

# Spring Framework

# About Me

- Maruthi R Janardhan
  - Been doing java since jdk 1.2
  - Been with IBM, ANZ, HCL-HP, my own startup Leviossa..
  - Total 16 years programming C, C++, Java, Ruby, Perl, Python, PHP, etc

# UserManager v1

```
public class UserManager {  
  
    private UserDao dao = new UserDao();  
  
    public List<User> getAllUsers(){  
        return dao.getAllUsers();  
    }  
  
}  
  
public class UserDao {  
  
    public List<User> getAllUsers(){  
        //Do database queries here  
        return new ArrayList<User>();  
    }  
  
}
```

# UserManager v2

```
public class UserManager {  
  
    private UserDao dao = new UserDaoImpl();  
  
    public List<User> getAllUsers(){  
        return dao.getAllUsers();  
    }  
  
}  
  
public interface UserDao {  
  
    public List<User> getAllUsers();  
  
}  
  
public class UserDaoImpl implements UserDao {  
    public List<User> getAllUsers() {  
        // Do database queries here  
        return new ArrayList<User>();  
    }  
  
}
```

# UserManager v3

```
public class UserManager {  
  
    private UserDao dao = UserDaoFactory.getDao();  
  
    public List<User> getAllUsers(){  
        return dao.getAllUsers();  
    }  
  
}  
public class UserDaoFactory {  
    public static UserDao getDao(){  
        return new UserDaoImpl();  
    }  
}  
  
UserManager mgr = new UserManager();  
List<User> users = mgr.getAllUsers();
```

# UserManager v4

```
public class UserManager {  
  
    private UserDao dao;  
  
    public UserDao getDao() {  
        return dao;  
    }  
    public void setDao(UserDao dao) {  
        this.dao = dao;  
    }  
    public List<User> getAllUsers() {  
        return dao.getAllUsers();  
    }  
}
```

```
UserManager mgr = new UserManager();  
mgr.setDao(new UserDaoImpl());  
List<User> users = mgr.getAllUsers();
```

# UserManager v5 (Spring)

```
<bean id="userManager" class="com.mydomain.biz.UserManager">  
    <property name="dao" ref="userDao"></property>  
</bean>  
<bean id="userDao" class="com.mydomain.dao.UserDaoImpl">  
</bean>
```

```
ApplicationContext context = new  
ClassPathXmlApplicationContext(new String[] {"application-  
context.xml"});  
userManager mgr = (userManager)context.getBean("userManager");  
List<User> users = mgr.getAllUsers();
```

# UserManager v6 & v7

## Using static factory

```
<bean id="userManager" class="com.mydomain.biz.UserManager">  
    <property name="dao" ref="userDao"></property>  
</bean>  
<bean id="userDao" class="com.mydomain.dao.UserDaoFactory" factory-  
method="getDao">  
</bean>
```

## Using instance factory

```
<bean id="userDao" factory-bean="daoFactory" factory-method="getDao">  
</bean>  
<bean id="daoFactory" class="com.mydomain.dao.UserDaoFactory" factory-  
method="getDao">  
</bean>
```



# UserManager v8

```
<bean id="userManager" class="com.mydomain.biz.UserManager">  
    <constructor-arg ref="dao"></constructor-arg>  
</bean>  
<bean id="dao" class="com.mydomain.dao.UserDaoImpl">  
</bean>
```

```
public class UserManager {  
    private UserDao dao;  
  
    public UserManager(UserDao dao) {  
        this.dao = dao;  
    }  
  
    public List<User> getAllUsers() {  
        return dao.getAllUsers();  
    }  
}
```

# UserManager v9

```
<bean id="userManager" class="com.mydomain.biz.UserManager">  
  <property name="dao">  
    <bean class="com.mydomain.dao.UserDaoImpl">  
    </bean>  
  </property>  
</bean>
```

# UserManager v10

- Auto wiring byType and byName are possible

```
<bean id="userManager" class="com.mydomain.biz.UserManager"  
autowire="byType" >  
</bean>
```

```
<bean id="dao" class="com.mydomain.dao.UserDaoImpl" >  
</bean>
```

# Creating With Values

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
</bean>
```

-----  
xmlns:p="http://www.springframework.org/schema/p"

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="com.mysql.jdbc.Driver"
      p:url="jdbc:mysql://localhost:3306/mydb"/>
```

