Spring Framework

Luiz Fernando Rodrigues
lfrodrigues@teccomm.les.inf.puc-rio.br

Spring

- Open source application Framework
  - Tem como objetivo facilitar o desenvolvimento J2EE
- Single tier frameworks
  - Struts
  - Tapestry
  - Hibernate
  - Ibatis
  - ...
- Permite a união desses frameworks utilizando-se de uma arquitetura coerente.

Abordagem J2EE tradicional

- Muito complexas
- Necessitam de muito esforço de desenvolvimento
- Baixa performance
- Excesso de “plumbing” code
  - Código nem sempre útil
    - JNDI lookup
    - Transfer Objects
    - Try/catch para alocar e desalocar recursos
  - Manter tal código pode sair caro

Abordagem J2EE tradicional (cont.)

- Modelo distribuído de objetos
  - Nem sempre é adequado
- Padrões de Projeto J2EE
  - Não são padrões
  - “Workarouds” para limitações tecnológicas
- Difíceis de se aplicar teste de unidade
  - Componentes EJBs foram definidos antes de práticas ágeis

Solução aplicada para problemas do J2EE

- Geração automática de código
  - Mais “plumbing code”
- Uso de Frameworks
  - Mais flexíveis que a geração automática de código
  - É possível configurar o comportamento de uma pequena parte do framework, ao invés de modificar diversas classes geradas automaticamente.
  - Uso de abstração
- Logo,
  - Vamos utilizar um Framework!

Frameworks

- No início,
  - Cada companhia desenvolvia seu próprio framework
  - Manutenção é muito cara e torna a abordagem inviável
- Após,
  - Necessidade de um framework
    - Usado por muitos
    - Testado por muitos
    - Open Source
- Assim,
  - Nascimento dos single tier frameworks
    - Hibernate
    - Struts

Lightweight Frameworks

- Redução da complexidade J2EE
  - Evitar complexidade desnecessária
  - Startup rápido
  - Poucos dependências
  - Independente de ambiente
  - Infra-estrutura para testes
- Não restringir o desenvolvedor
- Usar componentes “out of the box”

Spring

- Lightweight Framework
- Envolve todas as camadas de uma aplicação J2EE típica
- Módulos
  - Inversion of Control Container
  - AOP framework
  - Data access abstraction
  - Simplificação JDBC
  - Transaction Management
  - Framework Web MVC
  - Simplificação para trabalhar com
    - JNDI
    - JTA
  - Lightweight remoting
    - RMI, IIOP
  - Suporte para estratégias de teste

Valores do Spring

- É um framework não invasivo
  - Alguns frameworks forçam que a aplicação seja implementada para eles.
    - EJB
    - Apache
  - Minimiza a dependência da aplicação com o framework.
  - Vantagens
    - A aplicação pode ser executada sem o framework
    - Migração para futuras versões do Spring é mais facil
    - Aplicação do framework em código legado.

Valores do Spring

- Modelo de programação consistente
  - Utilizável em qualquer ambiente
  - Não é necessário por exemplo, um servidor de aplicação caro, pode-se optar por utilizar um gratuito, como o Tomcat.
  - Separa aplicação do ambiente de execução
    - Configurações JNDI por exemplo
    - Independência de contexto
- Facilita um Design Orientado a Objetos
  - Algumas aplicações J2EE simplesmente não são OO. Embora tenham sido desenvolvidas com uma linguagem OO.

Valores do Spring

- Facilita o uso de boas práticas
  - Implementação com interfaces ao invés de classes
  - Inversão de Controle (IoC)
    - Código responsável pela chamada de objetos está protegido se a aplicação for desenvolvida com interfaces
- Promove “Plugabilidade”
  - Uso de serviços
  - Dependência entre serviços é expressa em termos de interfaces
  - Troca de serviços sem causar impacto no resto da aplicação
- Configuração
  - Uso de arquivos de configuração ao invés de configuração em código java. (Hardcoded)
  - Arquivos XML ou .properties

