joinPoint.getSignature();
            //some logging code here
    }
}
```

The JoinPoint object provides the runtime information about the current point of invocation

Useful JoinPoint context

- this
  - The currently executing object that been intercepted (a proxy in Spring AOP)
- target
  - The target of the execution (typically your object)
- args
  - The method arguments passed to the join point
- signature
  - The method signature of the join point

Pointcuts

- One of the reason why AspectJ was adopted by Spring was because of a very powerful and easy to learn pointcut expressions
- This is the most important thing in our configuration, because a pointcut identifies the places where our aspects will execute

Pointcut expressions

- Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:
  - *execution, within, this, target, args*
- Spring AOP also supports an additional PCD named as:
  - *bean(idorname)*
- The *execution* expression is the most commonly used PCD used in our Spring applications

execution expression

- The syntax of execution expression is:
  - execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
- Can chain together to create composite pointcuts
  - &&(and), ||(or), !(not)

Example

execution(public * apply*(..))

Public method returning anything and the method name begins with apply taking any number of arguments

execution(public * com.package.service.*. *(..))

Any public method, return type anything, belonging to any class from the service package any number of arguments

execution(public * apply*(java.lang.String,*))

Any void method of any class taking a String as the first argument and the second argument can be anything

Example

execution(public * com.package.service..*.*(..))

Any method within the whole service package including subpackages taking any number of arguments

execution(@Log * *(..))

Only those methods which are annotated with @Log annotation taking any number of arguments and returning anything

execution(public * apply*(..)throws CustomException)

Only those methods whose signature contains the following throws clause returning anything taking any number of arguments

Example on other expressions

within(com.package.service.*)

For any class within the service package. When using within expression we cannot specify the method name or argument bindings

within(com.package.service..*)

Within the service package and any of it's sub packages and any class

within(com.package.service.CustomerService+)

Withing the service package and for any bean of type CustomerService. The + operator over here indicates, any bean of that type

Cont'd…

@annotation(com.package.Log)

Only those methods which are annotated with @Log annotation

@target(com.package.Log)

Only those bean classes which are annotated with @Log annotation

bean(*Service)

Only those beans whose id/name matches the expression

Passing parameters to advice

- As we seen, a JoinPoint object provides all the details required by the advice to perform any kind of operation
- Another way of achieving the same is by using *target*, *args* and *this* expressions in the Pointcut declaration

Example


```
@Aspect
public class OrderValidator {

        @Before("execution(* placeOrder(..)) && args(order)")
        public void validateOrder(Order order) {
                //some order validation logic here
        }
}
```

Over here, we are telling the container to pass the Order object passed as a parameter to the placeOrder method to the validateOrder advice for validation purpose

Named Pointcut

- This is another very important feature of Spring AOP. Since pointcut expressions are lengthy and if you need to use the same pointcut in multiple places, then instead of repeating the expression we can give a logical name to that pointcut and refer to it by that name
- It will be more clear with the help of an example!

Example


```
@Aspect
public class PointcutConfig {

    @Pointcut("execution(public * com.package.service.*.*(..))")
    public void serviceComponents() {}

    @Pointcut("execution(* apply*(..))")
    public void applyMethods() {}

}
```

Basically we are creating aliases for the pointcuts so that we can refer to the same easily now.

Cont'd…


```
@Aspect
public class LoggingAspect {

        @Before("com.package.PointcutConfig.applyMethods()")
        public void log(JoinPoint joinPoint) {
                System.out.println("log advice executed for method : "+
                        joinPoint.getSignature().getName());
        }

        @Before("com.package.PointcutConfig.serviceComponents()")
        public void timeLog(JoinPoint joinPoint) {
                System.out.println("request for method : "+
                        joinPoint.getSignature().getName()+
                                " occurred at "+new Date());
        }
}
```

Instead of writing the actual expression, we are referring to the pointcut aliases

After and AfterReturning advice


```
@Aspect
public class LoggingAspect {

	@After("bean(*Service)")
	public void log(JoinPoint joinPoint) {
		System.out.println("log advice got executed after call to method : "+
				joinPoint.getSignature().getName());
	}

	@AfterReturning(pointcut="execution(* balance(..)) && args(acno)",
					returning="balance")
	public void validate(JoinPoint joinPoint, long acno, double balance) {
		System.out.println("validate advice called after successful return");
		System.out.println("acno passed "+acno+" and the balance returned "+balance);
	}
}
```

In AfterReturning advice, we can even process the return value of the target method which was called.

Exception handling using AOP

- Exception handling is one more important aspect of our application development
- In many cases exeption handling is merely catching system exceptions, wrapping it up as user defined exceptions and throwing it to the caller
- This means we end up writing duplicate try/catch blocks in many places, which can easily be eliminated using AOP

Example


```
public class BusinessComponent implements BusinessInterface {

    public void someBusinessMethod() throws BusinessException {
        //we are raising a different exception to see the aspect
        //coming into action
        //assume that you are connecting to the database and
        //something goes wrong
        throw new RuntimeException();
    }
}
```

We are trying to create a scenario where the code is raising a system exception and the exception handling code has been into a common place, i.e. Aspects

AfterThrowing Advice


```
@Aspect
public class ExceptionHandler implements MessageSourceAware {

    private MessageSource messageSource;
    ...

    @AfterThrowing(pointcut="execution(* businessMethod(..))",
                   throwing="ex")
    public void handleException(JoinPoint jp, RuntimeException ex)
    throws Throwable {
        System.out.println("Routine exception handling code here.");
        ...
        throw new BusinessException(
                messageSource.getMessage(jp.getSignature().getName(),
                                         null, null));
    }
}
```

Around Advice

- If we need to do both pre and post processing, i.e. we need to excute some code before and after the method is called
- Also we might want to handle exceptions, return values all in one place
- So basically, Around advice combines the capability of Before, After, AfterReturning and AfterThrowing advice in one place

Example


```
@Aspect
public class ProfilingAspect {

        @Around("execution(* *(..))")
        public Object profile(ProceedingJoinPoint joinPoint) throws Throwable {
                StopWatch watch = new StopWatch();
                watch.start(joinPoint.getSignature().getName());
                try {
                        return joinPoint.proceed(); //calling the target method
                }
                catch(Exception e) {
                        throw e;
                }
                finally {
                        watch.stop();
                        System.out.println(watch.prettyPrint());
                }
        }
}
```

Load Time Weaving

- LTW can be achieved from Spring 2.5 onwards instead of the proxy approach

Example


```
@Aspect
public class ProfilingAspect {

        @Around("execution(* *(..))")
        public Object profile(ProceedingJoinPoint joinPoint) throws Throwable {
                StopWatch watch = new StopWatch();
                watch.start(joinPoint.getSignature().getName());
                try {
                        return joinPoint.proceed(); //calling the target method
                }
                catch(Exception e) {
                        throw e;
                }
                finally {
                        watch.stop();
                        System.out.println(watch.prettyPrint());
                }
        }
}
```

We will try weaving this aspect to our bean methods

META-INF/aop.xml file


```
<aspectj>

    <weaver>
        <!-- only weave classes in our application-specific packages -->
        <include within="service.*" />
        <include within="ex8.*" />
    </weaver>

    <aspects>
        <!-- weave in just this aspect -->
        <aspect name="ex8.ProfilingAspect" />
    </aspects>

</aspectj>
```

To use LTW, AspectJ requires an aop.xml file containing information about classes which needs to be weaved and the aspects which will be woven in those classes

The test class


```
public class WeavingTest {

        @Autowired private CustomerService customerService;

        @Test
        public void testAroundAdvice() {
                //calling getClass to verify whether a proxy was created
                //or the same class was weaved

        System.out.println(customerService.getClass().getName());
                customerService.applyForChequeBook(12345);
                //manually creating a new instance bypassing sping
                customerService = new CustomerServiceImpl();
                customerService.applyForChequeBook(12345);
                customerService.stopCheque(12345);
        }
}
```

Running the test class

- For standalone and test environments, you will have to run the code by setting the following vm argument:
  - -javaagent:spring-instrument-{version}.RELEASE.jar

Spring and AspectJ

Lab No. 5

Refer to the lab guide provided along with the eclipse project to proceed further

Topics to be covered

- Spring's role in supporting data access in an enterprise application
- Support for data access technologies
  - JDBC, Hibernate, JPA

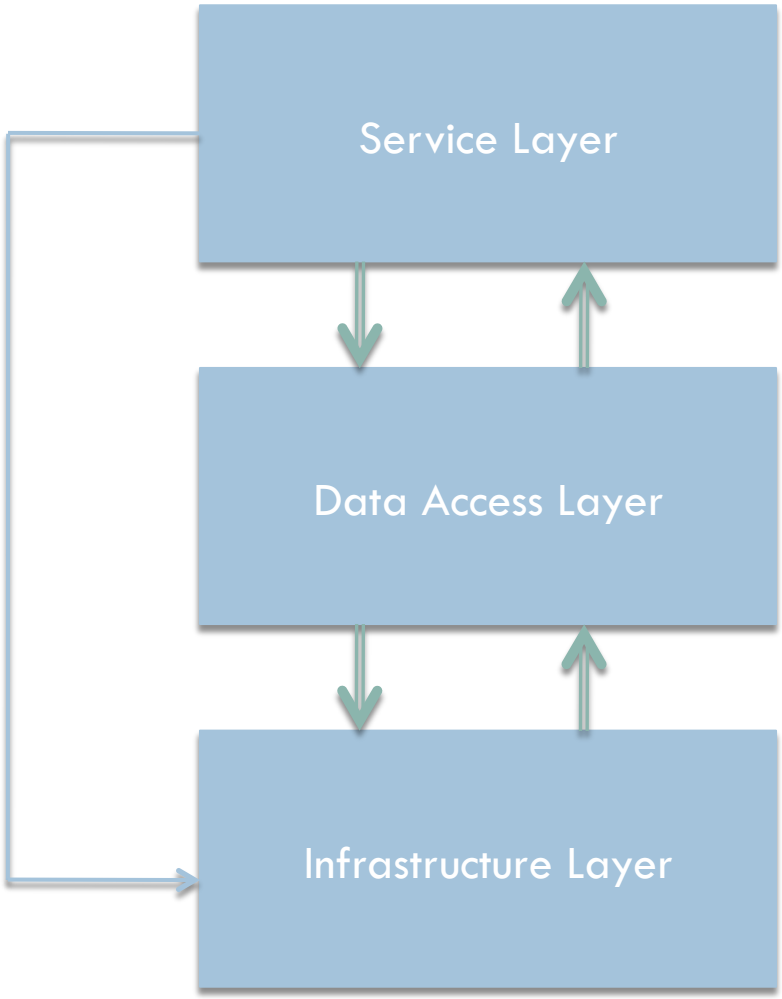
Role of Spring while accessing data

- Provide comprehensive data access support
  - To make data access easier to do it effectively
- Enable a layed application architecture
  - To isolate an application's business logic from the complexity of data access
- Spring manages resources for you
  - Eliminates boilerplate code
  - Reduces likelihood of bugs

Cont'd…

- Declarative transaction management
  - Transaction boundaries declared via configuration
  - Enforced by a Spring transaction manager
- Automatic connection management
  - Connections acquired/released automatically
- Intelligent exception handling
  - Root cause failures always reported
  - Resources always released properly

Layered Architecture



The Service Layer

- Defines the public functions of the application
  - Clients call into the application through this layer
- Encapsulates the logic to carry out each application function
  - Delegates to the infrastructure layer to manage transactions
  - Delegates to the data access layer to map persisent data into a form needed to execute the business logic

The Data Access Layer

- Used by the service layer to access data needed by the business logic
- Encapsulates the complexity of data access
  - The use of data access API
    - JDBC, Hibernate, etc
  - The mapping of data into a form suitable for business logic
    - A JDBC ResultSet to a domain object graph

The Infrastructure Layer

- Exposes low-level services needed by other layers
  - Infrastructure services are provided by Spring
  - Developers typically do not write them
- Likely to vary between environments
  - Production vs. test

Layers working together

- Client calls a function of the service layer
- A service initiates a function of the application
  - By delegating to the transaction manager to begin a transaction
  - By delegating to repositories to load data for processing
    - All data access calls participate in a transaction
    - Repositories oftern return domain objects that encapsulate domain behaviors

Cont'd…

- A service continues processing
  - By executing business logic
  - Often by coordinating between domain objects loaded by repositories
- And finally, completes processing
  - By updating changed data and committing the transaction

Introducing Spring JDBC Support

- Introduction to Spring JDBC
  - Problems with traditional JDBC
    - Results in reduntant, error prone code
    - Leads to poor exception handling
  - Spring's JdbcTemplate API
    - Configuration
    - Query execution
    - Working with result sets
    - Exception handling

Redundant, error prone code


```
public List<Flight> getAvailableFlights() {
        Connection conn = null;
        PreparedStatement pst = null;
        ResultSet rs = null;
        try {
                conn = dataSource.getConnection();
                pst = conn.prepareStatement("select * from flights_test");
                rs = pst.executeQuery();
                List<Flight> rows = new ArrayList<Flight>();
                while(rs.next()) {
                        Flight f = new Flight();
                        f.setFlightNo(rs.getString(1));
                        f.setCarrier(rs.getString(2));
                        f.setFrom(rs.getString(3));
                        f.setTo(rs.getString(4));
                        rows.add(f);
                }
                return rows;
        }
        catch(SQLException e) {
                throw new RuntimeException(e);
        }
        finally {
                try { rs.close(); pst.close(); conn.close(); } catch(Exception e) { }
        }
}
```

Spring's JdbcTemplate

- Greatly simplifies use of the JDBC API
  - Eliminates repetitive boilerplate code
  - Alleviates common causes of bugs
  - Handles SQLExceptions properly
- Without sacrificing power
  - Provides full access to the standard JDBC constructs

Introduction to JdbcTemplate API


```
public int getTotalFlights() {
        return jdbcTemplate.queryForObject(
                "select count(*) from flights_test", Integer.class);
    }
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling any exceptions
- Release of the connection

All the steps managed by the Template class

Spring JDBC – Who does what?