# Values for Collections

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <property name="adminEmails">
    <props>
      <prop key="administrator">administrator@example.org</prop>
      <prop key="support">support@example.org</prop>
    </props>
  </property>
  <property name="someList">
    <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource" />
    </list>
  </property>
  <property name="someMap">
    <map>
      <entry key="an entry" value="just some string"/>
      <entry key="a ref" value-ref="myDataSource"/>
    </map>
  </property>
  <property name="someSet">
    <set>
      <ref bean="someBean"/>
      <ref bean="myDataSource" />
    </set>
  </property>
</bean>
```

# Creation Order

- Spring can see dependencies and create beans in the order needed. If for some reason the dependency is not visible through property references, we can use depends-on

- This also controls destruction order

```
<bean id="userManager" class="com.mydomain.biz.UserManager" depends-on="myDataSource">
  <property name="dao">
    <bean class="com.mydomain.dao.UserDaoImpl">
      </bean>
    </property>
  </bean>
```

# Lazy Initialisation & Singletons

- By default all beans are singleton and are created at container start.
- scope of prototype causes a new bean to be created with each getBean call

```
<beans default-lazy-init="true" >
```

```
<bean id="userManager" class="com.mydomain.biz.UserManager" lazy-  
init="true" >
```

```
<bean id="userManager" class="com.mydomain.biz.UserManager"  
scope="prototype" >  
</bean>
```

# Proxy

```
public class ProxyImpl implements java.lang.reflect.InvocationHandler {

    private Object obj;

    public static Object newInstance(Object obj) {
        return java.lang.reflect.Proxy.newProxyInstance(obj.getClass()
            .getClassLoader(), obj.getClass().getInterfaces(),
            new ProxyImpl(obj));
    }

    private ProxyImpl(Object obj) {
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        Object result;
        try {
            //do something before
            result = m.invoke(obj, args);
            //do something after
        } catch (Exception e) {
            throw new RuntimeException("unexpected invocation exception: "
                + e.getMessage());
        }
        return result;
    }
}
```



# Lookup Methods

- When a singleton bean depends on a non-singleton bean, it can cause problems with threading

```
<bean id="userManager" class="com.mydomain.biz.UserManager" >  
  <lookup-method name="createUserDao" bean="dao" />  
</bean>
```

```
<bean id="dao" class="com.mydomain.dao.UserDaoImpl" scope="prototype">  
</bean>
```

```
public abstract class UserManager {  
  protected abstract UserDao createUserDao();  
  
  public List<User> getAllUsers() {  
    UserDao dao = createUserDao();  
    return dao.getAllUsers();  
  }  
}
```

# Bean Scopes

Scope	Description
<a href="#">singleton</a>	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
<a href="#">prototype</a>	Scopes a single bean definition to any number of object instances.
<a href="#">request</a>	New bean for each request
<a href="#">session</a>	New bean for each session
<a href="#">application</a>	Bean valid for the application life cycle (Web)

# BeanPostProcessors

- BeanPostProcessors act as extension points for the container. They are called for every bean that is instantiated in the container

```
public class MyBeanPostProcessor implements BeanPostProcessor {  
  
    public Object postProcessBeforeInitialization(Object bean,  
        String beanName) throws BeansException {  
        return bean; // we could potentially return any object reference here...  
    }  
  
    public Object postProcessAfterInitialization(Object bean,  
        String beanName) throws BeansException {  
        System.out.println("Bean '" + beanName + "' created : " +  
bean.toString());  
        return bean;  
    }  
  
}
```

# Property Placeholder

```
<bean  
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
<property name="locations" value="classpath:com/foo/jdbc.properties"/>  
</bean>
```

```
<bean id="dataSource" destroy-method="close"  
      class="org.apache.commons.dbcp.BasicDataSource">  
  <property name="driverClassName" value="${jdbc.driverClassName}"/>  
  <property name="url" value="${jdbc.url}"/>  
  <property name="username" value="${jdbc.username}"/>  
  <property name="password" value="${jdbc.password}"/>  
</bean>
```

# Annotation Based

```
<context:component-scan base-package="com.mydomain.*" />  
<context:annotation-config />
```

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class UserManager {
```

```
    @Autowired
```

```
    private UserDao dao;
```

```
@Component
```

```
public class UserDaoImpl implements UserDao {
```

```
    mgr = (UserManager)context.getBean(UserManager.class);
```

```
    mgr = (UserManager)context.getBean("userManager");
```

# Components That Are Scanned

- @Component – Indicates a auto scan component.
- @Repository – Indicates DAO component in the persistence layer.
- @Service – Indicates a Service component in the business layer.
- @Controller – Indicates a controller component in the presentation layer.

