ARTIGO:

Tirando o Maximo da Struts – Parte II

(TilesRequestProcessor, Validator e ExceptionHandler)

Por: Paulo Alvim

Março/2004

Powerlogic
Tecnologia Orientada ao Cliente

(31) 3286-1691
www.powerlogic.com.br
plc@powerlogic.com.br

Índice

TIRANDO O MAXIMO DA STRUTS

(TilesRequestProcessor, Validator e ExceptionHandler)
Nível: Intermediário

Introdução

Em um artigo anterior, vimos como explorar ao máximo o potencial da Struts, utilizando apenas programação Orientada a Objetos para realizar extensões interessantes e, inclusive, recomendadas por seus autores.

Dissertamos sobre a importância de um trabalho de "última milha" sobre tecnologias Open-Source, antes do início do desenvolvimento do dia-a-dia, e dos ganhos práticos que temos obtido com o jCompany (www.powerlogic.com.br) , que é um exemplo desta abordagem.

No presente artigo, iremos continuar a explorar as oportunidades reservadas pela Struts, com foco nos seguintes *Extension-Points*: TilesRequestProcessor, Validator e ExceptionHandler.

Mais Extensões da Struts

1. TilesRequestProcessor
2. Validation
3. ExceptionHandler

1. TilesRequestProcessor

(Descendente de RequestProcessor)
Esta classe tem métodos disparados a cada requisição feita ao *servlet* de controle da Struts.

Por quê estender? Esta classe perdeu parte de sua utilidade como ponto de extensão devido ao surgimento de *filters* e *listeners* na especificação Servlet 2.3, mas continua sendo ideal para quem não possui "Application Servers" nesta versão ou para lógica que precisam ocorrer após o ServletController e antes do Action.

Exemplos de extensão: Lógicas de *profiling* (recuperação de perfil) de usuário e lógicas de *caching* de objeto *Response* (portais).

No nosso tutorial, vamos criar uma classe de especialização do *TilesRequestProcessor* contendo uma lógica genérica para *profiling* que, usando o *login* do usuário recém-autenticado no padrão J2EE, recupere informações complementares como email, empresa, etc., disponibilizando-as, em seguida, em um Value Object na sessão. O benefício evidente desta extensão é a padronização e reutilização desta lógica por toda a corporação.

Passo a passo:

1.1. Criar uma classe que estenda TilesRequestProcessor e implemente o método "processPreprocess". No nosso exemplo, esta classe se chamará PlcRequestProcessor (Para curiosos, "Plc" abrevia "Powerlogic Consultoria")

```
public class PlcRequestProcessor extends TilesRequestProcessor {

  protected boolean processPreprocess(HttpServletRequest request,
                HttpServletResponse response) {

    super.processPreprocess(request,response);

    // Conteúdo deste método será visto adiante
    executaPerfil(request);
  }
}
```

1.2. Declarar, no arquivo de configuração da Struts (ex:"struts-config.xml"), elemento "controller", que pretendemos utilizar nossa classe especializada como "processadora de requisições" (No exemplo, o pacote do jCompany foi utilizado. Os outros parâmetros também são ilustrativos, podendo ser alterados ou retirados):

```
<controller  contentType="text/html;charset=UTF-8"  maxFileSize="2M" nocache="true"
processorClass="com.powerlogic.jcompany.controle.PlcRequestProcessor"
locale="true"></controller>
```

1.3. Implementar o método específico da lógica de registro de perfil. Este método executará a cada requisição, mas somente registrará o perfil do usuário uma vez, após o *login* (A forma de recuperação não é detalhada, o importante é a percepção da necessidade de um método que ocorra **antes** de qualquer Action, mas **após** o ServletController, para permitir o uso do parâmetro HttpServletRequest):

```
...
private void executaPerfil(HttpServletRequest request) {

if (request.getUserPrincipal() != null &&
 request.getSession().getAttribute(PlcConstantes.USER_KEY) == null) {
  // Pega interface Façade registrada em escopo de aplicação
  IPlcFacade plc = (IPlcFacade) request.getSession().getServletContext().
getAttribute(PlcConstantes.INTERFACE_PERSISTENCIA_KEY);

  // Recupera os complementos chamando a camada Modelo
  UsuarioVO usu = plc.recuperaPerfil(request.getUserPrincipal().getName());

  // Coloca na sessão o VO com todas as informações complementares
  request.getSession().setAttribute(PlcConstantes.USER_KEY,usu);
 }
}
...
```