Action	Spring	You
Define the connection parameters		X
Open the connection	X	
Specify the SQL statement		X
Declare parameters and provide parameter values		X
Prepare and execute the statement	X	
Setup the loop to iterate through the results (if any)	X	
Do the work for each iteration		X
Process any exception	X	
Handle transactions	X	
Close the connection, statement and resultset	X	

Creating a JdbcTemplate

- JdbcTemplate class requires a DataSource to be supplied for successful creation

```
jdbcTemplate = new JdbcTemplate(dataSource);
```

- NamedParameterJdbcTemplate class allows the usage of named parameters ':name' rather than traditional '?' similar to Hibernate

```
jdbcTemplate = new NamedParameterJdbcTemplate
                                        (dataSource);
```

Example


```
public class FlightRepositoryImpl implements FlightRepository {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
```

Cont'd…


```
public class FlightRepositoryImpl extends JdbcDaoSupport
implements FlightRepository {

        public List<Flight> getAvailableFlights() {
                JdbcTemplate jdbcTemplate = getJdbcTemplate();
```

Alternatively we can extend JdbcDaoSupport/SimpleJdbcDaoSupport/ NamedParameterJdbcDaoSupport class which will create the respective template object and provide convenient access to it.

Such support classes are not used now a days

Using JdbcTemplate class methods

- Simple select query

```
public int getTotalFlights() {
        return jdbcTemplate.queryForObject(
                "select count(*) from flights_test", Integer.class);
    }
```

- Query with bind parameters

```
public int getTotalFlights(String carrier) {
        return jdbcTemplate.queryForObject(
        "select count(*) from flights_test where carrier = ?",
                new Object[]{carrier}, Integer.class);
    }
```

Cont'd…

- JdbcTemplate class supports varargs

```
public int getTotalFlights(String carrier) {
        return jdbcTemplate.queryForObject(
        "select count(*) from flights_test where carrier = ?",
                        Integer.class, carrier);
}
```

- Query for a single row

```
public Map getFlightInfo(String flightNo) {
        return jdbcTemplate.queryForMap(
                "select * from flights_test where flightno=?",
                        flightNo);

}
```

Cont'd…

- Query for multiple rows

```
pubic List getFlights(String carrier) {
        return jdbcTemplate.queryForList(
        "select * from flights_test where carrier = ?", carrier);
}
```

- returns a List where each element of the List contains a Map object holding column name, column value pair data

Cont'd…

- Query for domain objects

```
public List<Flight> getAvailableFlights(String carrier) {
 class FlightMapper implements RowMapper<Flight> {
        @Override
        public Flight mapRow(ResultSet rs, int index) throws SQLException {
           Flight flight = new Flight();
           flight.setFlightNo(rs.getString(1));
           flight.setCarrier(rs.getString(2));
           flight.setFrom(rs.getString(3));
           flight.setTo(rs.getString(4));
           return flight;
        }
 }
 return jdbcTemplate.query(
 "select * from flights_test where carrier = ?", new FlightMapper(), carrier);
}
```

Cont'd…

- For all other DML operations

```
public void newFlight(Flight flight) {
        jdbcTemplate.update(
            "insert into flights_test values(?, ?, ?, ?)",
                    flight.getFlightNo(), flight.getCarrier(),
                    flight.getFrom(), flight.getTo());
}
```

Please check the reference and API documentation for further usages of JdbcTemplate API

Spring JDBC Support

Lab No. 6

Refer to the lab guide provided along with the eclipse project to proceed further

Topics in this Session

- Spring and Hibernate integration
  - Configure Hibernate SessionFactory in Spring
  - Using HibernateTemplate class for managing Hibernate Session object
- Spring and JPA integration
  - Confure EntityManagerFactory in Spring
  - Using JpaTemplate class for managing JPA EntityManager object

About Spring and Hibernate

- Hibernate is one of the most powerful ORM technology used widely today
- Hibernate is 100% JPA compliant ORM
- Spring supports native Hibernate integration as well as integration via JPA
- The real benefit of Spring used along with Hibernate/ JPA would be:
  - Delegating SessionFactory creation to Spring
  - Use DI wherever we need the SessionFactory
  - **Manage transactions using Spring's declarative TransactionManager**

Possible ways of writing Repository class


```
public class HibernateProductRepository implements
ProductRepository {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
```

Configure the SessionFactory in the xml like any other bean and then inject it in the repository like normal

Cont'd…

- Generally we write:

```
Session session = sessionFactory.getCurrentSession();
```

- But then we also need take care of binding the Session with a transaction. In the past we use to a helper class provided by Spring called as HibernateTemplate
- HibernateTemplate class is **not** required anymore, because Spring provides transactional session objects directly, so no need to use HibernateTemplate at all

Example using HibernateTemplate


```
public class HibernateProductRepository implements
ProductRepository {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        hibernateTemplate = new HibernateTemplate(sessionFactory);
    }
```

HibernateTemplate needs a SessionFactory object. Similar to JdbcTemplate, HibernateTemplate can be configured in the xml file and injected directly in the repository class as well

Basic configuration


```
<bean id="exampleSessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource"><ref local="exampleDataSource"/></property>
        <property name="hibernateProperties">
                <value>
                        hibernate.dialect=org.hibernate.dialect.MySQLDialect
                </value>
        </property>
        <property name="mappingResources">
                <list>
            <value>Product.hbm.xml</value>
                </list>
        </property>
    </bean>

    <bean id="productRepository" class="ex1.HibernateProductRepository">
        <property name="sessionFactory" ref local="exampleSessionFactory"/>
    </bean>
```

Configuring the SessionFactory and injecting it in the Repository class

Spring JPA Support

- In JPA, a persistent unit is represented as an object of EntityManagerFactory interface similar to SessionFactory interface in Hibernate
- To persist/load object graph we need to use EntityManager interface and it's method similar to Session interface in Hibernate

Example


```
public class JpaProductRepository implements ProductRepository {

    private EntityManagerFactory entityManagerFactory;

    public void setEntityManagerFactory(
                    EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
    }

    @Override
    public void add(Product product) {
        EntityManager entityManager =
                    entityManagerFactory.createEntityManager();
        ...
```

Once again we will have to deal with transactional issues in this code

Cont'd…


```
public class JpaProductRepository implements ProductRepository {

        @PersistenceContext
        private EntityManager entityManager;
```

Spring support the standard **@PersistenceContext** annotation, so you can directly inject the EntityManager object. Much

JPA related configuration


```
<bean id="entityManagerFactory"
class=”o.s.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource” ref local="exampleDataSource” />
</bean>

<bean id="productRepository" class="ex2.JpaProductRepository">
      <property name="entityManagerFactory"
                ref="entityManagerFactory" />
</bean>
```

Instead of configuring the EntityManagerFactory, we can also lookup and access the EntityManagerFactory instance provided by the Application Server

Spring Hibernate/JPA Integration

Lab No. 7

Refer to the lab guide provided along with the eclipse project to proceed further

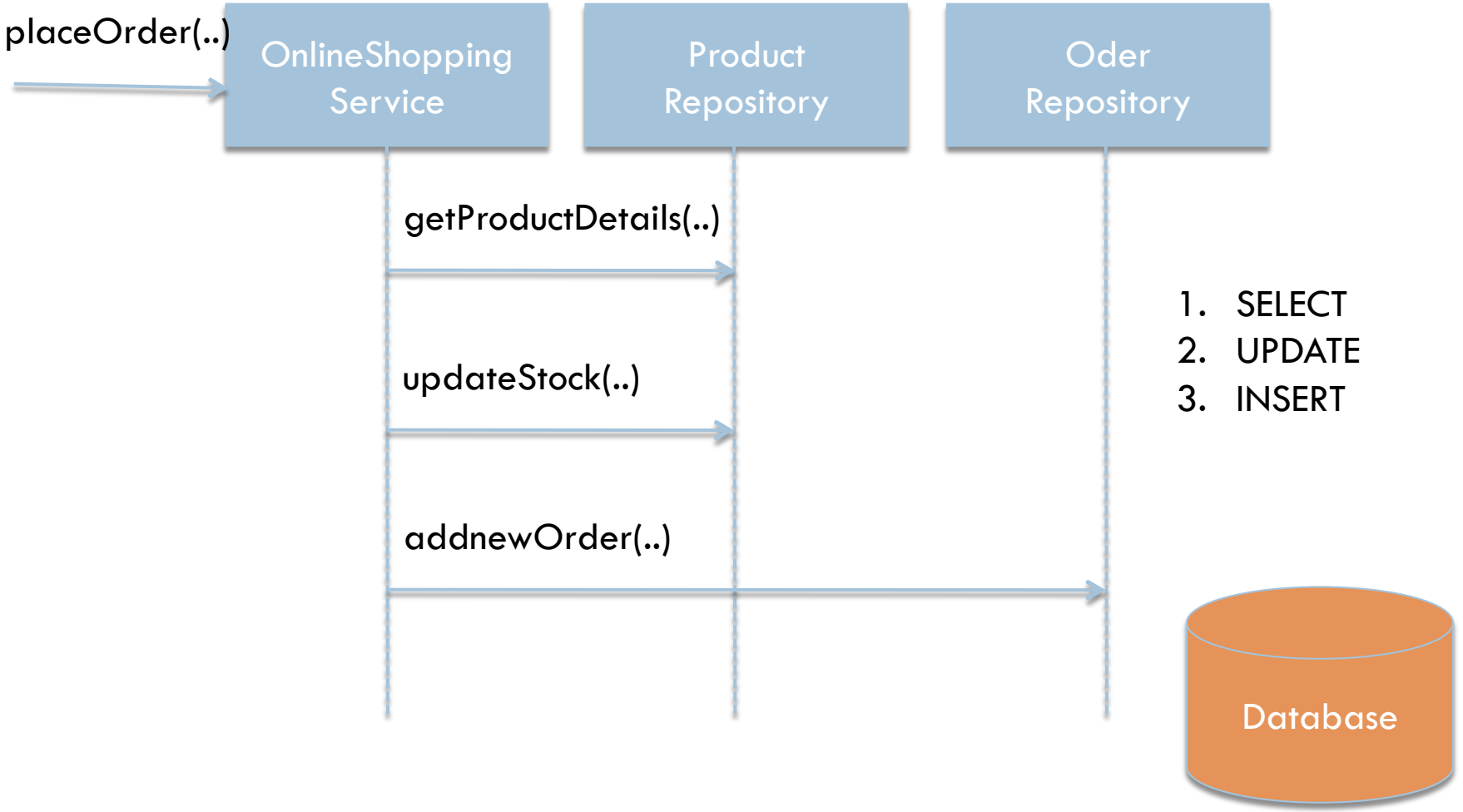
Topics to be covered

- Why transactions?
- Local transaction management
- Distributed transaction management
- Spring transaction management
- Declarative transaction management
- Transaction propagation