Valores do Spring

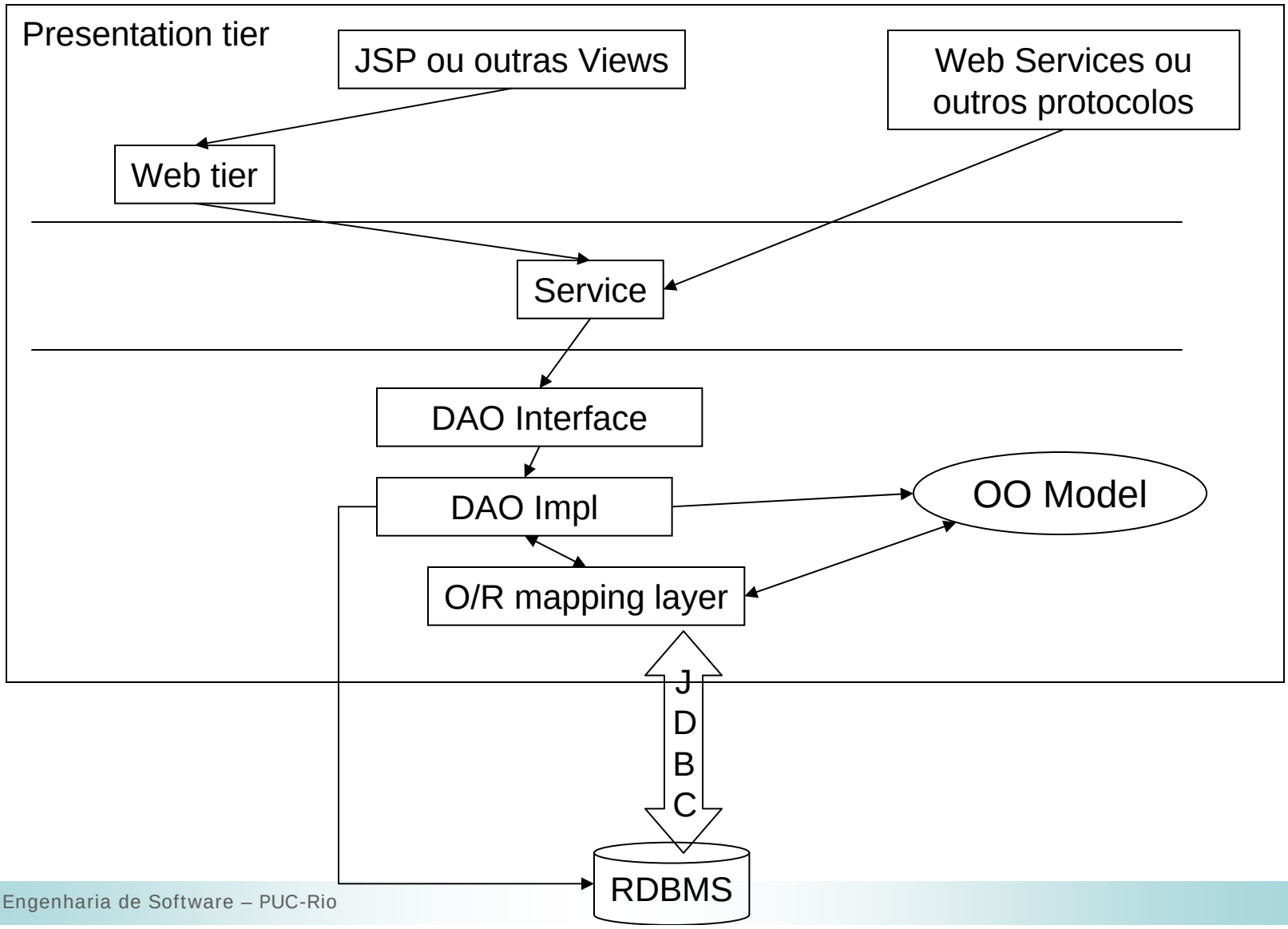
- Facilita o teste
  - Desenvolvimento com POJOs
    - Fáceis de testar
- Consistente
  - Abordagem consistente em diversas partes do framework
    - Uma vez aprendida um parte é mais fácil compreender o resto
- Escolha da Arquitetura
  - Embora ofereça um “backbone” a troca de camadas é facilitada
  - Exemplo
    - Troca entre frameworks de mapeamento O/R sem impacto na camada de lógica de negócio
    - Troca entre frameworks MVC sem impacto nas camadas de persistência

Valores do Spring

- Não reinventa a roda
  - Embora possua um escopo abrangente, o Spring não introduz soluções de:
    - Mapeamento O/R
    - Abstrações de Log
    - Pool de Conexões
    - Coordenador de transações distribuidas
    - Protocolos

Arquitetura de Aplicações Spring

Big Picture


Presentation tier
JSP ou outras Views
Web Services ou outros protocolos
Web tier
Service
DAO Interface
DAO Impl
OO Model
O/R mapping layer
J
D
B
C
RDBMS

Camadas

- Apresentação
  - Tipicamente em versões web
- Negócios
  - Fachada para o modelo
- DAO
  - Interface independente do tipo de acesso de dados utilizado para busca ou persistência.
- Modelo OO
  - Objetos que solucionam o problema em questão
- Base de Dados ou Sistemas Legados
  - RDBMS
  - Sistemas antigos que devem “conversar” com a nova aplicação

Persistência - DAO

- Data Access Objects
- Spring encoraja o uso de DAOs
- Encapsula o acesso aos objetos persistentes do domínio
  - Persiste objetos transientes
  - Atualiza objetos persistidos
  - Busca objetos persistidos
- Desacoplamento entre objetos de serviço e objetos de persistência
- Implementação por interfaces
  - Permite a troca da camada de persistência sem impactar a camada de serviço
  - Container de IoC usa injeção de dependência dos objetos DAOs aos objetos de serviço

DAO - Exemplo

- Uma interface típica para um DAO:

```
public interface ReminderDao
{
public Collection findRequestsEligibleForReminder()
       throws DataAccessException;

void persist(Reminder reminder) throws DataAccessException;

void countOutstandingRequests() throws DataAccessException;
}
```

DAO - Exemplo

- Implementação utilizando Hibernate:

```
public class HibernateReminderDao
        extends HibernateDaoSupport
        implements ReminderDao
{
public Collection findRequestsEligibleForReminder()
                throws DataAccessException
{
        getHibernateTemplate().find("from Request r where
r.something = 1");
}

public void persist(Reminder reminder)
                throws DataAccessException
{
        getHibernateTemplate().saveOrUpdate(reminder);
}
}
```

DAO - Exemplo

- Implementação utilizando JDBC:

```
public class JdbcReminderDao extends JdbcDaoSupport
        implements ReminderDao {
public Collection findRequestsEligibleForReminder()
        throws DataAccessException {
        return getJdbcTemplate().query("SELECT NAME, DATE, ... " +
                " FROM REQUEST WHERE SOMETHING = 1",
        new RowMapper() {
                public Object mapRow(ResultSet rs, int rowNum)
                        throws SQLException
                {
                        Request r = new Request();
                        r.setName(rs.getString("NAME"));
                        r.setDate(rs.getDate("DATE"));
                        return r;
                }
        });
}
public int countRequestsEligibleForReminder()
        throws DataAccessException {
        return getJdbcTemplate.queryForInt("SELECT COUNT(*) FROM...”);
}
}
```