2. Validation

A framework de validação que acompanha a Struts permite que eliminemos programação Java no dia a dia para averiguação da consistência dos dados informados pelos usuários. A Struts já oferece validações prontas para verificação de obrigatoriedade, tipos de campos (String, Long, etc.), formatos válidos de emails, cartões de crédito, tamanhos mínimos e máximos, datas válidas, máscaras e algumas outras. Porém, carece de importante suporte para validação entre campos, envolvendo "modos" (ex: obrigatoriedade somente na alteração), dentre outros.

Por quê estender? Para complementar as regras de validação do Validator de forma e eliminar quase que por completo a necessidade de código Java para este tipo de programação.

Exemplos de extensão: Lógicas de validação complementares que permitam regras declarativas para validação entre campos, "modos" (obrigatório somente na inclusão ou alteração), que consideram 0 (zero) valor não significativo (não informado), validação de detalhes em lógicas "Master-Detail", além de CGC, CPF e validações específicas da organização. No projeto jCompany, praticamente toda a programação de validação de entrada foi eliminada com estas extensões.

No nosso tutorial, vamos criar uma nova regra de validação similar à funcionalidade da validação "required" da Struts, que obriga ao usuário informar um campo, mas no nosso caso queremos considerar "0" (zero) como valor não

significativo, ou seja, como sinônimo de "não informado". Essa variação é muito útil em diversos cenários onde o valor zero não faz sentido.

Apesar de simples, este caso serve para assimilarmos todo o mecanismo de extensão do Validator.

Passo a passo:

2.1. Criar uma classe que conterá nossas regras de validação e implementar o método de validação no padrão do exemplo abaixo. De acordo com o mecanismo do Validator, essa classe não precisa especializar nenhuma outra e nem implementar nenhuma *Interface*.

```
public class PlcValida implements Serializable {

  // 1
  public static boolean validateRequeridoZero (Object bean,
        ValidatorAction va, Field field,
        ActionErrors errors, HttpServletRequest request) {

        String valorInformado = "";

       if (field.getProperty() != null &&
          field.getProperty().length() > 0) {
         // 2
          valorInformado = ValidatorUtil.
                              getValueAsString(bean,field.getProperty());
        }

    // 3
     if (valorInformado == null ||
        valorInformado.trim().equals("") ||
        valorPrincipalInformado.equals("0")) {
            errors.add(field.getKey(), Resources.getActionError(request,va,field));
            return false;
      }
      return true;
    }

}
```

1. O nome do método deve iniciar com "validate", seguido do nome da regra de validação, e a assinatura deve ser exatamente a exemplificada acima.

2. Nesta linha, o valor informado na propriedade sendo validada é recuperado.

3. Finalmente, a validação é realizada, retornando "false" caso um valor válido não tenha sido encontrado. Obs: somente a cláusula "valorInformado.equals('0')" difere este método da validação "required" padrão.

2.2. Declarar a nova regra no arquivo "validator-rules.xml". Este arquivo cataloga, para o Validator, quais são as regras disponíveis. O XML define o nome da regra (como será utilizado pelos desenvolvedores no dia a dia), a classe de validação e nome do método a ser chamado, bem como parâmetros para o método e a mensagem de erro a ser exibida (no caso, mantivemos a mensagem padrão da Struts para validação de *required*).

```
...
<validator name="requeridoZero"
          classname="com.powerlogic.jcompany.controle.PlcValida"
            method="validateRequeridoZero"
                        methodParams="java.lang.Object,
                    org.apache.commons.validator.ValidatorAction,
                    org.apache.commons.validator.Field,
                    org.apache.struts.action.ActionErrors,
                    javax.servlet.http.HttpServletRequest"
            msg="errors.required">
</validator>
...
```

