UNIPÊ
Pós-graduação

# Desenvolvimento Web Frameworks

MSc. Moacir Lopes de Mendonça Junior

# Hibernate Validator

- Em qualquer tipo de aplicação, receber dados sempre foi um problema.
  - Não é possível prever o que se passa na cabeça do usuário ou a interpretação que ele faz da interface da aplicação.

- É comum usuários digitarem números ou caracteres especiais em campos que não estão preparados para receber esse tipo de entrada.

# Hibernate Validator

- Se falarmos de formatos mais específicos, como datas, por exemplo, esse problema fica ainda mais evidente.

- Dessa forma, se não for feito o tratamento adequado por parte das camadas da aplicação, problemas podem ocorrer

- O framework Hibernate Validator, é uma alternativa para a validação de dados em qualquer arquitetura (Web, Desktop, etc.)

# Hibernate Validator

- O Validator permite expressar as regras de maneira padronizada através do uso de anotações que representem a restrições desejadas.

- Como o Validator expressa as regras e restrições da aplicação por meio de anotações, a implementação das validações é transparente

# Hibernate Validator

- Algumas anotações
  - @max
  - @min
  - @notempty
  - @notnull
  - @size
  - @email
  - @past
  - @future
  - @assertfalse
  - @asserttrue

# Capturando Mensagens de Erro

- Vamos usar o Hibernate Validator para validar o objeto

```
@RequestMapping("/addAutorizacao")
private String cadastro(@Valid Autorizacao autorizacao,
BindingResult result, Model model) {
    if (result.hasErrors()) {
        return "cadastroUsuario";
    }
    autorizacaoDAO.create(autorizacao);
    return "redirect:/prepararListarAutorizacao";
}
```

# Integração com Hibernate/JPA

- Em aplicações que necessitam de acesso a informações persistidas em banco de dados, é comum utilizar o controle de transações oferecido pelo Spring em sua forma declarativa

- Este controle pode ser conseguido através do uso da anotação @Transactional em conjunto com as configurações adequadas na execução do container Spring

# Integração com Hibernate/JPA

- Classes e/ou métodos de beans anotados com @Transactional recebem o comportamento transacional, definindo o limite existencial de uma transação

- A anotação @Transactional geralmente é aplicada nas classes que requererem a recuperação e/ou alteração de dados persistidos

# Configurando no Application Context

- É necessário adicionar o schema no spring-tx, como também adicionar o schema na propriedade schemaLocation

```
xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
```

# Configurando no Application Context

- Vamos configurar o entitymanager

- No persistence.xml comentamos as propriedades referentes a conexão com a base de dados

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="PersistenceUnit" />
    <property name="dataSource" ref="postgresDataSource" />
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
</bean>

<bean id="postgresDataSource" class="org.apache.commons.dbcp2.BasicDataSource">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="url"
        value="jdbc:postgresql://localhost:5432/advogados" />
    <property name="username" value="postgres" />
    <property name="password" value="mv13wavaty" />
</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>

<tx:annotation-driven />
```

# Configurando os DAOs

- Nos DAOs onde utilizaremos o EntityManager colocamos o seguinte atributo com a anotação @PersistenceContext que permitirá ao spring injetar a instancia do objeto

```
@PersistenceContext
protected EntityManager entityManager;
```

# Configurando os DAOs

- Já na classe DAO colocamos a anotação @Repository

- Nos métodos onde ocorre as operações de persistência colocaremos a anotação @Transactional

# Agendamento de Tarefas

- A funcionalidade do agendamento de tarefas é muito útil para aproveitar horas de menor tráfego da rede
  - na geração de relatórios,
  - na replicação de dados,
  - na geração de dados para fechamento de períodos fiscais,
  - na atualização de bases de dados, etc

- É possível cuidar do agendamento (scheduling) de métodos, seja em momentos específicos, seja em determinados intervalos de tempo.

# Agendamento de Tarefas

- Adicionar no application-context.xml o schema:

```
xmlns:task="http://www.springframework.org/schema/task"
```

- Como também o schema location:

```
http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task-4.0.xsd
```

# Agendamento de Tarefas

- Adicionar a tag task:scheduler para que o spring mapeie os métodos com a anotação @Schedule

```
<task:scheduler id="taskScheduler" pool-size="6" />
```

# Spring Security

- O Spring Security foi lançado em 2003 com o propósito de incorporar recursos avançados de segurança à plataforma Java EE.

- O Spring Security trabalha a segurança através de declarações baseadas em papéis (roles).

- O Spring Security não necessita chamar método algum para realizar uma autenticação ou autorização.

# Spring Security

- Através dos roles definidos, podemos informar ao aplicativo em questão, ao qual está sendo assegurada uma área, quais recursos podem ser acessados ou restringidos a uma determinada pessoa que acessou a área restrita.

# Configuração do Spring Security

- É necessário adicionar o schema no application-context.xml

```
xmlns:sec="http://www.springframework.org/schema/security"
```

- Como também o schemaLocation

```
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-4.0.xsd"
```

# Configuração do Spring Security

- Para configurar o Spring Security, utilizamos o filtro DelegatingFilterProxy, devidamente configurado na figura abaixo.

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

# Configuração do Spring Security