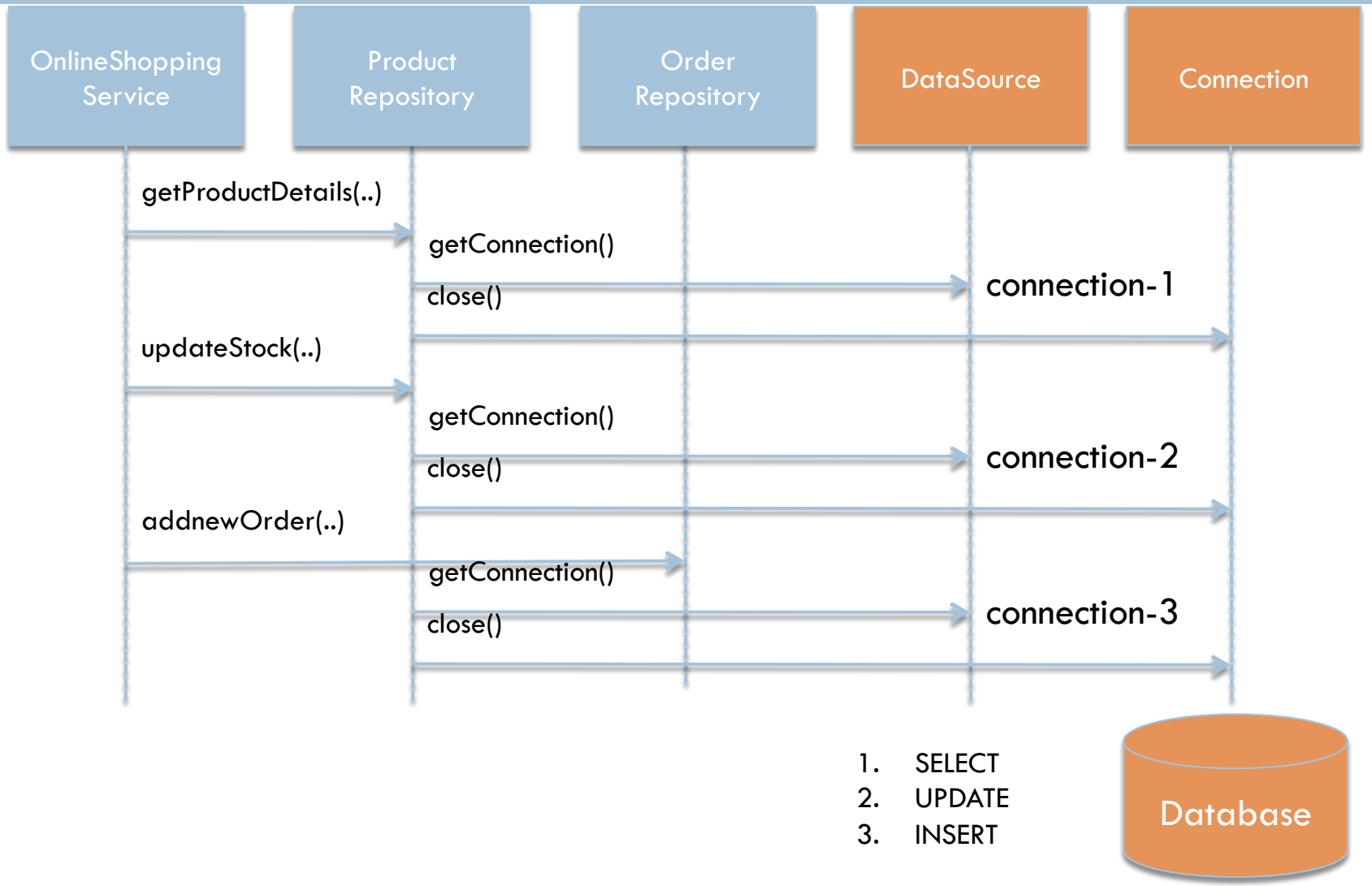
Why use Transactions?

- To enforce ACID principles:
  - Atomic
    - Each unit of work is an all-or-nothing operation
  - Consistent
    - Database integrity constraints are never violated
  - Isolated
    - Isolating transactions from each other
  - Durable
    - Committed changes are permanent

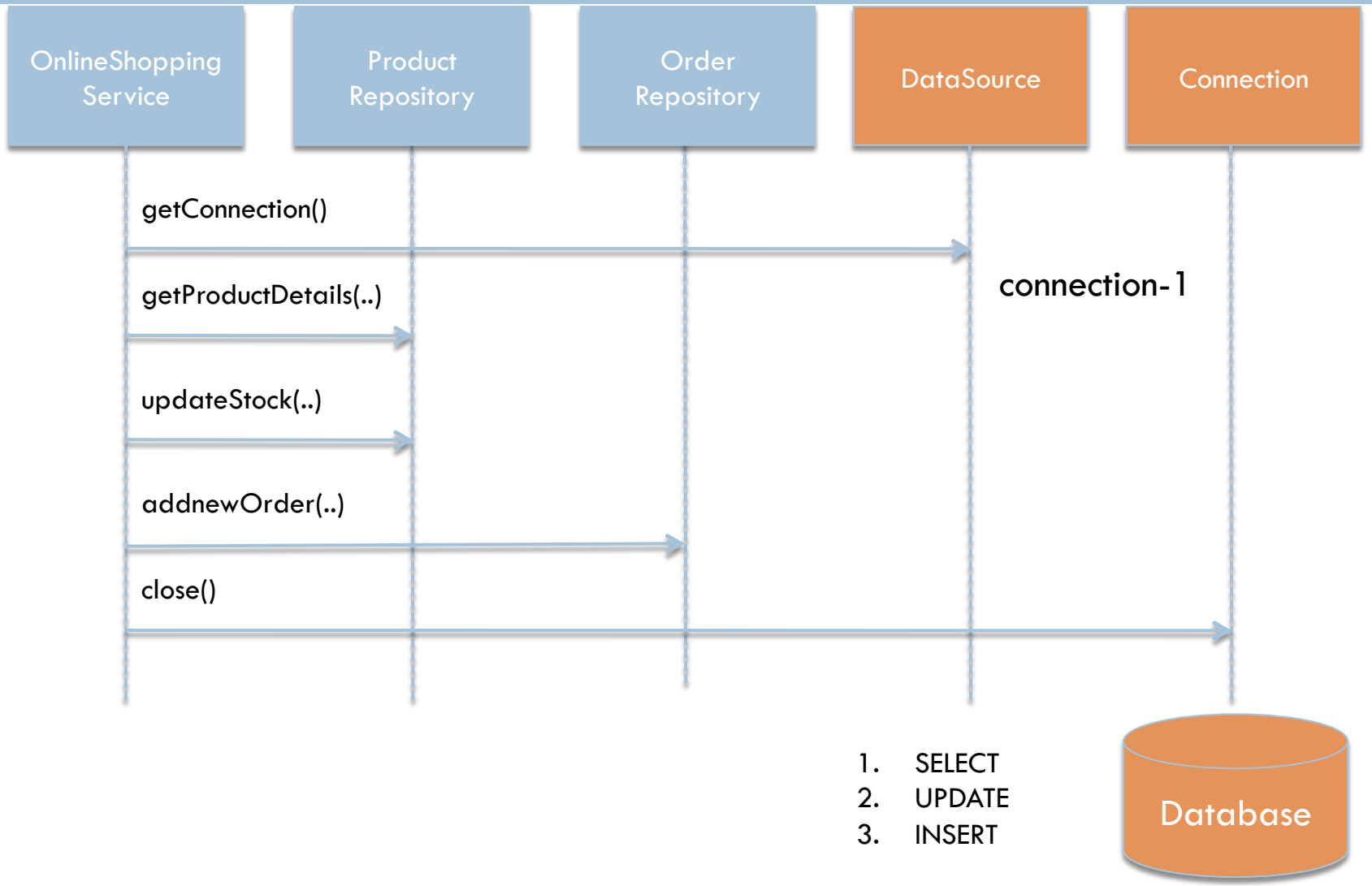
Example


placeOrder(..)
OnlineShopping Service
Product Repository
Oder Repository
getProductDetails(..)
updateStock(..)
addnewOrder(..)
1. SELECT
2. UPDATE
3. INSERT
Database

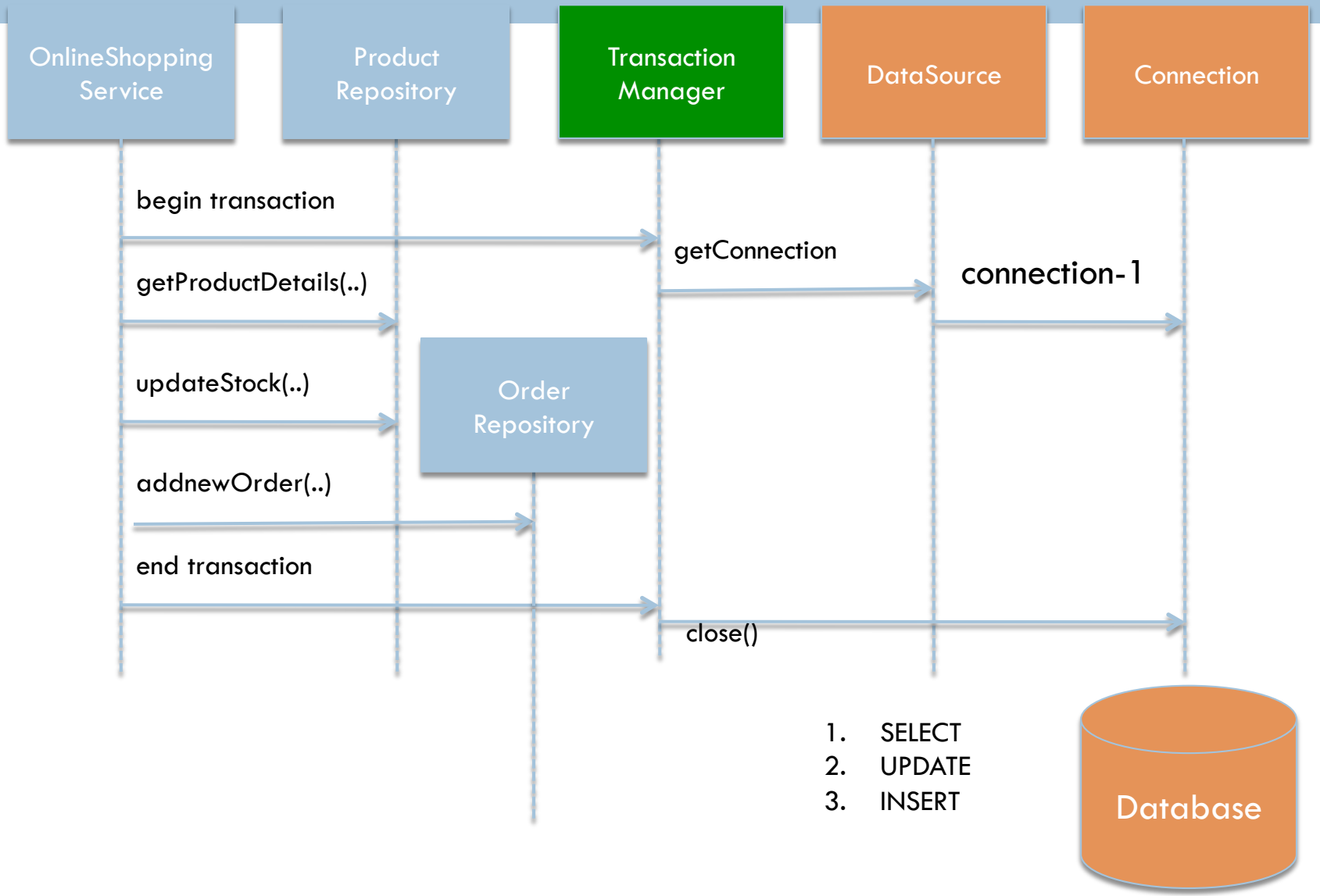
Running without any transaction



Running in a transaction



Running in a transaction



The risks of not using transactions

- Efficient connection management is important
- Proper transaction management is even more important
  - Using a Transaction API to dealing with this
- The problem is that the API for managing local transaction management is part of JDBC itself whereas the API for distributed transaction management is JTA
- It might happen that initially we are using a single database and using JDBC for managing transactions and in future we might want to distribute the data across multiple databases and this would mean a good amount of change in transaction management code

Local Transaction management

- Transactions can be managed at the level of the local resource
  - Database
- Cannot be used if multiple resources are paticipating in a transaction
- The *java.sql.Connection* interface itself contains methods for managing local transactions:
  - setAutoCommit(boolean), commit(), rollback(), etc…

Distributed Transaction management

- JTA is used if we need to manage transactions across multiple resources
- Application Servers provide full support for 2PC/ XA transactions
  - We need to use UserTransaction interface and it's methods like begin(), commit() and rollback()

PlatformTransactionManager

- *PlatformTransactionManager* is the base interface for the abstraction
- Several implementations are available
  - DataSourceTransactionManager
  - HibernateTransactionManager
  - JtaTransactionManager
  - WeblogicJtaTransactionManager
  - WebspeheUowTransactionManager
  - …

Configuring the transaction manager

- Pick the specific implementation

```
<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
</bean>
```

