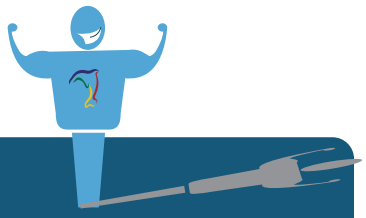
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

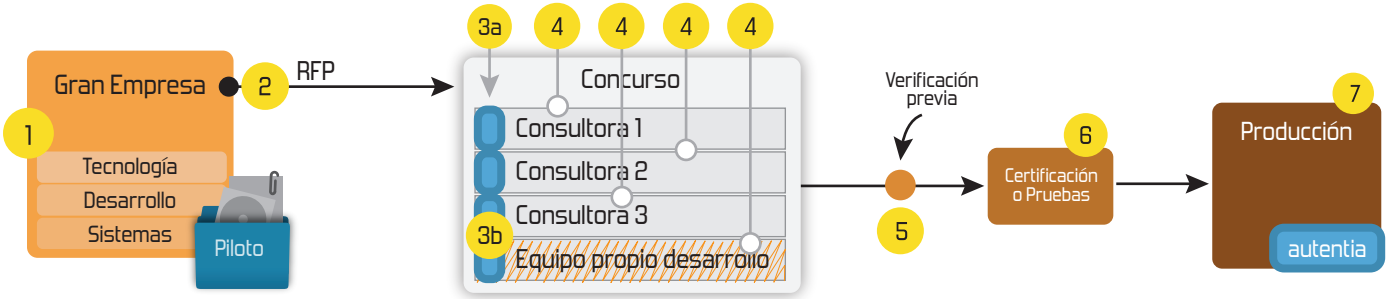
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

AdictosAlTrabajo
Terrakas 1x05 ¡¡Ya está en la web!! ;-)
terrakas.com
autentia Soporte a desarrollo informático
Hosting patrocinado por enredados

Entra en Adictos a través de
E-mail
Contraseña
Entrar
Deseo registrarme
Olvidé mi contraseña

Inicio Quiénes somos Formación Comparador de salarios Nuestro libro Más

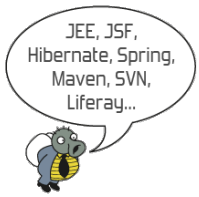
» Estás en: Inicio Tutoriales Acciones JSF basadas en clases internas de Java



Cristóbal González AlmirónConsultor de desarrollo de proyectos informáticos. Su experiencia profesional se ha desarrollado en empresas como Compaq, HP, Mapfre, Endesa, Repsol, Universidad Autónoma de Madrid, en las áreas de Desarrollo de Software (Orientado a Objetos), tecnologías de Internet, Técnica de Sistemas de alta disponibilidad y formación a usuarios.

Ver todos los tutoriales del autor

Catálogo de servicios Autentia



Fecha de publicación del tutorial: 2009-02-26 Tutorial visitado 2 veces Descargar en PDF

Acciones JSF basadas en clases internas de Java.

1. Resumen
2. Introducción
3. Acciones simples basadas en clases internas
4. Introducción de captura de excepciones en nuestro ActionMethod
5. Reescritura del Bean de Login usando la AccionBase con control de excepciones
6. Añadiendo flexibilidad y acceso a datos al ActionMethod
7. Conclusión

1. Resumen

En este tutorial aprenderemos a crear nuestros backbeans para Java Server Faces implementando las acciones de los beans como clases internas de Java. Esto nos facilitará tanto la escritura de las mismas como la claridad con la que quedan reflejadas las funciones que se realizan en cada acción.

2. Introducción

Cuando desarrollamos aplicaciones JSF gran parte de la lógica de interfaz y negocio se desarrolla dentro de las acciones de los BackBeans, las clases Java asociadas a la página JSF que está sirviendo la petición actual. Por ejemplo, cuando el usuario pulsa el botón de "Inicio" en la página de inicio de sesión, el controlador de JSF invocará a la acción asociada al botón de inicio de esta página. Dicha acción será normalmente un método del backbean de la página de login.