Objetos de Serviço

- Implementação por Interfaces
- Lógica de negócio para casos de uso específicos
- Atuam como fachada para o modelo
- Gerência de Transações
- Garantia de Restrições de Segurança

Apresentação

- Camadas inferiores (Serviço e Persistência) não devem "conhecer" esta camada

- Camada Web MVC
  - Responsável por interagir com o sistema
  - Controlador
    - Processam entradas de usuários e invocam o serviço necessário para isso
  - Modelo
    - Contém informação da lógica de negócio e devem ser exibidos na resposta
  - View
    - Exibem objetos do modelo

Apresentação

- Spring suporta diversos frameworks com funcionalidade de apresentação
  - Struts
  - Spring-MVC
  - Tapestry
  - WebWork
  - JSP

Container de IoC

Inversão de Controle

Container de IoC

- Idéia central do Spring
- Permite a “cola” de aplicações sem que as mesmas estejam preparadas para isso
- Dependências entre objetos são tratadas por meio de interfaces java ou classes abstratas
- BeanFactory e ApplicationContext

WTHI IoC?

- Objetos devem interagir com outros objetos para que realizem suas funções
  - Dizemos que os objetos possuem dependências
- Inversion of Control
  - Padrão arquitetural que permite que uma entidade externa ligue os objetos dependentes, sem que eles precisem lidar com isso diretamente.
  - Princípio de Hollywood
    - “Don’t call me, I’ll call you”
- Entidade Externa
  - Container de IoC do Spring

Exemplo non-IoC - WeatherService


```
public class WeatherService {

WeatherDAO weatherDao = new StaticDataWeatherDAOImpl();

        public Double getHistoricalHigh(Date date)
        {
                WeatherData wd = weatherDao.find(date);
                if (wd != null)
                        return new Double(wd.getHigh());
                return null;
        }
}
```

Exemplo non-IoC - WeatherDAO


```
public interface WeatherDAO {
WeatherData find(Date date);
WeatherData save(Date date);
WeatherData update(Date date);
}
```

```
public class StaticDataWeatherDAOImpl implements WeatherDAO {
public WeatherData find(Date date) {
        WeatherData wd = new WeatherData();
        wd.setDate((Date) date.clone());
        ...
        return wd;
        }
public WeatherData save(Date date) {
        ...
        }
public WeatherData update(Date date) {
        ...
        }
}
```

Exemplo non-IoC - WeatherServiceTest


```
public class WeatherServiceTest extends TestCase {

public void testSample1() throws Exception {
        WeatherService ws = new WeatherService();
        Double high = ws.getHistoricalHigh(
        new GregorianCalendar(2004, 0, 1).getTime());
        // ... do more validation of returned value here...
        }
}
```

Problemas da Abordagem non-IoC

- WeatherService
  - Instancia diretamente o DAO e controla seu ciclo de vida
  - Não possui uma interface
  - Se trocarmos a implementação de DAO o serviço é alterado

- WeatherServiceTest
  - Instancia diretamente o objeto de Serviço, pois Serviço não possui interface
  - Se trocarmos o objeto de serviço a classe de teste é alterada

Exemplo IoC - WeatherService


```
public interface WeatherService {

        Double getHistoricalHigh(Date date);
}
```

```
public class WeatherServiceImpl implements WeatherService
{
        private WeatherDAO weatherDao;
        public void setWeatherDao(WeatherDAO weatherDao) {
                this.weatherDao = weatherDao;
        }

        public Double getHistoricalHigh(Date date) {
                WeatherData wd = weatherDao.find(date);
                if (wd != null)
                return new Double(wd.getHigh());
                return null;
        }
}
```

Declaração da Dependência

- Arquivo XML
  - aplicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
	<bean id="weatherService"
	class="ch02.sample2.WeatherServiceImpl">
		<property name="weatherDao">
			<ref local="weatherDao"/>
		</property>
	</bean>
	<bean id="weatherDao"
	class="ch02.sample2.StaticDataWeatherDaoImpl">
	</bean>
</beans>
```

Alteração do teste


```
public class WeatherServiceTest extends TestCase
{
	public void testSample2() throws Exception {

	ApplicationContext ctx = new
	ClassPathXmlApplicationContext(
		"ch03/sample2/applicationContext.xml");

	WeatherService ws = (WeatherService)
		ctx.getBean("weatherService");

	Double high = ws.getHistoricalHigh(
		new GregorianCalendar(2004, 0, 1).getTime());
	// ... do more validation of returned value here...
	}
}
```

Container de IoC

Injeção de Dependência

Injeção de Dependência

- Descreve o processo de fornecer um objeto (componente) a outro objeto (componente) seguindo uma arquitetura IoC
- Injeção por construtor
- Injeção por setter
- Injeção por métodos

Injeção por Construtor


```
public interface WeatherService {
        Double getHistoricalHigh(Date date);
}
```

```
public class WeatherServiceImpl implements WeatherService
{
        private final WeatherDao weatherDao;

        public WeatherServiceImpl(WeatherDao weatherDao) {
                this.weatherDao = weatherDao;
        }

        public Double getHistoricalHigh(Date date) {
                WeatherData wd = weatherDao.find(date);
                if (wd != null)
                return new Double(wd.getHigh());
                return null;
        }
}
```

Declaração de injeção por construtor