- OR

```
<bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

- For JTA,

```
<tx:jta-transaction-manager />
```

Declarative transactions

- When using declarative transaction management, we don't write the code to begin/commit/rollback again and again
  - We just need to declare that a service method represents a logical, atomic unit-of-work
  - Container/Framework manages transactions based on the metadata provided
  - Spring and EJB both support declarative transaction management
- No need for transaction management code getting cluttered along with business logic
- If we manage transactions on our own, it can be referred to as Bean Managed Transaction

Bean managed transactions

- Declarative transactions is highly recommended
- If you wish to programatically deal with transactions, you can always inject the *TransactionManager* like any other bean. Also Spring provides *TransactionTemplate* class for programatically managing the same

Declarative transaction example


```
public class OnlineShoppingServiceImpl implements OnlineShoppingService {

        @Transactional
        public void placeOrder(Order order) {
                //atomic unit of work
        }
}
```

When using annotations for tx management, we need to inform the container about the same by adding **<tx:annotation-driven />** in the xml file.

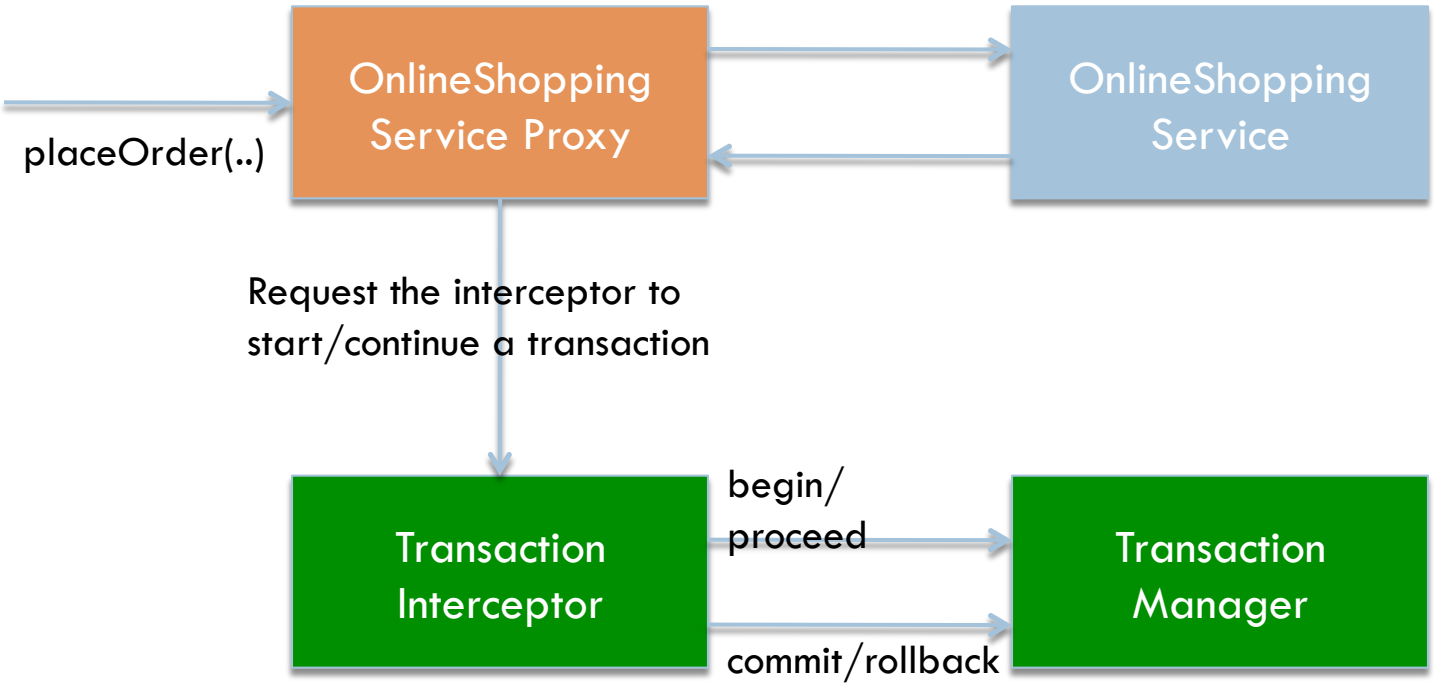
@Transactional annotation can be used at class/interface level.