Importante: O ideal aqui é criar um novo arquivo de regras (no nosso caso chamamos de "plc-validator-rules.xml") para evitar conflitos com novas versões de validações da Struts. Para que a Struts reconheça este arquivo, basta declará-lo na parte de plug-ins do "struts-config.xml"

```
...
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
 <set-property property="pathnames" value="/WEB-INF/validator-rules.xml, /WEB-
INF/plc-validator-rules.xml, /WEB-INF/validation.xml" />
</plug-in>
...
```

2.3. No dia a dia, o desenvolvedor pode utilizar a regra declarando, para cada "action" ou "Form-Bean", as validações no arquivo "validation.xml" (praticando reusabilidade de forma elegante, como mandam as boas práticas!).

```
...
  <form name="/cliente">
        <field property="nome"
              depends="required">
              <arg0 key="label.nome"/>
        </field>
        <field property="areaInteresse"
              depends="requeridoZero">
              <arg0 key="label.area.interesse" />
        </field>
...
```

No caso do jCompany, as validações são exibidas como no exemplo da figura 1.

Figura 1. Mensagens de validação declaradas em XML sendo exibidas.

3. ExceptionHandler

A Struts traz uma arquitetura de tratamento de exceções bem elaborada, que também propicia a possibilidade de eliminação de programação excessiva para estes casos, além de uma padronização. Para este fim, ela também provê uma classe que deve ser especializada.

Por quê estender? Para centralizar e prover lógicas genéricas de tratamento de exceções que garantam uma "pauta mínima" de tratamento, diminuindo a dependência de conhecimento dos programadores.

Exemplos de extensão: No jCompany, toda exceção disparada recebe o seguinte tratamento mínimo:
1. Exibição para os usuários uma mensagem elegante e a mensagem original (java null pointer, por exemplo), sendo esta última em cor diferente.
2. Envio de *stack-trace* do erro para a console e para arquivo.
3. Envio do *stack-trace* também por email para uma lista de responsáveis, via um appender JMS do Log4j.

No nosso tutorial, vamos criar uma classe que centraliza o tratamento de exceções testando o tipo de exceção disparada e fazendo um tratamento mínimo em conformidade com este tipo. Exemplos de tratamentos mínimos são o envio de uma mensagem de *logging* e a exibição de mensagens para usuários incluindo, para casos inesperados, uma mensagem "elegante" e outra com a causa raiz do problema. (Ver figura 2)

Passo a passo:

3.1. Criar uma classe específica de exceção, que deverá ser disparada caso o desenvolvedor trate exceções de forma específica. Os pontos importantes a observar são que a classe deve estender *Exception* e possuir construtores que recebam a exceção original, a chave da mensagem base de erro (internacionalizada) e um *array* para argumentos para serem utilizados na mensagem, no padrão de mensagens da Struts.

```
public class PlcException extends Exception {

        protected Throwable causaRaiz = null;
        private String messageKey = null;
        private Object[] messageArgs = null;

   public PlcException(){ super(); }

   public PlcException(String messageKeyLoc,
                       Throwable causa, Object[] messageArgsLoc)  {
          setMessageKey(messageKeyLoc);
          setCausaRaiz(causa);
          setMessageArgs(messageArgsLoc);  }

        public PlcException(Throwable causa ) {
                this.causaRaiz = causa;         }

        public PlcException(String novaMessageKey) {
                this.messageKey = novaMessageKey;  }

        public void setMessageKey( String key ) {
                this.messageKey = key;         }
```

```
        public String getMessageKey () {
                return messageKey;    }

        public void setMessageArgs( Object[] args ) {
                this.messageArgs = args;       }

        public Object[] getMessageArgs() {
                return messageArgs;   }

        public void setCausaRaiz(Throwable anException) {
                causaRaiz = anException;       }

        public Throwable getCausaRaiz() {
                return causaRaiz;       }

}
```