```
<beans>
<bean id="weatherService"
      class="ch02.sample3.WeatherServiceImpl">
      <constructor-arg>
            <ref local="weatherDao"/>
      </constructor-arg>
</bean>

<bean id="weatherDao"
      class="ch02.sample3.StaticDataWeatherDaoImpl">
</bean>

</beans>
```

Injeção por Setter

- Igual ao exemplo utilizado na sessão de IoC
- Utiliza-se do método de set para injetar a dependência

Injeção por Métodos

- Raramente utilizada
- Container implementa método em tempo de execução
- Uso
  - Objeto singleton e sem estado utiliza um objeto não-singleton, com estado e não-threadsafe
- Exemplo
  - WeatherService
  - Utiliza um StatefulWeatherDAO
  - Cada chamada pelo getWeatherDAO deve voltar um novo objeto
- Solução
  - Injeção de dependência por métodos em getWeatherDAO

Exemplo

```
public abstract class WeatherServiceImpl implements
WeatherService {

protected abstract WeatherDao getWeatherDao();

public Double getHistoricalHigh(Date date) {

        WeatherData wd = getWeatherDao().find(date);
        if (wd != null)
                return new Double(wd.getHigh());
        return null;
        }
}
```

Exemplo

```
<beans>
<bean id="weatherService"
              class="ch02.sample4.WeatherServiceImpl">
      <lookup-method name="getWeatherDao" bean="weatherDao"/>
</bean>

<bean id="weatherDao" singleton="false"
              class="ch02.sample4.StatefulDataWeatherDaoImpl">
</bean>

</beans>
```

Container de IoC

Bean Factory e Application Context

BeanFactory

- Interface que caracteriza o container de IoC
- Implementações fornecem a instanciação e injeção de dependência adequada
  - Uso de reflexão
- Código de cola
  - Pouca quantidade conhece essas interfaces
- Código Aplicações
  - Não conhecem essas interfaces
- Diversas implementações
  - org.springframework.beans.factory.BeanFactory
- Tipicamente configurada a partir de um arquivo XML

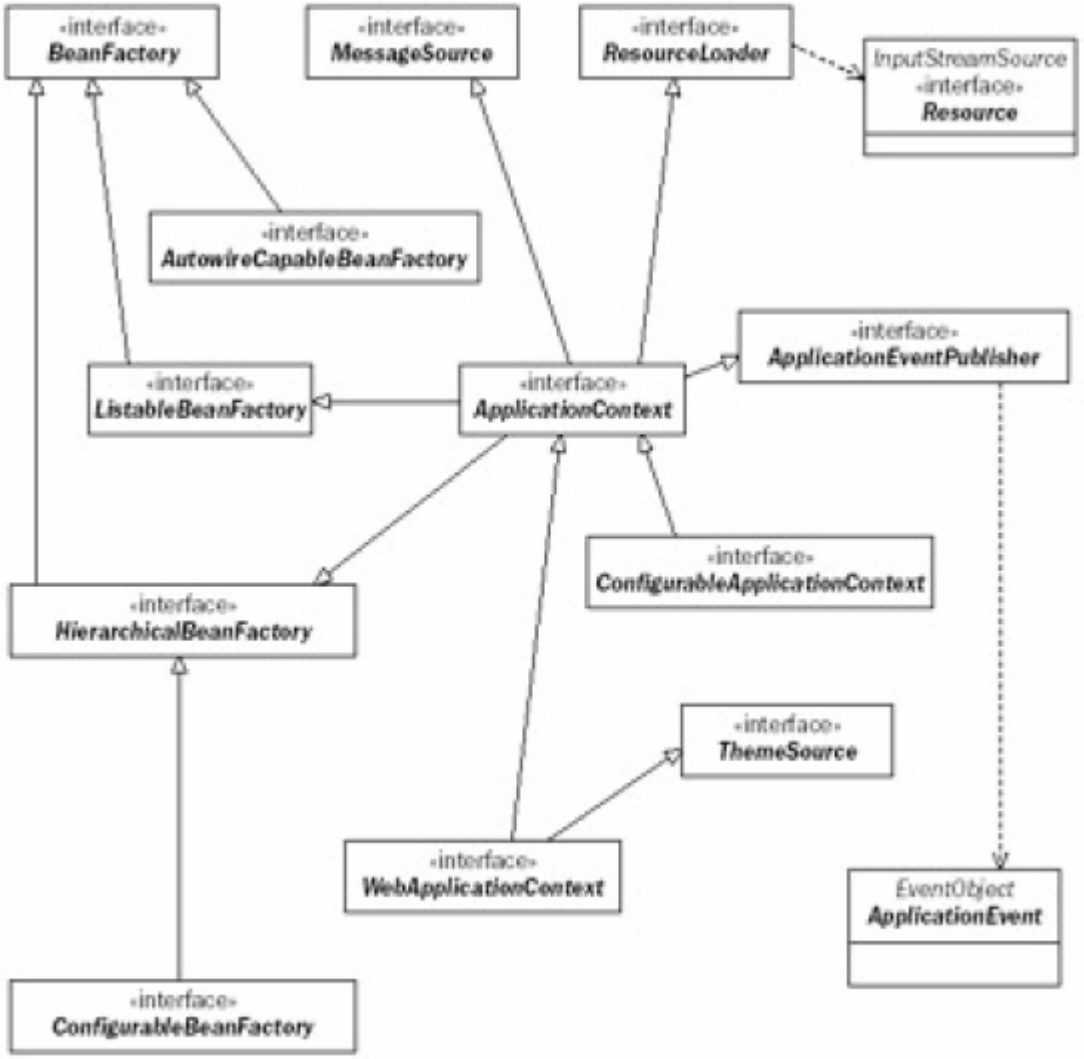
Hierarquia de Interfaces

- BeanFactory
  - Métodos para acessar as fábrica
    - getBean(String name)
    - getBean(String name, Class requiredType)
    - Métodos de query
- HierarchicalBeanFactory
  - Fábricas podem ser compostas em hierarquia
  - Caso o bean não seja encontrado na fábrica é procurado em seu pai
- ListableBeanFactory
  - Permite a listagem de bean de uma fábrica
- AutowireCapableBeanFactory
  - Cria as dependências externas de um bean
    - autowireBeanProperties()
    - applyBeanPropertiesValues()
    - Trabalho com código de terceiros

Application Context

- É um tipo de BeanFactory
  - org.springframework.context.ApplicationContext
- Diferenças com BeanFactory
  - Pos-processadores
    - Bean
    - Factory Beans
  - Message Source
    - Interface que ajuda a localização de mensagens
  - Suporte para eventos da aplicação e do framework
    - Disparo de eventos para “listeners” registrados
  - Resource Loader
    - Tratamento de recursos de baixo nível
- Quando utilizar Application Context no lugar de BeanFactory
  - Sempre ;)

Interfaces Existentes



Laçamento do Container

- Único arquivo de configuração
- Vários arquivos de configuração
  - Combinação de Fragmentos
- Utilização de Recursos
- Obtendo os Beans

Único arquivo de configuração

- A partir de um recurso no classpath