```
public interface OnlineShoppingService {

        @Transactional
        public void placeOrder(Order order);
}
```

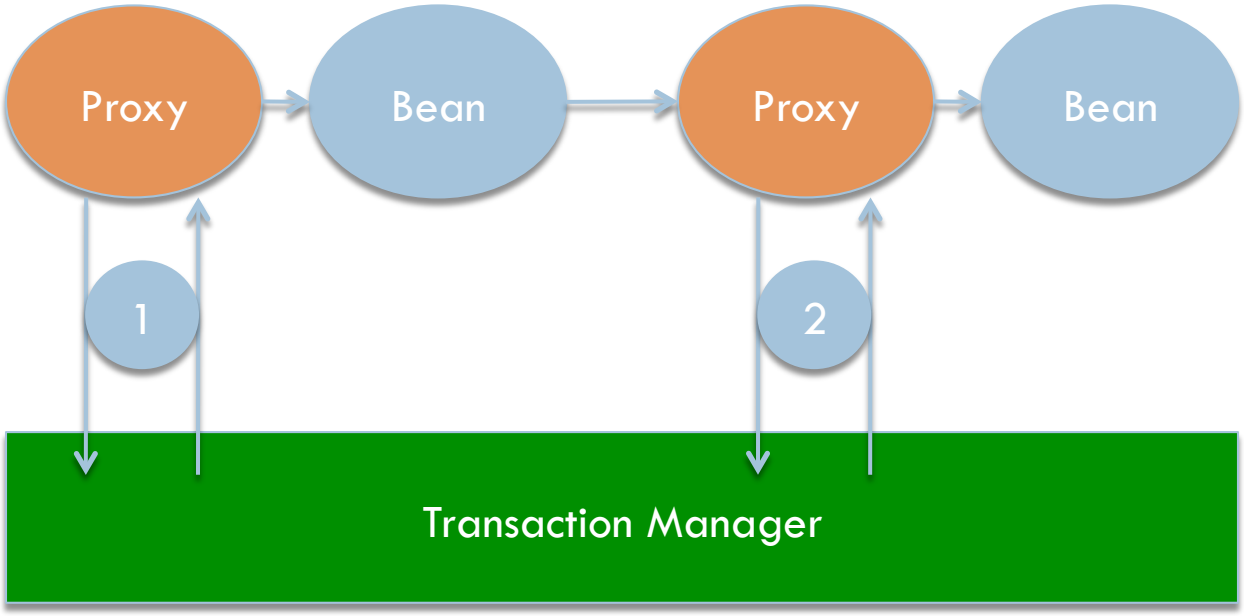
But who reads those annotations?

- Spring uses AOP proxies for beans requiring transaction support by default
- A **proxy** class would be generated for OnlineShoppingService bean which will use an interceptor called as **TransactionInterceptor** to communicate with the transaction manager and based on the configuration will propagate the transaction accordingly
- No problem if LTW has to be used, it's a minor change in the configuration required

Cont'd…


placeOrder(..)
OnlineShopping Service Proxy
OnlineShopping Service
Request the interceptor to start/continue a transaction
Transaction Interceptor
begin/ proceed
commit/rollback
Transaction Manager

Transaction propagation



1. The Transactional proxy requests the tx manager to start a transaction. Since for the current thread, no transaction has been started, the tx manager begins a new transaction.
2. The proxy requests the tx manager to start a transaction. Since already a transaction has been started for the current thread, the transaction manager decides to propagate the same transactional context. If the bean method fails, the proxy informs about the failure to the tx manager and the tx manager decides to rollback the transaction.

@Transactional attributes

@Transactional(isolation=Isolation.READ_COMMITTED)

Informs the tx manager that the following isolation level shoud be used for the current tx. Should be set at the point from where the tx starts because we cannot change the isolation level after starting a tx.

@Transactional(timeout=60)

Informs the tx manager about the time duration to wait for an idle tx before a decision is taken to rollback non-responsive transactions.

@Transactional(propagation=Propagation.REQUIRED)

If not specified, the default propagational behavior is REQUIRED. Other options are REQUIRES_NEW, MANDATORY, SUPPORTS, NOT_SUPPORTED, NEVER and NESTED.

REQUIRED

- Indicates that the target method can not run without an active tx. If a tx has already been started before the invocation of this method, then it will continue in the same tx or a new tx would begin as soon as this method is called.
- Methods which can be atomic as well as sub-atomic in nature are the best candidates for this option
  - withdraw() method can be a separate tx on its own as well as can be a part of a tx in case of transfer()

REQUIRES_NEW

- Indicates that a new tx has to start everytime the target method is called. If already a tx is going on, it will be suspended before starting a new one.
- Methods which are atomic in nature and cannot be sub-atomic unit of work are the best candidates for this option.
  - transfer(), placeOrder() cannot participate in an ongoing tx. It should always run in it's own tx

MANDATORY

- Indicates that the target method requires an active tx to be running. If a tx is not going on, it will fail by throwing an exception.
- Methods which are purely sub-atomic in nature and cannot be atomic unit or work are the best candidates for this option.
  - addPaymentRecord() cannot be an atomic unit of work. It's always a part of an existing transaction like billPayment(), placeOrder(), etc…

SUPPORTS

- Indicates that the target method can execute irrespective of a tx. If a tx is running, it will participate in the same tx
- If executed without a tx it will still execute if no errors
- Methods which fetch data are the best candidates for this option
  - getProductDetails() can utilize locking capabilities if running in a tx, or else it will anyway fetch the data based on the isolation level set.

NOT_SUPPORTED

- Indicates that the target method doesn't requires the transaction context to be propagated
- Mostly those methods which run in a transaction but perform in-memory operations are the best candidates for this option.

NEVER

- Indicates that the target method will raise an exception is executed in a transactional process
- This option is mostly not used in projects

NESTED

- Indicates that the target method should create a Savepoint in the ongoing transaction so that if the target method fails, instead of rolling back the whole tx, only rollback till the savepoint created
- createBackup() method can be good candidate for this option, as we don't wan't to cancel the order just because we were not able to create a backup of the order

@Transactional attributes

@Transactional(rollbackFor=ShoppingCartException.class)

Similar to EJB in Spring, to rollback a transaction we need to raise a runtime exception by default. In Spring, all API classes throw RuntimeException, which means if any method fails, the container will always rollback the ongoing transaction. The problem is only with checked exceptions. So this option can be used to declaratively rollback a transaction if ShoppingCartException occurs.

@Transactional(noRollbackFor=IllegalStateException.class)

Indicates that a rollback should not be issued if the target method raises this exception. So what will happen is all SQL operations performed prior to this exception getting raised, would get committed in the database.

Transaction management

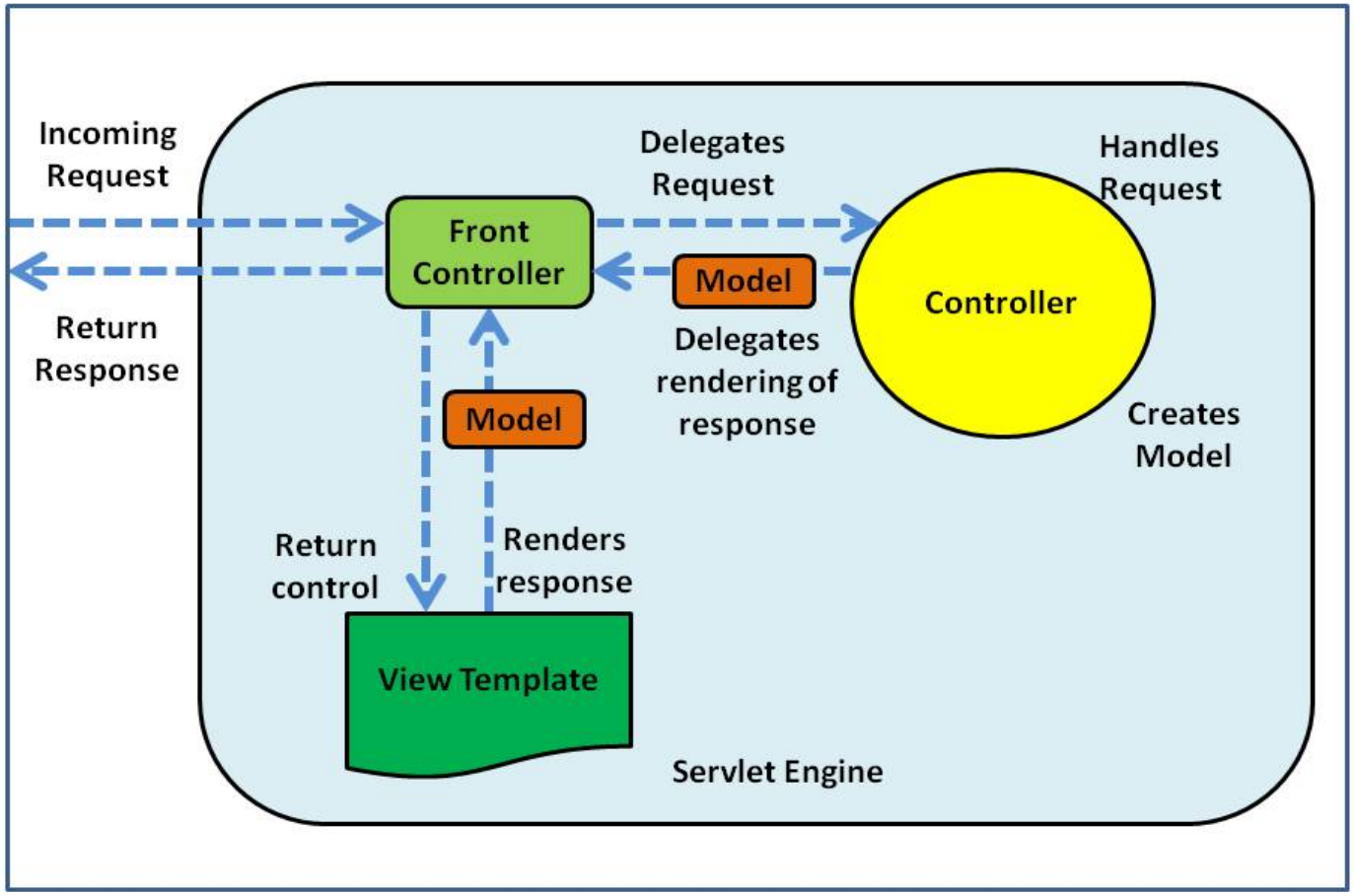
Lab No. 8

Refer to the lab guide provided along with the eclipse project to proceed further

Topics to be covered

- Spring MVC
  - Understanding support for developing Web applications
  - Understanding terms like Controllers, Handlers & View Resolvers
  - Understanding support for RESTful WebServices
    - Support for JSON & XML

Introduction



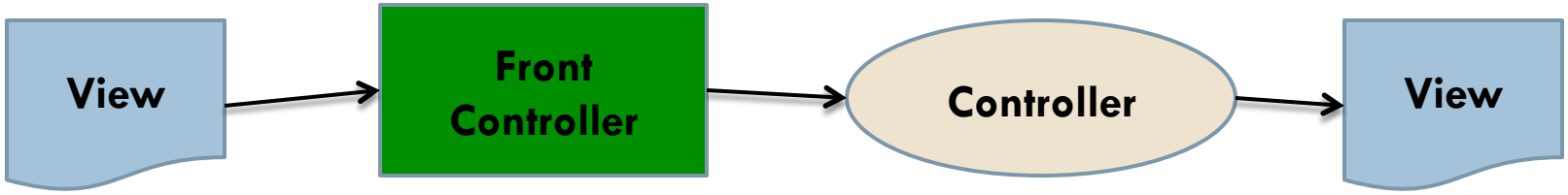
Spring MVC is like any other MVC implementation from outside but more fine-grained and extensible from within

Controllers

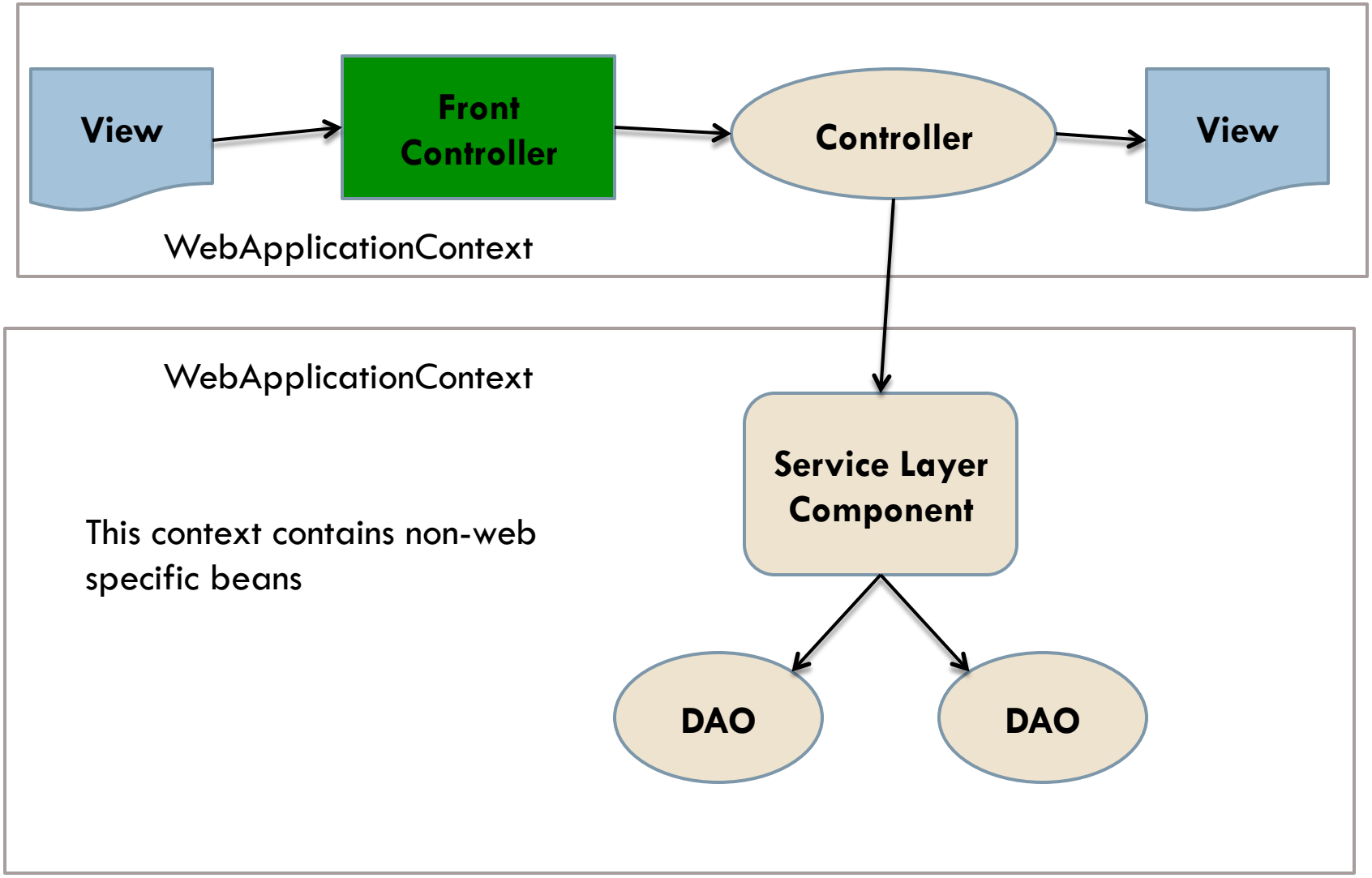
- Spring MVC delegates the responsibility for handling HTTP requests to **Controllers**. Controllers are much like servlets, mapped to one or more URIs and built to work with HttpServletRequest and HttpServletResponse objects.
- The Controller API makes no attempt to hide its dependence to the Servlet API, instead fully embracing it and exposing its power.

Cont'd…

- Controllers are responsible for processing HTTP requests, performing whatever work necessary, composing the response objects, and passing control back to the main request handling work flow. The Controller does not handle view rendering, focusing instead on handling the request and response objects and delegating to the service layer.

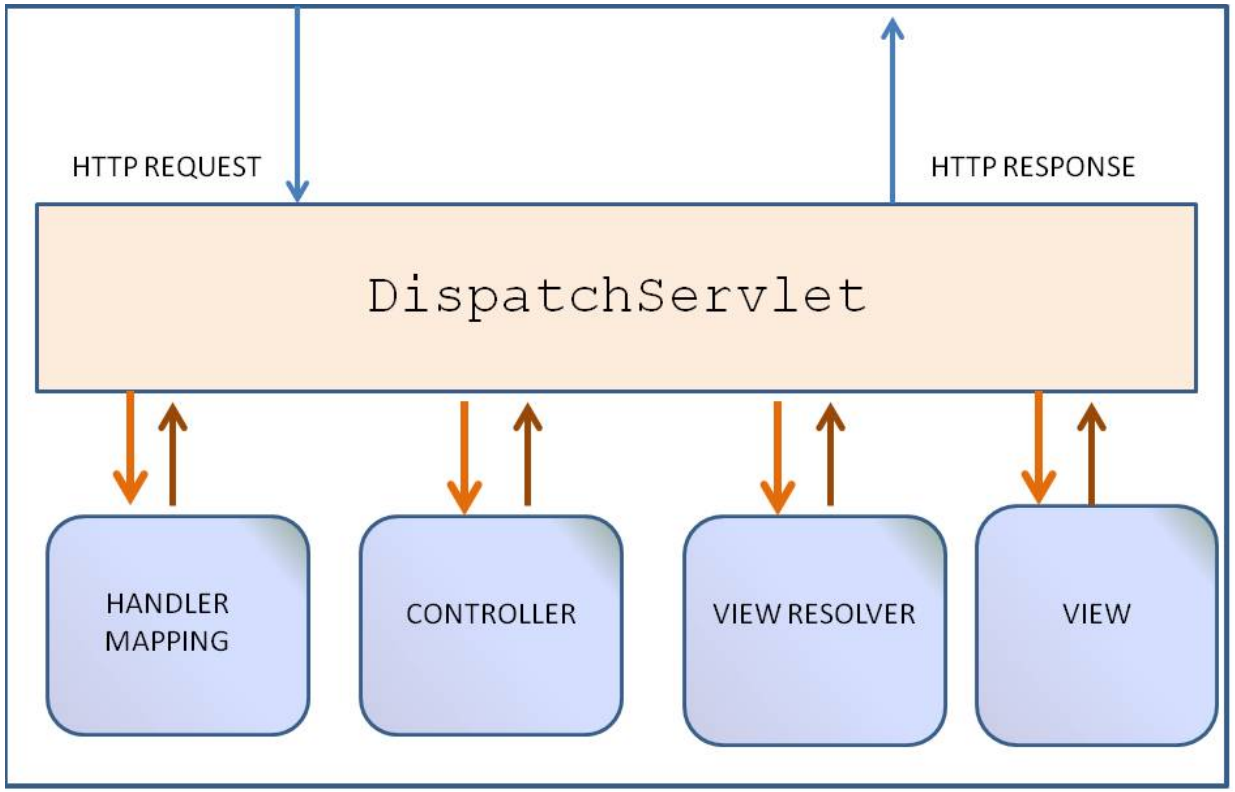


Cont'd…


View
Front Controller
Controller
View
WebApplicationContext
WebApplicationContext
This context contains non-web specific beans
Service Layer Component
DAO
DAO

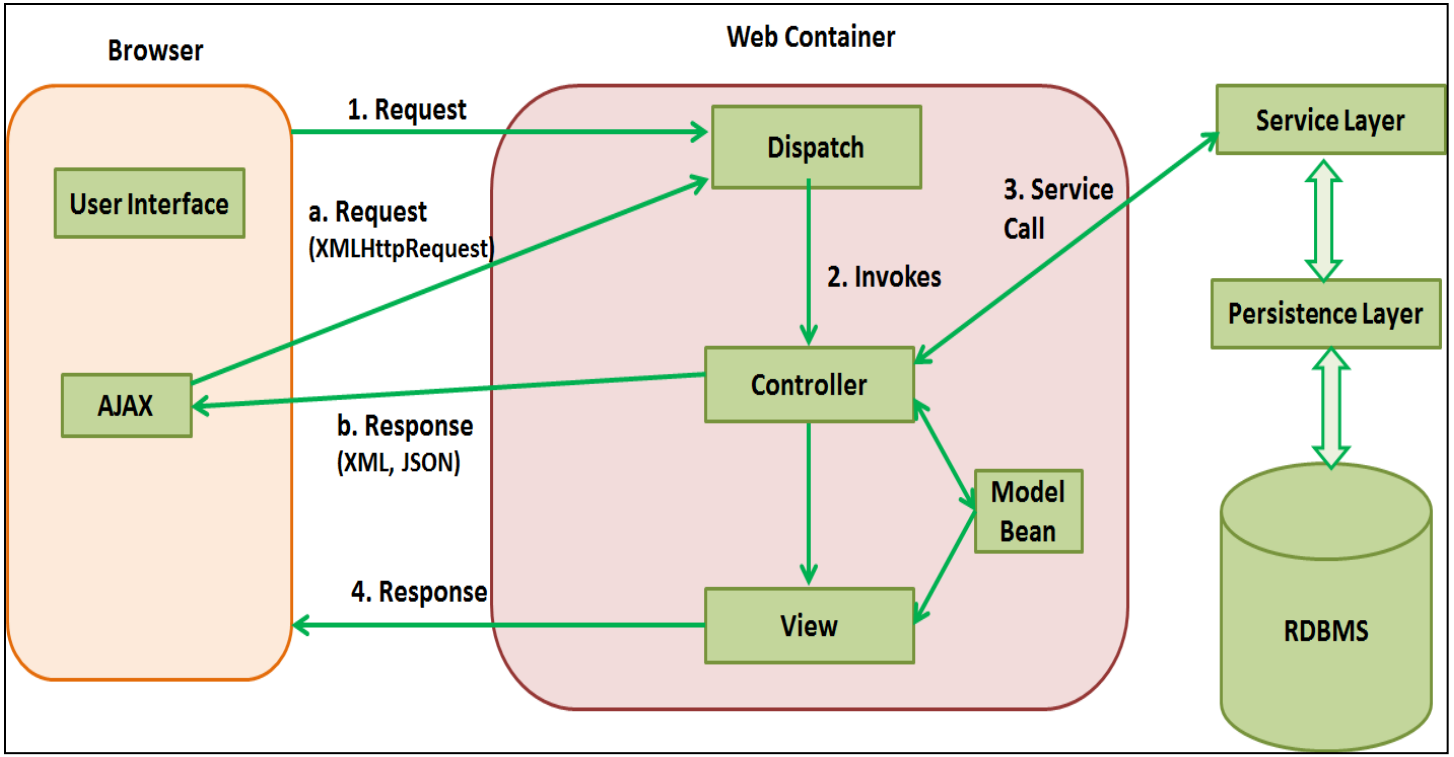
Cont'd…

- The role of the Front Controller in Spring MVC is played by DispatcherServlet class



2 in 1

- Spring MVC can be used for developing traditional web applications as well as modern AJAX and RESTful applications as well



Configuration