# Lets Try It

- Build an UserManager, UserManagerImpl, UserDao and UserDaoImpl backed with a real database and real method logic branching out of an UserManager class provided on share

# JSR 330 Annotations

@Named

```
public class UserDaoImpl implements UserDao {
```

```
import javax.inject.Inject;
```

```
import javax.inject.Named;
```

@Named

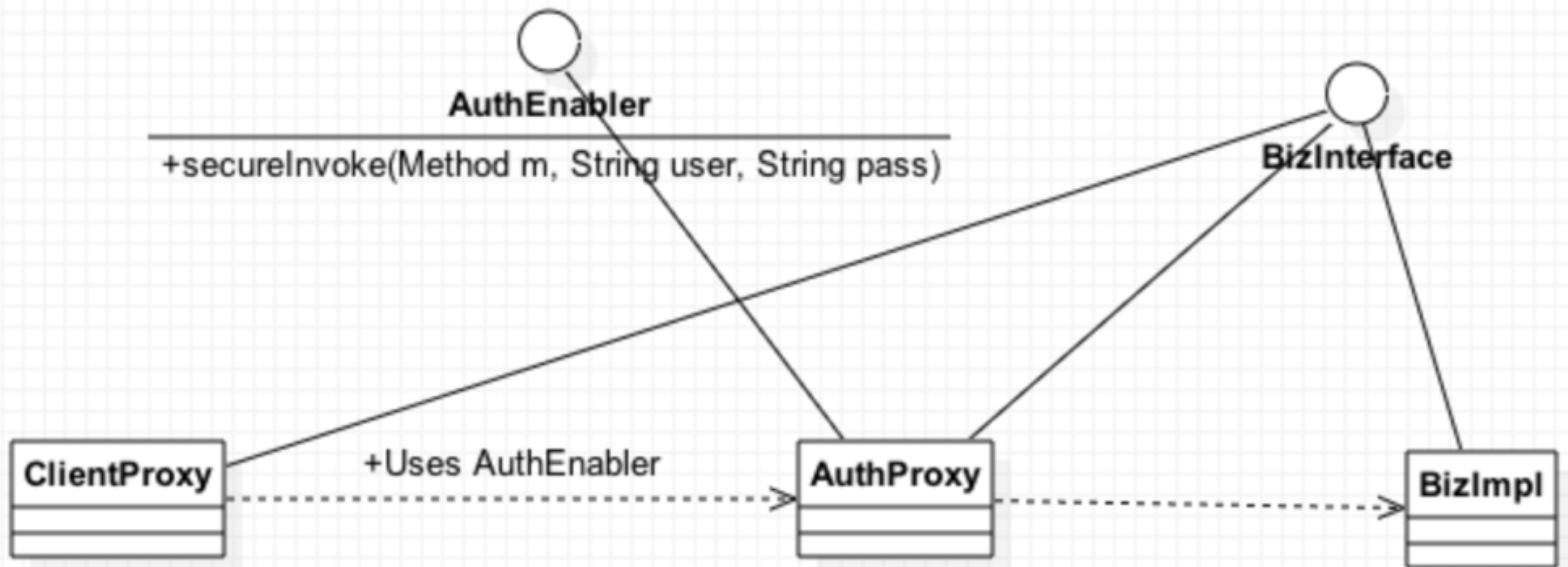
```
public class UserManager {
```

```
    @Inject
```

```
    private UserDao dao;
```



# Implement a Proxy for Auth Checking



# Steps

- AuthEnabler interface

```
public interface AuthEnabler {  
    public Object secureInvoke(Method m, String user, String password, Object[] args);  
}
```

- Create a dynamic proxy that can proxy any object plus implement AuthEnabler interface

```
List<Class> interfaces = new ArrayList(Arrays.asList(obj.getClass().getInterfaces()));  
interfaces.add(AuthEnabler.class);  
Class[] classArr = interfaces.toArray(new Class[1]);
```