```
ApplicationContext appContext =
new ClassPathXmlApplicationContext("ch03/sample2/applicationContext.xml");
// side note: an ApplicationContext is also a BeanFactory, of course!
BeanFactory factory = (BeanFactory) appContext;
```

- A partir de um recurso no sistema de arquivos

```
ApplicationContext appContext =
new FileSystemXmlApplicationContext("/some/file/path/applicationContext.xml");
```

Combinando fragmentos

applicationContext-dao.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean id="weatherDao" class="ch02.sample2.StaticDataWeatherDaoImpl">
        </bean>
</beans>
```

applicationContext-services.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean id="weatherService" class="ch02.sample2.WeatherServiceImpl">
                <property name="weatherDao">
                        <ref bean="weatherDao"/>
                </property>
        </bean>
</beans>
```

Combinando Fragmentos (2)

```
ApplicationContext appContext =
        new ClassPathXmlApplicationContext(
        new String[]
                {
                        "applicationContext-serviceLayer.xml",
                        "applicationContext-dao.xml“
                }
        );
```

```
ApplicationContext appContext =
        new
        ClassPathXmlApplicationContext("classpath*:ApplicationContext.xml");
```

Uso de Recursos


```
ClassPathResource res =
new ClassPathResource("org/springframework/prospering/beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(res);
```

```
FilesystemResource res =
new FilesystemResource("/some/file/path/beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(res);
```

```
InputStream is = new FileInputStream("/some/file/path/beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(is);
```

Obtendo os beans

Por meio do método de get:

```
WeatherService ws = (WeatherService) ctx.getBean("weatherService");
```

Por meio do método da listagem:

```
Map allWeatherServices = ctx.getBeansOfType(WeatherService.class);
```

Configuração dos Beans

- Definição Básica
- Mecanismo de Criação
- Singleton x Não-singleton beans
- Especificação de Dependências
  - ref
  - idref
  - bean
  - list
  - set
  - map
  - props
  - value
  - Null
- Dependência de Declarações
- Autowire de dependências
- Match de argumentos de construtor
- Validação de dependências

Definição básica

- Id
- Name
  - Url path
  - Alias (diversos nome para um bean)

```
<beans>
        <bean id="bean1" class="ch02.sample5.TestBean"/>
        <bean name="bean2" class="ch02.sample5.TestBean"/>
        <bean name="/myservlet/myaction" class="ch02.sample5.TestBean"/>
        <bean id="component1-dataSource"
                name="component2-dataSource,component3-dataSource"
                class="ch02.sample5.TestBean"/>
</beans>
```

Mecanismos de Criação

- Factory method
  - Retorna instância de qualquer tipo
  - Static

```
<bean id="testBeanObtainedViaStaticFactory"
class="ch02.sample4.StaticFactory" factory-method="getTestBeanInstance"/>
```

```
public class StaticFactory {
        public static TestBean getTestBeanInstance() {
                return new TestBean();
        }
}
```

Mecanismos de Criação

- Factory method
  - Non-Static

```
<bean id="nonStaticFactory" class="ch02.sample4.NonStaticFactory"/>

<bean id="testBeanObtainedViaNonStaticFactory"
        factory-bean="nonStaticFactory"
        factory-method="getTestBeanInstance"/>
```

```
public class NonStaticFactory {

        public TestBean getTestBeanInstance() {
                return new TestBean();
        }
}
```

Singleton X Non-Singleton

- Singleton
  - Apenas uma instância é criada
  - Bean sem estado
  - Comportamento *default*
  - Referência mantida pela fábrica
- Non-Sigleton
  - Uma instancia diferente é fornecida a cada dependência
  - Bean com estado
  - Referência perdida pela fábrica