3.2. No dia a dia, o desenvolvedor que precisa tratar uma exceção qualquer (tipicamente, na camada Modelo), deve faze-lo e, em seguida, disparar uma exceção do tipo PlcException, passando a mensagem e exceção original como parâmetros, como no exemplo abaixo:

```
public void recuperaSaldoCliente(Session sess, Long idCliente) throws PlcException  {

 try {

        List l = recuperaLista(sess,"from obj in class"+
                " com.empresa.app.vo.SaldoCliente where obj.idCliente=?",
                new Object[] {idCliente},
                new Type[]{Hibernate.LONG},"");

        return l;

 } catch (HibernateException e2) {
     log.fatal("Erro de persistência ao recuperar saldo: " + e2);
    throw new PlcException("app.erros.saldo",new Object[] {e2},e2);
 } catch( Exception ex ){
     log.error("Erro inesperado ao tentar recuperar saldo:"+ex);
     throw new PlcException("app.erros.saldo",new Object[] {ex},ex);
 }

}
```

No exemplo acima, a diferença de tratamento foi apenas uma diferenciação de níveis de *log* entre uma exceção da Hibernate e outras, mas o tratamento específico poderia conter qualquer lógica necessária.

3.3. Implementar uma classe para tratamento genérico de exceções no padrão da Struts. Esta classe deve estender *ExceptionHandler* da Struts e implementar o método "execute", conforme nosso exemplo:

```
public class PlcExceptionHandler extends ExceptionHandler {

   public ActionForward execute (Exception ex, ExceptionConfig config,
         ActionMapping mapping, ActionForm formInstance,
         HttpServletRequest request, HttpServletResponse response)
         throws ServletException {
         // Exemplo de código abaixo
    }
}
```

3.4. No corpo do método, fazemos o *casting* da exceção para o nosso tipo (já que todas as exceções são "transformadas" em PlcException pelos desenvolvedores) e procedemos tratamentos padrões diferenciados, conforme a causa raiz.

```
...
   public ActionForward execute (Exception ex, ExceptionConfig config,
          ActionMapping mapping, ActionForm formInstance,
          HttpServletRequest request, HttpServletResponse response)
          throws ServletException {

        // 1
         if (ex instanceof PlcException) {

            // 2
            PlcException plcException = (PlcException) ex;
            Exception causaRaiz = null;

            if (plcException.getCausaRaiz() != null) {
               causaRaiz = (Exception) plcException.getCausaRaiz();

              // 3
              if (causaRaiz instanceof HibernateException) {

                // Tratamento genérico para exceções da
                // camada Modelo com Hibernate
                //(inclui JDBCException)
                tratamentoGeralModelo(request,  (HibernateException) causaRaiz);

               } else if (causaRaiz instanceof InvocationTargetException)  {

                // Tratamento genérico para exceções causadas
                // por chamadas com reflexão. Precisa pegar
                // exceção usando getTargetException().
                 tratamentoGeral(request,
                      ((InvocationTargetException)
                      causaRaiz). getTargetException());
              }

            } else
              // Então é exceção controlada, devendo
              // somente exibir mensagem
              tratamentoGeralControlado(request,  plcException);
            }

        } else {
            log.error("Faltou tratar a excecao "+ex+" conforme padrao");
            tratamentoGeral(request,ex);
         }
    }
...
```

1. Primeiramente, testamos se a exceção foi tratada na forma padrão pelo desenvolvedor, caso contrário desviamos ("else") para um tratamento genérico de Exception, exibindo um *log* de ERROR para alertar o desenvolvedor para que faça o tratamento.

2. Em seguida, transformamos a exceção para o nosso tipo padrão estendido (PlcException) e procuramos pela causa raiz. Aqui seguimos uma regra:
   - Se o desenvolvedor criou PlcException utilizando o construtor que inclui uma exceção raiz, então fazemos tratamentos diversos conforme o tipo

desta exceção.
- Se o desenvolvedor criou PlcException utilizando o construtor somente com uma mensagem, então somente exibimos esta mensagem no padrão da Struts, já que esta é uma exceção controlada.