```
<listener>
      <listener-class>
            o.s.web.context.ContextLoaderListener
      </listener-class>
</listener>
```

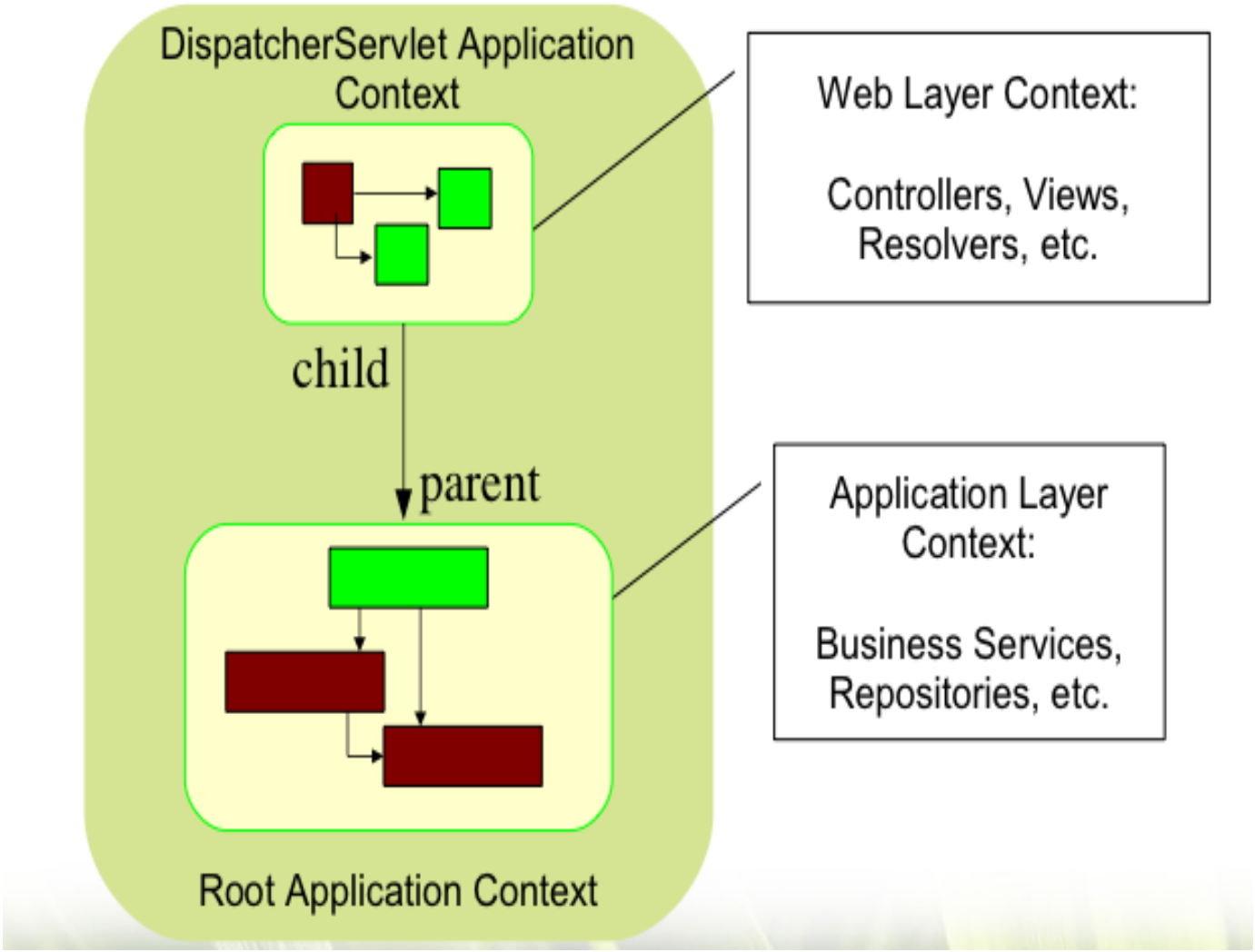
ContextLoaderListener provided by Spring will create a WebApplicationContext instance by reading applicationContext.xml file from the WEB-INF folder by default

Cont'd…

```
<servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
                o.s.web.servlet.DispatcherServlet
        </servlet-class>
</servlet>

<servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/controller/*</url-pattern>
        <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

DispatcherServlet is the FrontController in a Spring MVC application. DispatcherServlet will handle incoming request mapped to the given url. By default will read the <servlet-name>-servlet.xml file from WEB-INF folder.

Separate contexts



Views

- The response from a Controller is rendered for output by an instance of the View class. The Controller does not perform any view rendering, nor is it aware of the view rendering technology used.
- Spring MVC has excellent support for many different view rendering technologies, including template languages such
  - JSP and JSTL
  - Velocity & FreeMarker
- Other bundled view rendering toolkits include
  - PDF, Excel
  - JasperReports

ModelAndView

- When processing is complete, the Controller is responsible for building up the collection of objects that make up the response (the Model) as well as choosing what page (or View) the user sees next. This combination of Model and View is encapsulated in a class named ModelAndView
- In the recent versions of Spring MVC, **java.util.Map** can be used to represent the Model and a **String** can be used to represent the name of View making programming further more simple

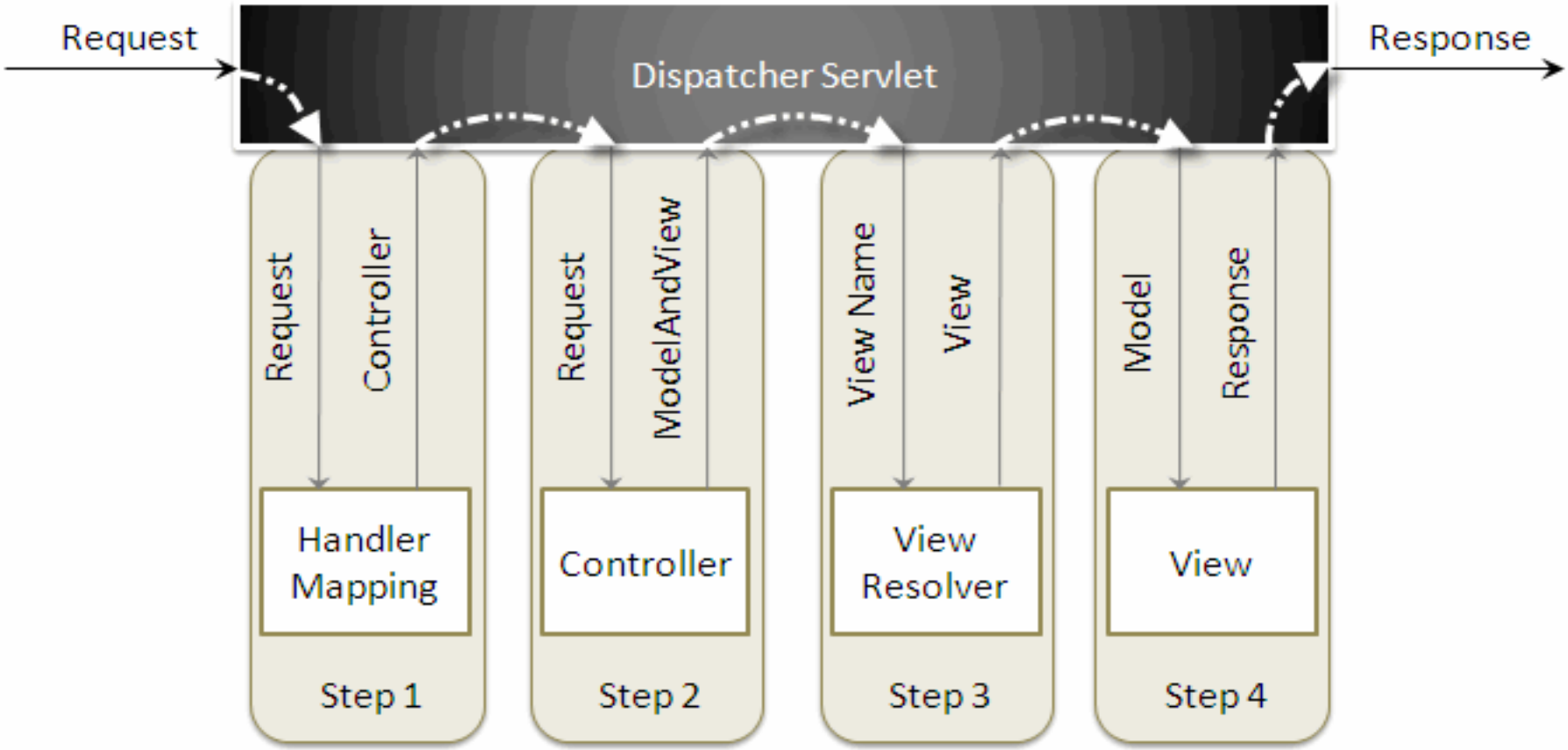
Overview

- **Views** are single pages, usually created with a template language such as JSP, Velocity, or FreeMarker. Spring MVC also supports such technologies as PDF, Excel, and JasperReports.
- View systems can be mixed and matched in an application.