```
<bean id="singleton1" class="ch02.sample4.TestBean"/>

<bean id="singleton2" singleton="true" class="ch02.sample4.TestBean"/>

<bean id="prototype1" singleton="false" class="ch02.sample4.TestBean"/>
```

Especificação de Dependências

- ref
  - Elemento utilizado para configurar propriedade ou construtor com beans existentes
  - 3 atributos mutuamente exclusivos
    - local
      - Id do bean referenciado em tempo de parsing
    - bean
      - Id de bean localizado neste xml ou em outro arquivo, incluindo arquivos pais
      - Resolvido no momento em que as dependências são executadas
    - parent
      - Bean procurado a partir da fábrica pai
      - Utilizado quando existem conflitos com os nomes dos beans
- value
  - Elemento utilizado para configurar propriedade ou construtor com valores “simples”
    - String
    - Inteiros
    - Class
    - Etc..
  - PropertyEditors
    - Mapeamento para diversos valores

Especificação de Dependências

- value
  - Exemplo
  - Se classname for do tipo Class, um PropertyEditor transforma o valor de String em Class

```
<property name="classname">
        <value>ch02.sample6.StaticDataWeatherDaoImpl</value>
</property>
```

- null
  - Valores vazios são tratados com Strings vazias e não como null

```
<property name="optionalDescription"><null/></property>
```

Especificação de Dependências

- idRef
  - ???

Especificação de Dependências

- list, set, map e props
  - java.util.List
  - java.util.Set
  - java.util.Map
  - java.util.Properties
- Implementação de Collections java
- Podem possuir qualquer valor no interior

Especificação de Dependências

```
<beans>
        <bean id="collectionsExample" class="ch02.sample7.CollectionsBean">
        <property name="theList">
                <list>
                        <value>red</value>
                        <value>red</value>
                        <value>blue</value>
                        <ref local="curDate"/>
                        <list>
                                <value>one</value>
                                <value>two</value>
                                <value>three</value>
                        </list>
                </list>
        </property>
        <property name="theSet">
                <set>
                        <value>red</value>
                        <value>red</value>
                        <value>blue</value>
                </set>
        </property>
```

```
<beans>
        …
        <property name="theMap">
                <map>
                        <entry key="left">
                                <value>right</value>
                        </entry>
                        <entry key="up">
                                <value>down</value>
                        </entry>
                        <entry key="date">
                                <ref local="curDate"/>
                        </entry>
                </map>
        </property>
        <property name="theProperties">
                <props>
                        <prop key="left">right</prop>
                        <prop key="up">down</prop>
                </props>
        </property>
</bean>

<bean id="curDate" class="java.util.GregorianCalendar"/>
</beans>
```

Dependência de Declarações

- Força que um bean seja iniciado antes que outro
- Operação estática no momento que é carregado

```
<bean id="load-jdbc-driver" class="oracle.jdbc.driver.OracleDriver "/>

<bean id="dataBaseUsingBean" depends-on="load-jdbc-driver" class="..." >
...
</bean>
```

Autowire de dependências

- Propriedades não são explicitamente declaradas
- Tradeoff
  - Perda de transparência
  - XML mais enxuto
- Tipos de Autowire
  - no
    - Nenhum autowire declarado. Significa que todas as propriedades e construtores devem ser declarados explicitamente
  - byName
    - Faz um match com o nome da propriedade e algum outro bean declarado com mesmo nome
    - Caso o match não seja realizado a propriedade é configurada como nula

Autowire de dependências

- Tipos de Autowire (cont.)
  - byType
    - Para cada propriedade, verifica se existe exatamente um bean declarado que possua o mesmo tipo que a propriedade.
    - Caso existam dois beans de mesmo tipo é considerado um erro fatal
  - constructor
    - Funciona de forma parecida ao byType
    - Necessita que existam exatamente um match de tipo de bean por argumento do construtor
  - autodetect
    - Escolhe o mais apropriado entre byType e constructor
    - Se não existir construtor com argumentos byType é utilizado
    - Caso contrário, constructor é utilizado

Autowire de Dependências

- Exemplo
  - byName

```
<beans>
        <bean id="weatherService" autowire="byName"
        class="ch02.sample2.WeatherServiceImpl">
        <!-- no more weatherDao property declaration here -->
        </bean>

        <bean id="weatherDao"
        class="ch02.sample2.StaticDataWeatherDaoImpl">
        </bean>
</beans>
```

Match de Argumentos do Construtor

- Atributos
  - index
  - type

```
<beans>
        <bean id="errorBean" class="ch02.sampleX.ErrorBean">
                <constructor-arg index="0"><value>1000<value></constructor-arg>
                <constructor-arg index="1"><value>Unexpected Error<value></constructor-arg>
        </bean>
</beans>
```