- Implement Proxy to check security

# Proxy Code

```
if(m.getName().equals("secureInvoke")){  
    //Perform auth and call the target method  
    Method targetMethod = (Method)args[0];  
    String user = (String)args[1];  
    String pass = (String)args[2];  
    if(user.equals("admin") && pass.equals("admin123")){  
        Object[] methodArgs = Arrays.copyOfRange(args, 3, args.length-1);  
        return targetMethod.invoke(obj, methodArgs);  
    }else{  
        throw new Exception("Authentication info denied");  
    }  
}else{  
    //Direct method invocation - prevent it  
    throw new Exception("Authentication info not provided");  
}
```

# We Need A Client Proxy

- newInstance checking

```
if(!(obj instanceof AuthEnabler)){  
    throw new IllegalArgumentException("This class can only proxy Auth  
Enabled classes");  
}
```

- Proxy code

```
if(m.getName().equals("secureInvoke")){  
    //This should never be called directly  
    throw new IllegalStateException("Secure invoke should not be called on  
the client proxy directly. Call the target method");  
}else{  
    //Grab auth info and call the secureInvoke method on the target  
    String user = "admin";  
    String pass = "admin123";  
    return ((AuthEnabler)obj).secureInvoke(m, user, pass, args);  
}
```

# Using Secure Factories

```
UserDaoImpl daoImpl = new UserDaoImpl();  
UserDao secureDao = (UserDao)AuthProxyImpl.newInstance(daoImpl);  
  
UserDao authEnabledClient =  
(UserDao)ClientProxyImpl.newInstance(secureDao);  
System.out.println(authEnabledClient.getAllUsers());
```

# Create Factory Beans For Use in Spring

- Create proxy factories

```
public class AuthProxyFactory implements FactoryBean {  
public class ClientProxyFactory implements FactoryBean {
```

- We need to support configuration like this:

```
<!-- Secure Bean Setup -->  
<bean name="dao" class="com.mydomain.dao.UserDaoImpl"></bean>  
<bean name="secureDao" class="com.mydomain.security.AuthProxyFactory">  
    <property name="beanToSecure" ref="dao"></property>  
</bean>  
  
<bean id="authenticatedDao" name="authenticatedDao"  
        class="com.mydomain.security.ClientProxyFactory">  
    <property name="secureBean" ref="dao"></property>  
</bean>
```

# Can we improve on this

- We need a way to proxy all beans that need security without having to declare a proxy for each one
- Answer is BeanPostProcessors

# Proxying with BeanPostProcessors

- Create an annotation for securing beans

```
@Retention(RetentionPolicy.RUNTIME)
```

```
public @interface Secure {
```

```
}
```

- Proxy all beans that have this annotation

```
public Object postProcessBeforeInitialization(Object bean, String  
                                              beanName)
```

```
    throws BeansException {
```

```
    if (bean.getClass().isAnnotationPresent(Secure.class)) {
```

```
        return AuthProxyImpl.newInstance(bean);
```

```
    } else {
```

```
        return bean;
```

```
    }
```

```
}
```



# We Still Need Proxy On Client

- Client Proxy Config

```
<bean id="authenticatedDao" name="authenticatedDao"  
      class="com.mydomain.security.ClientProxyFactory">  
  <property name="secureBean" ref="dao"></property>  
</bean>
```

- Use proxied instances for injection instead of autowiring

```
@Resource(name="authenticatedDao")  
private UserDao dao;
```

# Spring AOP

- Provide declarative enterprise services, especially as a replacement for EJB declarative services.
- Allow users to implement custom aspects, complementing their use of OOP with AOP.

# AOP Terminology

- Aspect: a modularisation of a concern that cuts across multiple classes.
- Join point: a point during the execution of a program, such as the execution of a method or the handling of an exception.
- Advice: action taken by an aspect at a particular join point.
- Pointcut: a predicate that matches join points.

# Advice Types

- Before advice: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- After returning advice: Advice to be executed after a join point completes normally
- After throwing advice: Advice to be executed if a method exits by throwing an exception.
- After (finally) advice: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- Around advice: Advice that surrounds a join point such as a method invocation.