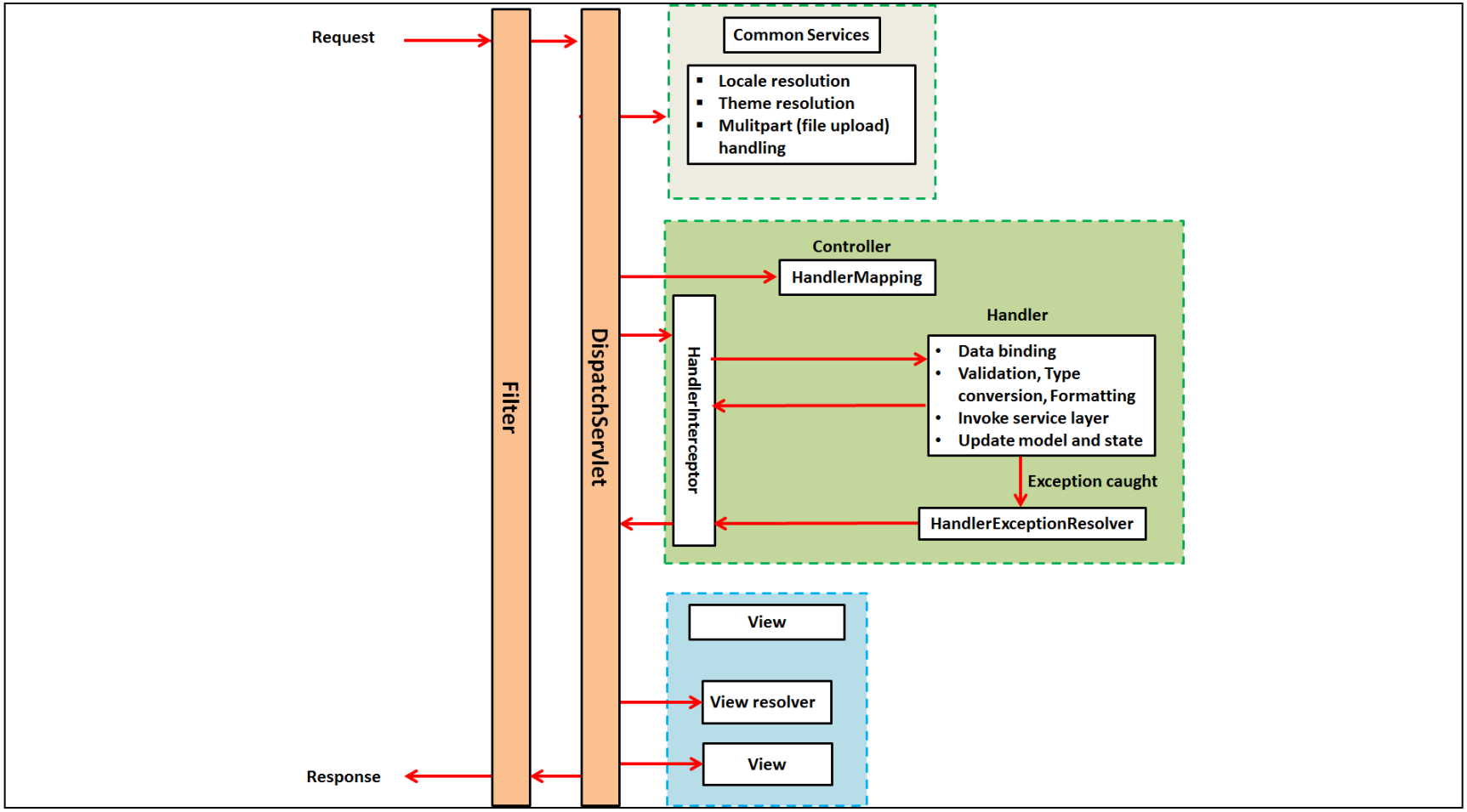
Overview

- **ViewResolvers** translate a logical view name into a physical View instance. This class is used to keep the Controllers blissfully unaware of the actual view technology in use.
- ViewResolvers can be chained together if multiple view strategies are required.

Flow


Request
Dispatcher Servlet
Response
Request
Controller
Handler Mapping
Step 1
Request
ModelAndView
Controller
Step 2
View Name
View
View Resolver
Step 3
Model
Response
View
Step 4

Cont'd…


Request
Filter
DispatchServlet
Common Services
Locale resolution
Theme resolution
Mulitpart (file upload) handling
Controller
HandlerMapping
HandlerInterceptor
Handler
Data binding
Validation, Type conversion, Formatting
Invoke service layer
Update model and state
Exception caught
HandlerExceptionResolver
View
View resolver
View
Response

Spring MVC

Lab No. 9

Demo examples, no lab guide as of now!

Topics to be covered

- Introducing Spring Security previously called as Acegi Security
- Why Spring security?
- Authentication and Authorization support

Security concepts

- Principal
  - User, device or system that performs an action
- Authentication
  - Establishing that a principal's credentials are valid
- Authorization
  - Deciding if a principal is allowed to perform an action
- Secured resource
  - Only accessible after successful authentication/ authorization

Authentication

- There are many authentication mechanisms
  - basic, digest, form, X.509
- There are many storage options for credential and authority information
  - Database, LDAP, properties files,…

Authorization

- Authorization depends on authentication
  - Before deciding if a user can perform an action, user identity must be established
- The decision process is often based on roles
  - ADMIN can can cancel orders
  - MEMBER can place orders
  - GUEST can browse the catalog

Why Spring for Security?

- Servlet-Spec security is not portable
  - Requires container specific adapters and role mappings
- Spring Security is portable across containers
  - Secured archive (e.g. WAR) can be deployed as-is
  - Also runs in standalone environments

Cont'd…

- Supports all common authentication mechanisms
  - Basic, Form, X.509, Cookies, Single-Sign-On, etc.
- Provides configurable storage options for user details (credentials and authorities)
  - RDBMS, LDAP, Properties file, custom DAOs, etc.
- Uses Spring for configuration

Cont'd…

- Security requirements often require customization
- With Spring Security, all of the following are extensible
  - How a principal is defined
  - Where authentication information is stored
  - How authorization decisions are made
  - Where security constraints are stored

How it works?

- Spring provides a Servlet filter for managing security for our web applications
- springSecurityFilterChain is declared in web.xml
- This single proxy filter delegates to a chain of Spring-managed filters
  - Drive authentication
  - Enforce authorization
  - Manage logout
  - Maintain SecurityContext in HttpSession
  - *and more*

Cont'd…

A user requests access to a secured page through a web application.

The AuthenticationManager configured into the application checks the users credentials.

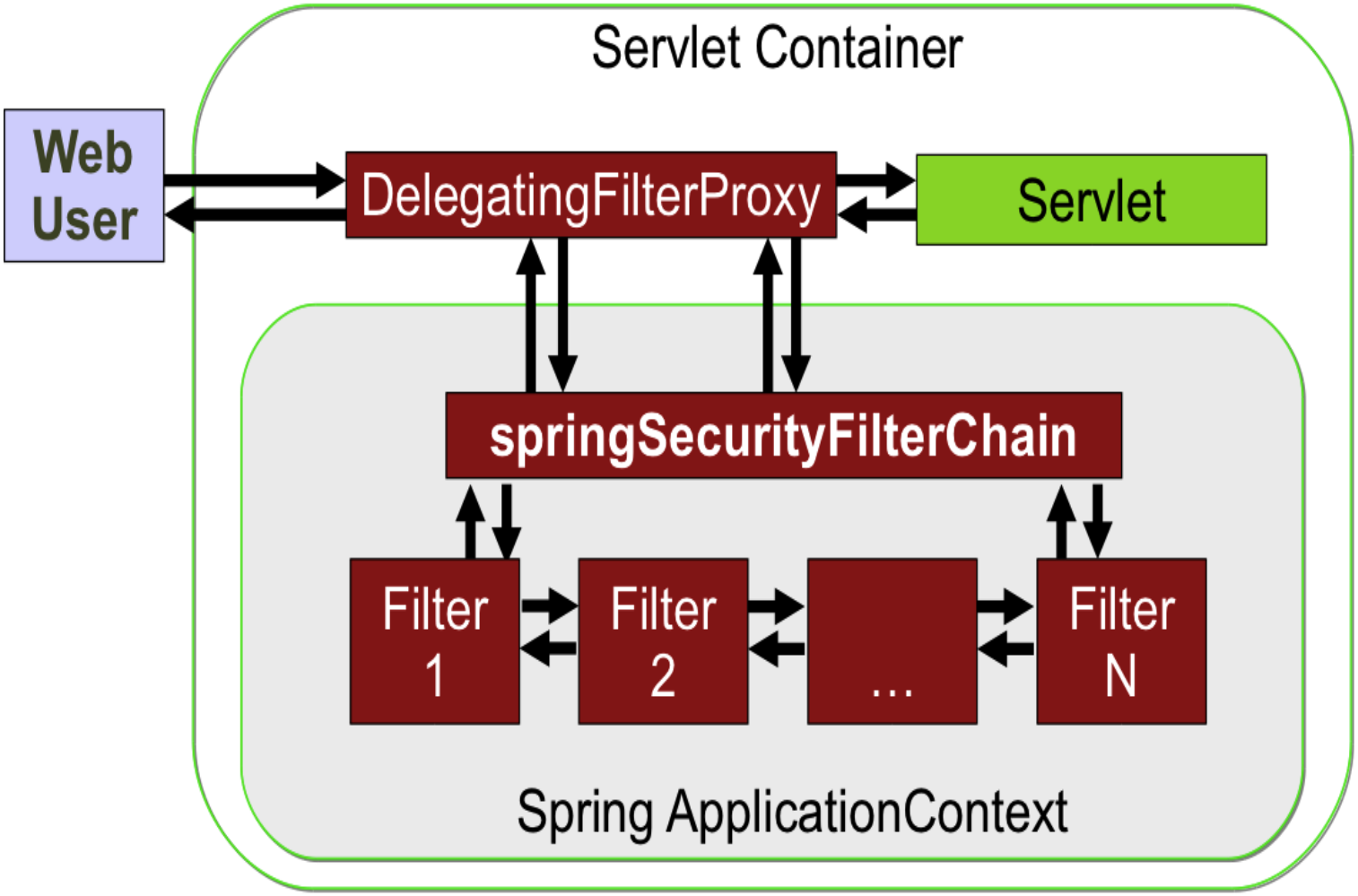
User provides necessary credentials such as a login ID and a password.

If the validation of the credential fails, access to the secured page is refused.

If the credentials are validated successfully, the AccessDecisionManager ensures that the user has the proper authority to access the secured page.

If the authorization fails, then access to the secured page is denied. If authorization succeeds, the secured page is displayed to the user.

Web Security Filter Configuration



Filter Chains

- Filters are components that translates user requests and responses, to a machine compatible form. When a user Request is initiated, filters process the request by performing the following steps:

Cont'd…

When a user submits a request through a web application, Spring Framework loads its SecurityContext.

If the user request is to log out of a particular URL, the user is logged out.

If the user requests an authentication form submission, then authentication is attempted and a default login page is displayed.

Filters check whether the request has an authorization header

If yes, user name and password is extracted for authentication. If authentication is successful, it is registered in the SecurityContext

Cont'd…

The user request is then wrapped with the SecurityContext into a single object.

If authentication fails, then the object is flagged as anonymous.

If the user authentication is successful, session authentication strategy is applied.

If the user has not been authenticated successfully, he is flagged as anonymous.

Any AccessDeniedException and AuthenticationException thrown by any of the above are handled now.

Authorization and access control decisions are delegated to an AccessDecisionManager interface.

Authorization and access control decisions are delegated to an AccessDecisionManager interface.

Spring Security

Lab No. 10

Demo examples, no lab guide as of now!

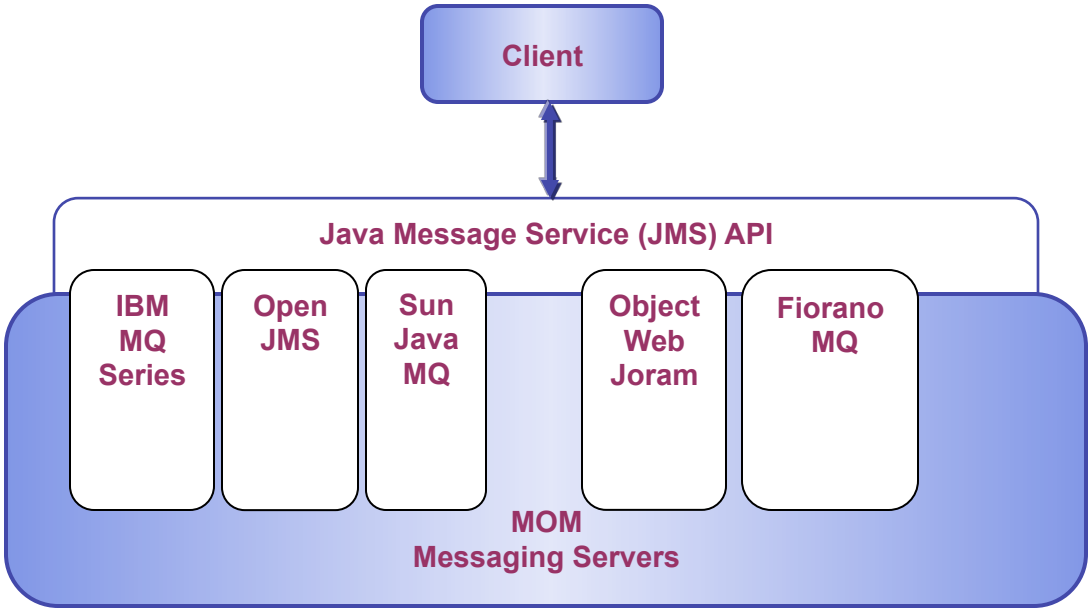
Topics to be covered

- Spring JMS Integration
- Understanding the need for JMS
- Common problem when using JMS API
- Introducing JMSTemplate class
- Using POJOs as MessageListener
- Annotation support