```
<beans>
        <bean id="errorBean" class="ch02.sampleX.ErrorBean">
                <constructor-arg type="int"><value>1000<value></constructor-arg>
                <constructor-arg type="java.lang.String">
                        <value>Unexpected Error<value>
                </constructor-arg>
        </bean>
</beans>
```

Validação de Dependências

- Lançamento de erro caso uma propriedade não seja fornecida
  - Declaração explicita
  - Autowiring
- Atributo dependency-check
- Tipos de validação
  - none
    - Nenhuma validação é feita
  - simple
    - Caso não sejam configuradas, Collections e tipos primitivos, um erro é lançado
    - Outras propriedades podem continuar sem configuração
  - objects
    - Complemento da validação simple: somente tipos não primitivos e não collections
  - All
    - Todas a Validações: Collections, tipos primitivos e tipos complexos

Tópicos não abordados

- Ciclo de vida dos beans
- Reutilização de definição de beans
- Uso de pos-processadores
  - Uso de PropertyEditors

Spring com MVC

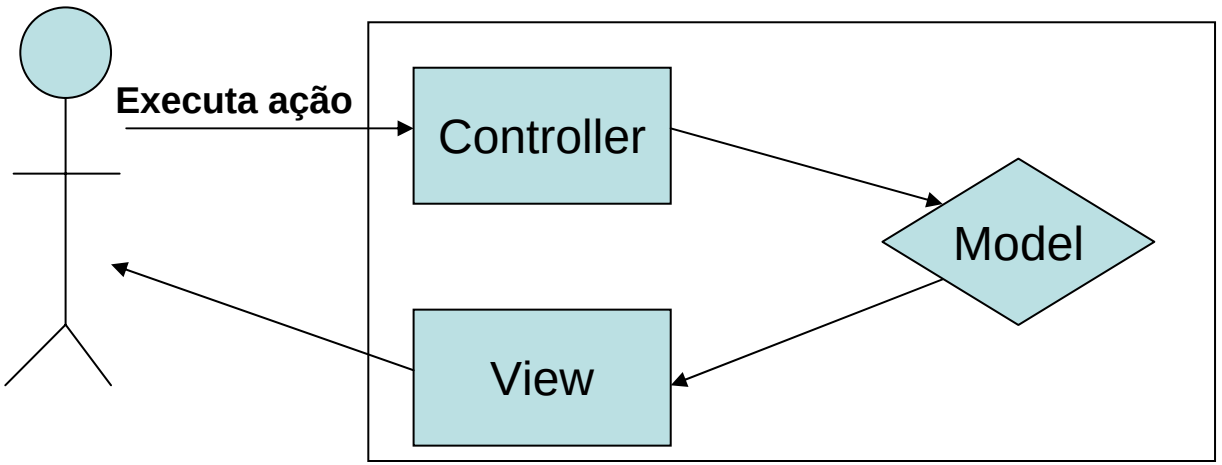
Andrew Diniz da Costa.

Guia da Apresentação

- Conceito MVC
- Spring MVC
- Classes importantes do framework
- Relação de URL com controlador
- Respostas dos controladores
- Exceções
- Principais classes Controller
- Data Binding
- Testar controladores

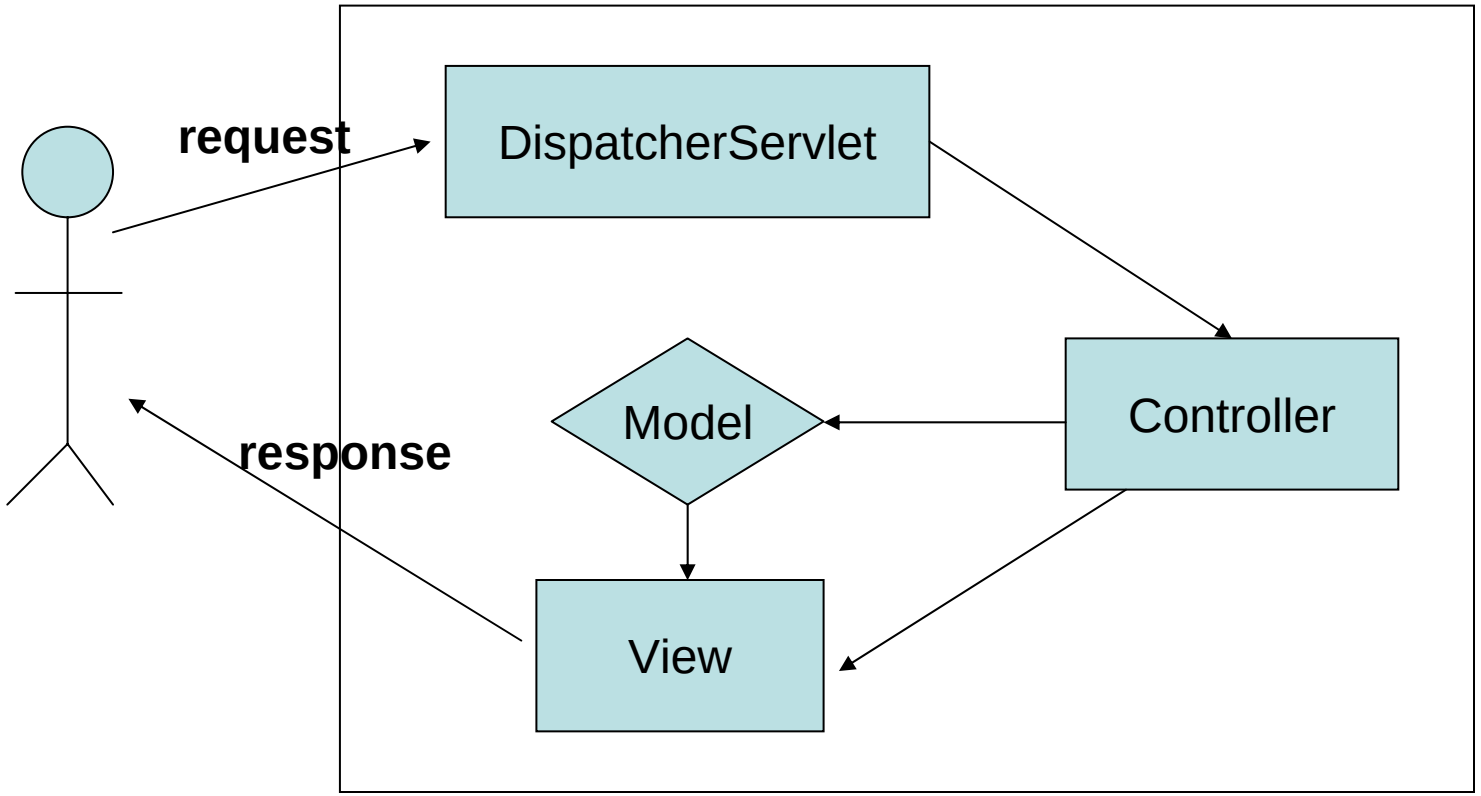
Conceito MVC

- Model View Controller – MVC
  - Model: Domínio da aplicação (dados dos objetos).
  - View: Interface.
  - Controller: Intermediário entre as camadas view e model.
- Implementação GUI



Spring MVC

- Dispatcher
  - Determina qual controlador será o usado



Spring MVC

- Spring provê diversas implementações para controladores.
- Maioria dos controladores Spring derivam da interface: "org.springframework.web.servlet.mvc.Controller"

ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res) throws Exception

  - Retornar um ModelAndView;

Spring MVC

- ModelAndView
  - Permite especificar uma resposta para o cliente a partir de ações de um controlador;
  - Nome da view especifica a página redirecionada.
  - Objetos podem ser usados pela página de destino.