# Pointcut Designators

- execution - for matching method execution join points
- within - within certain types
- this - where the bean reference (Spring AOP proxy) is an instance of the given type
- target - where the target object is an instance of the given type
- args - where the arguments are instances of the given types
- @target - where the class of the executing object has an annotation of the given type
- @args - where the runtime type of the actual arguments passed have annotations of the given type(s)
- @within - within types that have the given annotation
- @annotation - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation
- bean - limits to named bean(s)

# Execution PCD format

- execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
  - Any public method: execution(public \* \*(..))
  - All setter methods: execution(\* set\*(..))
  - Methods of usermanager: execution(\* com.mydomain.UserManager.\*(..))
  - All methods in biz package: execution(\* com.mydomain.biz.\*.\*(..))

# Steps to Create Aspects

- Annotate class with:

@Component

@Aspect

- Annotate method with one of the pointcuts:

@Pointcut("execution(\* getAll\*(..))")// the pointcut expression

public void getAllMethods() {}// the pointcut signature

- Declare the methods to use pointcuts

@Before("com.mydomain.aop.LoggingAspect.getAllMethods()")

- Invoke bean methods

# Advice Method Signatures

- Before:

```
@Before( value="com.domain.aop.LoggingAspect.getAllMethods()", argNames="joinPoint")  
public void logMethodCallstart(JoinPoint joinPoint) {
```

- After:

```
@AfterReturning(returning="retVal",  
value="com.mydomain.aop.LoggingAspect.getAllMethods()")  
public void logMethodCallEnd(Object retVal) {
```

- AfterThrowing

```
@AfterThrowing(throwing="exception",value="com.mydomain.aop.LoggingAspect.getAllMethods  
()")  
public void logErrors(Throwable exception){
```

- Around

```
@Around("com.mydomain.aop.LoggingAspect.getAllMethods()")  
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {  
    Object retVal = pjp.proceed();
```



# Passing Params To Advice

- Any advice method may declare as its first parameter, a parameter of type `org.aspectj.lang.JoinPoint/ProceedingJoinPoint`
- Using this joinpoint, everything can be found about the method that is being advised.

# Lets Try It

- Apply logging advice to methods of UserManager to find how long the methods takes to execute using an around advice
- Implement Transaction Management with AOP

# Custom Tx Manager

- We will need a tx manager class that acts as an around advice

@Component

@Aspect

```
public class CustomTxManager {
```

- This class needs to obtain connection and set it in thread local after starting the transaction on it

```
@Around("execution(* getAll*(..))")
```

```
    public Object startTransactions(ProceedingJoinPoint pjp) throws Throwable {
```

```
        Connection con = getConnection();
```

```
        con.setAutoCommit(false);
```

```
        Object result=null;
```

```
        try{
```

```
            threadLocalConnections.set(con);
```

```
            result = pjp.proceed();
```

```
            con.commit();
```

```
        }finally{
```

```
            threadLocalConnections.get().close();
```

```
        }
```

```
        return result;
```

```
}
```

# Custom Tx Manager

- UserDaoImpl needs to obtain the connection from this threadpool using a spring generated lookup method

```
public abstract class UserDaoImpl implements UserDao {  
  
    public abstract Connection getCon();  
  
    public List<User> getAllUsers() throws Exception{  
        Statement statement = getCon().createStatement();
```

- Factory bean

```
public class CustomTxConnectionFactory implements FactoryBean<Connection>{  
  
    public Connection getObject() throws Exception {  
        return CustomTxManager.threadLocalConnections.get();  
    }  
}
```