3. Nesta cadeia de "ifs", tratamos cada exceção raiz diferentemente, conforme seu tipo. No exemplo, exceções do tipo *HibernateException* (a Hibernate encapsula exceções de JDBC, inclusive) são tratadas pelo método "tratamentoGeralModelo" e exceções do tipo *Java.lang.reflect.InvocationTargetException (erros provacados dentro de chamadas de reflexão)* têm a causa original tratada através do uso de "getTargetException", e assim por diante.
Importante: Não é foco aqui explorar padrões de tratamento de exceção Java, mas somente mostrar a arquitetura para implementarmos o padrão da Struts. Portanto, os métodos de tratamento em si não são detalhados.

3.5. Declarar, na Struts, para que nossa classe *PlcExceptionHandler* seja utilizada para tratamento de exceções em geral. Para isso, devemos alterar o elemento "global-exceptions", conforme abaixo.

```
<global-exceptions>
  <exception
   handler="com.powerlogic.jcompany.comuns.PlcExceptionHandler"
   key="jcompany.erros"
   scope="request"
   type="java.lang.Exception">
   </exception>
</global-exceptions>
```

Com isso, todas as exceções disparadas pela camada de controle (Actions) passam a ser tratadas pela nossa classe genérica. Como a prática é tratar exceções o mais "cedo" possível (ou seja, na camada modelo), a tendência é sermos dispensados, na maior parte dos casos, de "try – catch" nos métodos de controle!

Figura 2. Exceção de integridade referencial tratada genericamente.

Time-to-Market e Open-Source

Após compreendermos todas as vantagens obtidas com as técnicas de última milha citadas, vem a pergunta: Quanto tempo e investimento são necessários para integrarmos e especializarmos a Struts, o Tiles, a Hibernate, o Log4j, o Eclipse, JasperReports, o Junit, etc., para obtermos um patamar realmente diferenciado e produtivo de desenvolvimento com soluções Open-Source?

Na nossa experiência, já contabilizamos 20 meses de intensa pesquisa, programação, *refactorings* e integrações, com uma equipe variando entre 2 a 4 pessoas. Porém, a cada evolução obtida, o retorno tem sido compensador!

Com tudo isso, uma boa opção para empresas que desejem "tirar o máximo" de benefício do mundo Open-Source, e ainda obter um bom "*time-to-market*", seria utilizar uma *framework* como o jCompany. Neste caso, esta *framework* atuaria como uma solução de "penúltima milha", já que uma extensão de "última milha" ficaria a cargo da referida empresa.

Nesta opção, 90% do trabalho de extensão estaria na penúltima milha, e somente 10% restaria para decisões específicas de cada organização.

Conclusão

Com este e o último artigo, esperamos ter contribuído com caminhos de produtividade interessantes no uso da Struts. Em próximas oportunidades, iremos explorar trabalhos de última milha que fizemos em outras *frameworks,* com foco no V e M do MVC.

Em "Tirando o Máximo do Tiles", veremos como tornar o *Browser* uma Interface com o Usuário de alto nível (multi-pele, multi-layout, multi-lingual), e que rivaliza com interfaces de *desktop*. E em "Tirando o Máximo da Hibernate", como fazer persistência performática, portável e escalável, sem as rédeas e gorduras dos EJBs!

Bibliografia Rápida

- Struts Fast Track by Vic Cekvenich
- Programming Jakarta Struts by Chuck Cavaness
- Jakarta Pitfalls by Bill Diudney, Jonathan Lehr
- Struts in Action by Ted Husted, Cedric Dumoulin
- Struts Survival Guide by Skikanth Shenoy, Nithin Mallya
- Professional Struts Applications by John Carnell, Jeff Linwood
- Patterns of Enterprise Application Architecture by Martin Fowler
- **http://jakarta.apache.org/struts**

Autor

- Paulo Alvim é Diretor de Tecnologia da Powerlogic Consultoria e Sistemas S.A., e responsável pelo projeto jCompany. Pode ser contatado pelo email **alvim@powerlogic.com.br**