```
ModelAndView mav = new ModelAndView("showDetailsTemplate");

mav.addObject("show", showDao.getShow());
return mav;
```

Spring MVC

- Dispatcher Servlet
  - Classe oferecida pelo framework Spring;
  - Deve ser definido no web.xml da aplicação (acesso Http);
  - Trabalha em conjunto com arquivos xml: WebApplicationContext (controladores, etc) e ApplicationContext (define validações, etc).
  - Gerenciador da manipulação de request-response;

web.xml

```
<servlet>
        <servlet-name>sample</servlet-name>
        <servlet-class>
        org.springframework.web.servlet.Dispatcherservlet
        </servlet-class>
        ...
</servlet>
```

Spring MVC

```
<servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.edit</url-pattern>
</servlet-mapping>

<init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value> /WEB-INF/ctx/controllers.xml
                 /WEB-INF/ctx/utils.xml
                </param-value>
</init-param>
```

Spring MVC

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
        /WEB-INF/applicationContext.xml
        </param-value>
</context-param>
```

Classes importantes do framework

- HandlerMapping
  - Responsável por determinar o apropriado controlador.
- ViewResolver
  - Capaz de mapear o nome das views.
- MultipartResolver
  - Provê suporte para realizar upload de arquivos.
- HandlerExceptionResolvers
  - Provê suporte para exceções inesperadas (ex: fornecer alguma informação ao usuário de algum erro).
- HandlerInterceptors
  - Capaz de interceptar requisições HTTP. Muito útil para adicionar comportamentos, como: logging, etc.

Relação de URL com controlador

- Existem várias maneiras de representar relações de urls com controladores.
- BeanNameIUrlHandlermapping

<bean name="account.edit /secure/*account.edit"
class="example.AccountFormController"/>

/secure/*account.edit -> /secure/smallaccount.edit
/secure/test.?sp -> /secure/test.jsp /secure/test.asp
//test.jsp** -> /secure/test.jsp /nonsecure/deeper/test.jsp

Relação de URL com controlador

- SimpleUrlHandlerMapping

```
<bean id=accountEditor“ class=“example.AccountFormController”/>

<bean id=“handlerMapping”
class=“org.spingframework.web.servlet.SimpleHandlerMapping”>
        <property name=“urlMap”>
                <map>
                        <entry key=“/secure/account*.edit”>
                                <ref bean=“accountEditor”/>
                        </entry>
                </map>
        </property>
</bean>
```

Respostas dos controladores

- Criação de um ModelAndView
- Nome do ModelAndView determina a página de destino. Mas como?

```
ModelAndView mav = new ModelAndView("lista"); // "/WEB-INF/jsps/lista.jsp"
```

```
<bean class="org.springframework...view.InternalResourceViewResolver">
        <property name="viewClass">
                <value>org.springframework...view.JstlView</value>
        </property>
        <property name="prefix">
                <value>/WEB-INF/jsps</value>
        </property>
        <property name="suffix">
                <value>.jsp</value>
        </property>
</bean>
```

Exceções

- Normal (interface Controller)
  - public ModelAndView handle(HttpServletRequest request, HttpServletResponse response) throws Exception
- Diferenciado (AbstractController)
  - public ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws **ShowNotFoundException**
  - **ShowNotFoundException** tem que ser implementado.
  - **ShowNotFoundException** implementa a interface **SimpleMappingExceptionResolver.**

Exceções

- SimpleMappingExceptionResolver

public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

Principais classes Controller

- AbstractController
  - Controlador mais básico que provem uma implementação abstrata da interface Controller.
- UrlFilenameViewController
  - Somente a view é necessária.
  - O model não é usado.
- ParameterizableViewController
  - Nome da view fornecida com propriedades (parêmetros) definidos em um arquivo a parte (bean);

Principais classes Controller

- MultiActionController
  - Diversas ações tratadas (Ex: comprar, vender, etc).
    - public ModelAndView comprar(HttpServletRequest request)
    - public ModelAndView vender(HttpServletRequest request)
- SimpleFormController
  - A página web possui algum formulário.
  - Formulário tem que ser preenchido com informações;
  - Formulário é alterado tendo que ser salvo.

Data Binding


```
### Modelo ####
Person{

private String nome;
public String getName() { return this.name; }
public void setName( String name ) { this.name = name }
...

}

### View (*.jsp) ####
person.nome
array[0].nome
array[“pessoaX”].nome
```

Testar controladores

- Classes Mock no framework
  - spring-mock.jar
- Site do mock
  - MockObjects.com