# Custom Tx Manager

- Configuration in xml

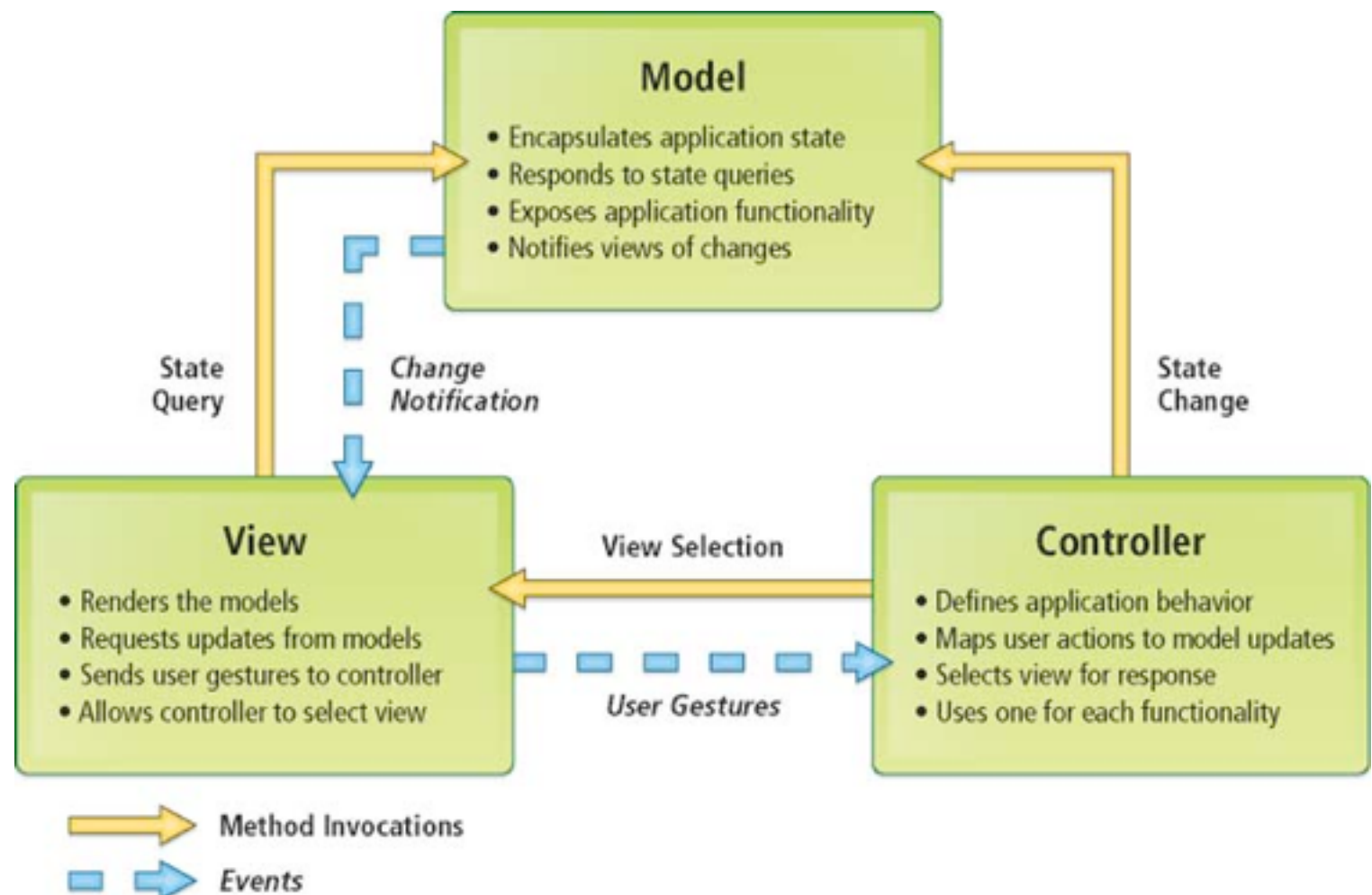
```
<bean name="dao" class="com.mydomain.dao.UserDaoImpl">  
  <lookup-method name="getCon" bean="connection" />  
</bean>
```

```
<bean name="connection"  
class="com.mydomain.aop.CustomTxConnectionFactory"/>
```