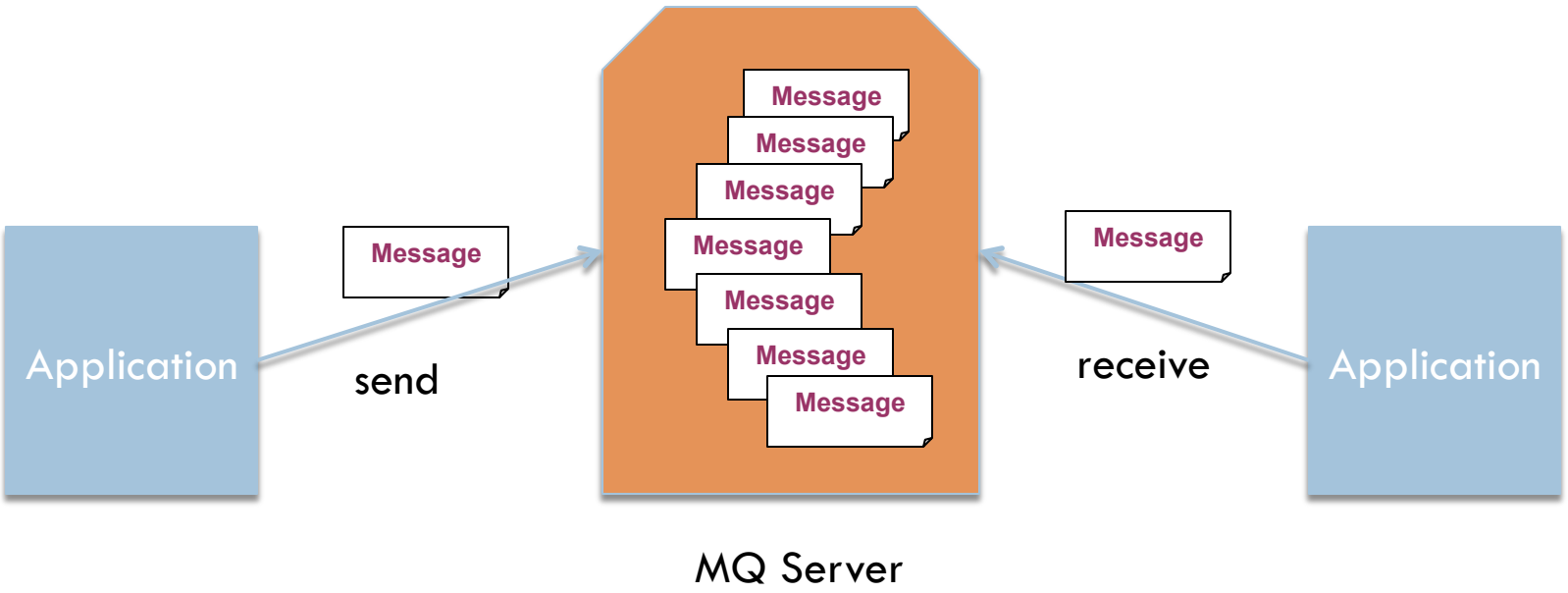
About JMS

- JMS (Java Messaging Service) is an API for managing communication between the application and an MQ (Message Queue) server, also called as MOM (Message Oriented Middleware)
- Messaging is used in B2B applications for asynchronous communication. So if the business process needs some time to execute and we don't want to wait for the time it takes to complete the task, we can use JMS
- Also MOM play a very important role in Enterprise integration with legacy systems, so there are lot's of reasons why they are still popular
- Communication between the application and the MQ is done in the form of Message objects. So whatever data we need to send/ receive will be encapsulated within these Message objects

Cont'd…


Client
Java Message Service (JMS) API
IBM MQ Series
Open JMS
Sun Java MQ
Object Web Joram
Fiorano MQ
MOM
Messaging Servers

Introduction



Messaging Domains

- Point-to-point

  Message queues, senders, receivers.

- Publish/Subscribe

  Message topics, publishers, subscribers

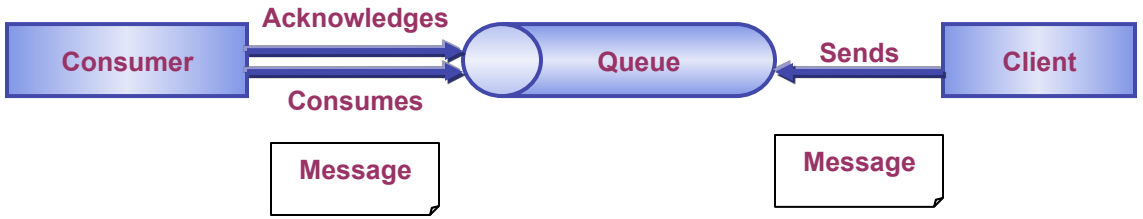
  Durable subscriptions

Point-to-Point

- Each message is delivered to a specific queue. Thus each message has only one consumer
- Queues retain all messages until consumed or expired
- Receiving clients extract the messages from the message queue(s) established to hold their messages. No timing dependencies. Client on its own wish can come alive and extract the message
- Receiver acknowledges the successful processing of a message

Overview

- Client/Sender sends a message to the Queue
- MQ stores the message in the Queue
- Client/Consumer receives the message and acknowledges the MQ
- After successful acknowledgement MQ removes the message from the Queue

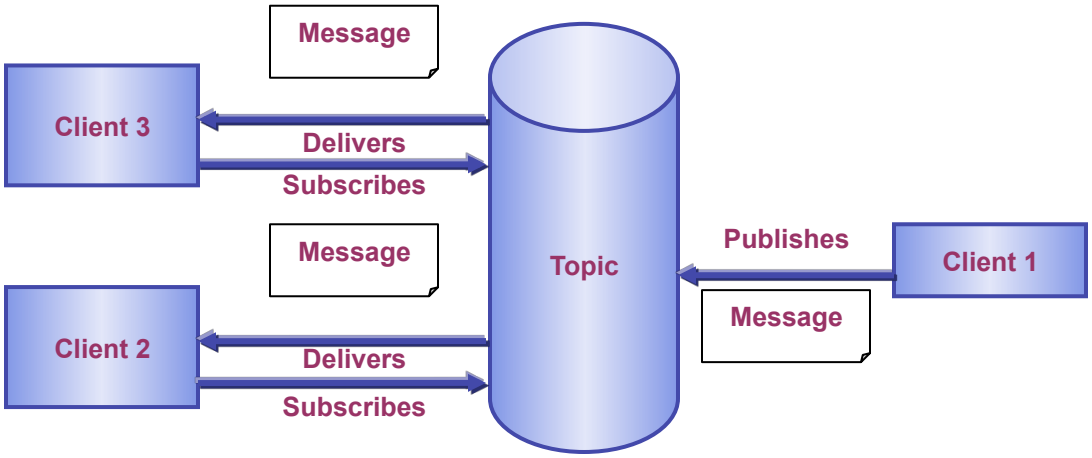


Publish/Subscribe

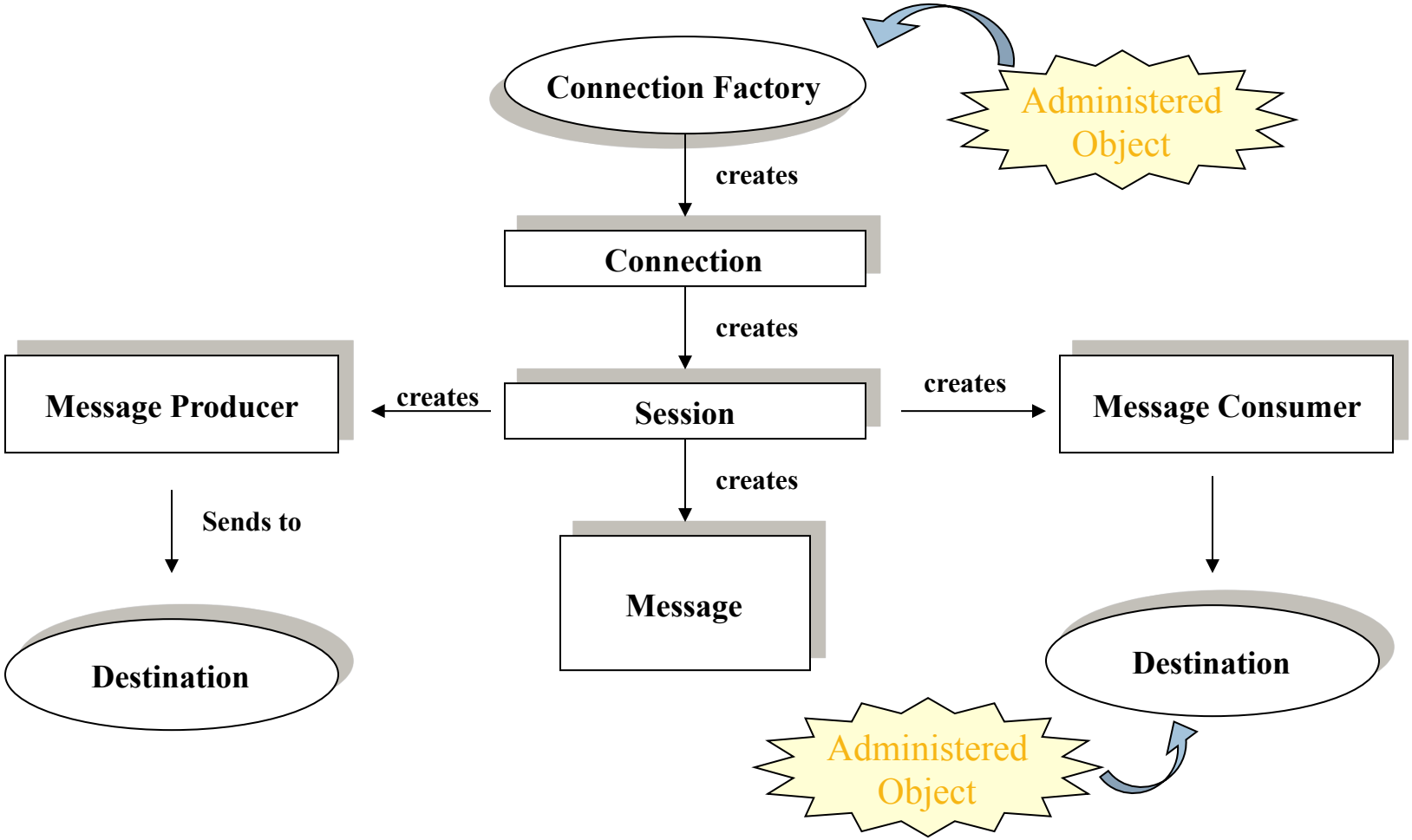
- Messages are bound for topics
- Provider has the responsibility to distribute the message to its multiple subscribers. Each message can have multiple consumers
- Topics retain messages only as long as it takes to distribute them to current subscribers
- There is a timing dependency. If the subscriber isn't alive at the moment the message arrived into the topic, the message is liable to be lost

Overview

- Sender/Publisher sends a message to a topic on the server
- All active Receiver/Consumer receives a copy of the message from the MQ



JMS programming model



The problem

- The problem when using the JMS API is that there are so many steps to perform even for simple communication, very much similar to the boilterplate code we write when using JDBC
- Spring provides a convenient **JMSTemplate** class to minimizes this overhead

Example


```
@Service
public class CreditCardProcessorImpl implements CreditCardProcessor {

        @Autowired
        private JmsTemplate jt;

        public void processCreditCardRequest(CreditCard ccInfo) {
                jt.convertAndSend("MyQueue", ccInfo);
                …
        }
}
```

For consuming

- Spring framework provides full support for a regular *MessageListener* interface implementation to act as a consumer
- One of the interesting features of Spring is support for plain POJOs acting as Message consumers

Example


```
@Service
public class AsyncCreditCardApprover implements MessageListener {

        public void onMessage(Message msg) {
                System.out.println("Inside AsyncCreditCardApprover..");
                ObjectMessage objMsg = (ObjectMessage) msg;
                CreditCard ccInfo = (CreditCard) objMsg.getObject();
                …
        }
}
```

As you can see below, a plain POJO class can act as a MessageListener without knowing anything about JMS at all. POJO power! :-)

```
@Service
public class AsyncPOJOCreditCardApprover {

        public void approve(CreditCard ccInfo) {
                System.out.println("Inside AsyncPOJOCreditCardApprover..");
                …
        }
}
```

Cont'd…

- Spring 4 introduces additional annotations to minimize the configuration required for Message consumers. For example:

```
@Service
public class AsyncPOJOCreditCardApprover2 {

        @JmsListener(destination = "MyQueue")
        public void approve(CreditCard ccInfo) {
                System.out.println("Inside AsyncPOJOCreditCardApprover2..");
```

Spring JMS

Lab No. 11

Demo examples, no lab guide as of now!

Conclusion

Time to say Goodbye!

Hope you all had a great learning experience!