```
<sec:http auto-config="true">
    <sec:csrf disabled="true" />
    <sec:intercept-url pattern="/index.jsp" access="isAuthenticated()" />
    <sec:form-login login-page="/login.jsp"
        default-target-url="/index.jsp" login-processing-url="/j_spring_security_check"
        authentication-failure-url="/login.jsp?erro=true" />
</sec:http>
<sec:authentication-manager>
    <sec:authentication-provider>
        <sec:user-service>
            <sec:user name="administrador" authorities="ROLE_USER"
                password="123456" />
        </sec:user-service>
    </sec:authentication-provider>
</sec:authentication-manager>
```

# Autenticação

- A o form da página deve apontar para o servlet do Spring security

```
<form action="j_spring_security_check" method="post">
    Usuário:<input id="username" name="username" type="text" /><br />
    Senha:<input id="password" name="password" type="password" /><br />
    <input type="submit" value="Enviar"/>
</form>
```

# Logout

- Adicionar no application-context

```
<sec:logout invalidate-session="true" logout-url="/j_spring_security_logout"/>
```

# Integrando com o Banco

- Exemplo

```
<sec:authentication-manager>
    <sec:authentication-provider>
        <sec:jdbc-user-service data-source-ref="dataSource"
            users-by-username-query="SELECT u.username as username, u.password as password,
            enable FROM Usuario u WHERE u.username=?"
            authorities-by-username-query="SELECT u.username as username, 'ROLE_' + a.nome as authority
            FROM Autorizacao a, Usuario u WHERE u.username=?
            and u.autorizacao_id = a.id"/>
    </sec:authentication-provider>
</sec:authentication-manager>
```

# Programação Orientada a Aspecto

- O conceito foi criado pela equipe da **Xerox PARC**, a divisão de pesquisa da **Xerox**. Eles desenvolveram o **AspectJ**, a primeira e mais popular linguagem POA.

- A programação orientada a aspectos surgiu a partir da identificação de uma dificuldade na programação orientada a objetos em modularizar certos tipos de interesses.

# Programação Orientada a Aspecto

- **Programação orientada a aspectos** (Aspect-Oriented Programming – AOP), é um paradigma de programação que separa e organiza o código de acordo com a sua importância para a aplicação

- Todo o programa escrito **orientado a objetos** possui um código que é alheio a implementação do comportamento do objeto.

- Este código implementa as funcionalidades e suas chamadas encontram-se espalhadas por toda a aplicação

# Programação Orientada a Aspecto

- Por exemplo, a funcionalidade de log de dados, numa linguagem orientada a objetos, é implementada em uma única classe, que é referenciada em todos os pontos onde é necessário fazer log de dados.

- Como praticamente todo método necessita que alguns dados sejam registrados em log, as chamadas a essa classe são espalhadas por toda a aplicação.

# Programação Orientada a Aspecto

- Tipicamente uma implementação da AOP busca encapsular essas chamadas através de uma nova construção chamada de "aspecto".

- Um aspecto pode alterar o comportamento de um código pela aplicação de um comportamento adicional, ***advice***, sobre um "ponto de execução", ou ***join point***.
  - A descrição lógica de um conjunto de ***join points*** é chamada de ***pointcut***.

# Programação Orientada a Aspecto

- Alguns conceitos introduzidos por esta abordagem:
  - Aspect: É a unidade modular que encapsula um interesse que atravessa vários objetos dentro do sistema
  - Join Point: É um ponto durante a execução do programa que será afetado pelo aspecto
  - Advice: É a ação executada pelo aspecto em um join point particular. Esta ação pode executar antes, após ou mesmo envolver o join point para decidir se o mesmo deve realmente ser executado

# Programação Orientada a Aspecto

- Pointcut: Ele permite que o aspecto saiba quais são os pontos na execução do programa em que um advice deve executar.
- Objeto Alvo: É o objeto que está sendo interceptado por um ou mais aspectos

# Programação Orientada a Aspecto

- Aqui estão os tipos de advice suportados pelo AspectJ:
  - Before - Advice que executa antes do join point
  - After returning - Advice que executa após o join point
  - After throwing - Advice que executa após o join point
  - After (finally) - Advice que executa após o join point
  - Around - Advice que envolve a execução de um join point.

# Exemplo

CLASSE A

//join point
//Param – Objeto Alvo
Public void método(Param){

}

//Aspect
CLASSE B

//advice
Public void método2(){

}

# Configurando o AOP no Spring

- Como nas outras funcionalidades é necessário adicionar o schema do spring aop no application-context.xml

```
xmlns:aop="http://www.springframework.org/schema/aop"
```

- Como também adicionar o schemaLocation

```
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
```

# Configurando o AOP no Spring

- Depois é necessário também adicionar a tag abaixo para que o spring ative o AOP

```
<aop:aspectj-autoproxy />
```

- As classes a serem mapeadas deverão utilizar a tag bean para isso

```
<bean class="[classe]" />
```

# Configurando o AOP no Spring

- Anotações do Spring AOP
  - @Aspect
  - @Before
  - @After
  - @Around
- Utilizamos o parâmetro "execution(* [metodo] (..))" nas anotações para mapear o método que queremos verificar na execução
- Ao adicionar o trecho && args(entity) no parâmetro da anotação se pode obter o objeto que está sendo parametrizado no método a ser escutado.

# Obrigado