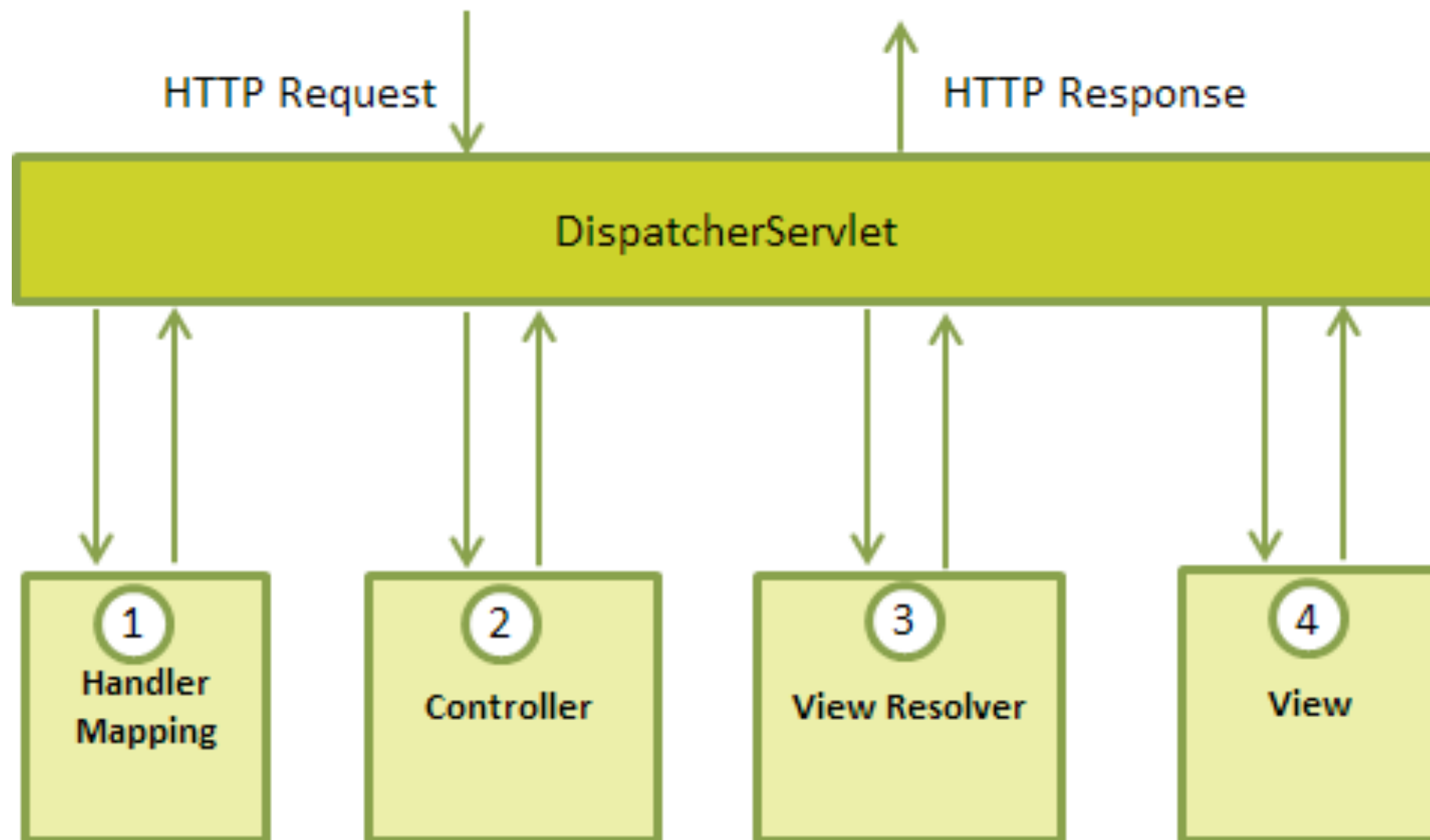
```
<bean name="customTxMgr" class="com.mydomain.aop.CustomTxManager">  
  <property name="driverClass"><value>org.apache...</value></property>  
  <property name="connectionUrl"><value>jdbc:derby:...</value></property>  
</bean>
```

# Spring MVC

- The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.



# Spring MVC Flow



# Hello World

- Web.xml

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

- WEB-INF/action-servlet.xml

```
<context:component-scan base-package="com.mydomain"/>
```

- Controller class

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "/helloWorld.jsp";
    }
}
```

- JSP

```
<%@ page isELIgnored="false"%>
<h2>${message}</h2>
```



# Passing Form Data

- Declare controller that accepts form data as a bean

@Controller

```
public class UserController {
```

```
    @Autowired
```

```
    UserManager userManager;
```

```
    @RequestMapping("/addUser")
```

```
    public String addUser(User u, Model model) throws Exception {
```

```
        userManager.addUser(u);
```

```
        model.addAttribute("message", "Added User!");
```

```
        return "/helloWorld.jsp";
```

```
    }
```

```
}
```

- Submit form to this servlet

# Validating Forms

- Implement Validators

```
public class UserValidator implements Validator{  
  
    public void validate(Object obj, Errors err) {  
        ValidationUtils.rejectIfEmptyOrWhitespace(err, "name", "field.required",  
            "Field Name is required");  
        ...  
    }  
}
```

- Change controller for validation handling

```
@RequestMapping("/addUser")  
public String addUser(@Valid User u, BindingResult bindingResult, Model model)  
throws Exception {  
    if (bindingResult.hasErrors()) {  
        return "UserForm";  
    } ...  
}
```

- Change Form to display validation errors

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="springForm"%>  
<springForm:errors path="user.age" cssClass="error" /><br>
```

# Model Attributes

- `@ModelAttribute` annotation on a method causes the method to be executed and its return value placed on a model whenever the controller is fired
- `@ModelAttribute` annotation on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model.
- Once present in the model, the argument's fields should be populated from all request parameters that have matching names.

# Populate Age List

- Create a model attribute method to return list of age values

```
@ModelAttribute("ageList")
```

```
public List<Integer> getAgeList(){  
    Integer[] ages = new Integer[]{18,19,20,21,22,23,24,25,26,27,28,29,30};  
    return Arrays.asList(ages);  
}
```

- Use it on JSP using jstl

```
<select name="age">  
    <c:forEach items="${ageList}" var="ageVal">  
        <option value="${ageVal}">${ageVal}</option>  
    </c:forEach>  
</select>
```

- Change it over to use Spring tags

```
<springForm:select path="user.age" items="${ageList}" ></springForm:select>
```

# URI Templates

- URI templates can be used for convenient access to selected parts of a URL in a `@RequestMapping` method.

```
@RequestMapping("/delete/{userId}")  
public String delete(@PathVariable int userId) throws Exception{
```

# Returning Data From Controller

- @ResponseBody annotation delivers the return value of the controller directly
- RequestMapping annotation needs to identify the return data content type properly

```
@RequestMapping(value="/listUsersJson", produces =  
MediaType.APPLICATION_JSON_VALUE)
```

# Create Users List - JSON

- Create Controller Method to get List of users

```
public String listUsersJson() throws Exception{  
    List<User> users = userManager.getAllUsers();
```

- Annotate it to allow direct data return and identify the response data to be json

```
@RequestMapping(value="/listUsersJson", produces = MediaType.APPLICATION_JSON_VALUE)  
@ResponseBody
```

- Convert the users list to json and return

```
String usersJson = JSONObject.valueToString(users);
```

- Write AJAX code to load this list

```
function loadUsers(){  
    $.ajax( {  
        url: 'listUsersJson',  
        method: 'get',  
        headers: { 'Accept': 'application/json' },  
        success: function(data) {  
            $("#userTable").html("");  
            for(var index in data){  
                var row = "<tr><td>" + data[index].name + "</td><td>" + data[index].age + "</td></tr>";  
                $("#userTable").append(row);  
            }  
        }  
    });  
}
```

# Remoting With Spring

- RMI
  - not possible to access the objects through the HTTP protocol
  - RMI is a fairly heavy-weight protocol
  - Important when using a complex data model that needs serialisation over the wire.
  - RMI-JRMP is tied to Java clients: It is a Java-to-Java remoting solution.



# Remoting With Spring

- Spring's HTTP invoker
- HTTP-based remoting but also rely on Java serialization.
- It shares the basic infrastructure with RMI invokers, just using HTTP as transport.

# Remoting With Spring

- JAX-WS
  - Standard WebServices using SOAP
  - Works with an compliant WebServices client
  - Complex datatypes can be hard to map and serialise to XML structures

# RMI Remoting

- Create Remote interface extending “Remote”
- Implement the interface in the business class

- Export it using a bean definition like this:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- does not necessarily have to be the same name as the bean to be
exported -->
    <property name="serviceName" value="UserManager"/>
    <property name="service" ref="userManager"/>
    <property name="serviceInterface"
                value="com.mydomain.service.UserManagerRemote"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="1199"/>
    <property name="alwaysCreateRegistry" value="true"/>
</bean>
```

# RMI Client

```
String name = "UserManager";  
Registry registry = LocateRegistry.getRegistry("localhost",1199);  
UserManagerRemote um = (UserManagerRemote) registry.lookup(name);  
List<User> users = um.getAllUsers();
```

OR Inject like this:

```
<bean id="userManagerRemote"  
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">  
    <property name="serviceUrl" value="rmi://localhost:1199/  
userManager"/>  
    <property name="serviceInterface"  
value="com.mydomain.service.UserManagerRemote"/>  
</bean>
```