Un ejemplo de código para la página y el backbean podría ser:

```
<%@ page contentType="text/html; charset=Cp1252" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=Cp1252"/>
        <title>Login
    </head>
    <body>
        <f:view>
            <h:form>
                UserID: <h:inputtext value="#{login.username}">
                <br/>Password: <h:inputtext value="#{login.password}">
                <br/><h:commandbutton value="Login" action="#{login.loginAction}">
            </h:form>
        </f:view>
    </body>
</html>

</h:commandbutton></h:inputtext></h:inputtext>
```

Y su backbean:

```
// SimpleLoginBean.java
package samples;

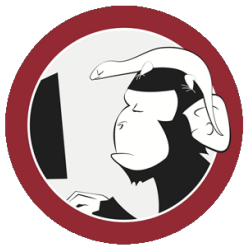
import java.io.Serializable;

public class SimpleLoginBean implements Serializable {
    private static final long serialVersionUID = 1L;

    public final static String OK = "OK";
    public final static String ERROR = "ERROR";

    private String username;
    private String password;

```



Síguenos a través de:

Últimas Noticias

» ¿Qué es aportar valor como técnico/programador?
» Como arrancar un nuevo proyecto e integrar el agilismo en una organización
» Hoy es el primer día para cambiar tu sector
» AdictosAlTrabajo os desea ¡¡¡FELICES FIESTAS!!!
» ESTRENO de Terrakas 1x05: "Un lugar en el mundo"
Histórico de noticias

Últimos Tutoriales

» Nos han cambiado el agua en el sector informático
» Sonar y Javascript: obteniendo la cobertura de nuestro código
» ZKPushState: Manejar el API de Historial de navegación en HTML5 directamente desde Java
» Modela tu mercado en base a la demanda y no a la oferta.
» Mirar el todo: llevar el agilismo a las grandes

```
    private boolean logged;

    public SimpleLoginBean() {

    }

    // ------- Actions ---------------------

    /** very simple login action */
    public String login() {

        if (!"user".equals(username))
            return ERROR;
        if (!"sesamo".equals(password))
            return ERROR;

        logged = true;
        return OK;
    }

    // ------- Getters and setters ---------
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public boolean isLogged() {
        return logged;
    }

    public void setLogged(boolean logged) {
        this.logged = logged;
    }


}
```

En este ejemplo la comprobación de usuario y contraseña es muy simple. Lo normal es que en una acción se mezclen varios tipos defunciones:

1. Verificación de las precondiciones para la ejecución de la acción.
2. Validación con lógica de negocio de los datos de entrada. No siempre las validaciones de los datos de entrada son simples (comprobaciones de formato, de tamaños, etc.) En aplicaciones complejas la validación de los datos puede involucrar realizar llamadas a diferentes partes de la lógica de negocio.
3. Ejecución de lógicas de negocio relativas a la acción.
4. Comprobaciones de resultados de la lógica de negocio.
5. Actualización de los datos de la interfaz de usuario y del estado de la sesión de aplicación.
6. Control de las excepciones y errores, con inclusión de mensajes de error o de confirmación en la página del usuario.
7. Selección del flujo de salida para esta página.

Todo ello se adereza con las clásicas funciones auxiliares, como pueden ser:

- Inserción de trazas de información, depuración y error.
- Comprobaciones de seguridad (credenciales de usuario y otras).
- Registro de acciones de usuario en tablas de auditoría, etc.

Si aplicamos esto a una acción JSF, el código de la acción realmente se complica. Un ejemplo podría ser el siguiente: imaginemos una pantalla de una aplicación de gestión de un sistema de aprovisionamiento de servidores para un proveedor de Internet.

```
package samples;

public class LoadServerBackBean {

    /** stores the server type selected by customer */
    private String serverKind;

    final static String OK = "OK";
    final static String ERROR = "ERROR";

    // ----------- Actions ---------------

    /** creates a new server for a customer */
    public String loadServerActiion() {
        log("loadServerAction Action begin");

        DatabaseSession dbsession = null;
        UserBean user;

        try {
            user = UserBean.getUserBean();
            dbsession = DataBaseManager.getDatabaseSession();

            // validate if user can load a new server
            if (!validateUserForLoadServer(user, dbsession)) {
                insertResponseErrorMessage("loadServer.badUserValidation");
                log("loadServerActiion - User validation error");
                return "ERROR";
            }

            // validates server data
            if (!validateServerData()) {
                insertResponseErrorMessage("loadServer.badUserValidation");
                log("loadServerActiion - data server validation error");
```

organizaciones.

Últimos Tutoriales del Autor

» Rendimiento en espacio y transferencia de un servidor Subversion

» Instalación de subversion

» Gestión de los Requisitos (y II) : los casos de uso

» Gestión de los requisitos

» Introducción a JSF Java

Últimas ofertas de empleo

2011-09-08
Comercial - Ventas - MADRID.

2011-09-03
Comercial - Ventas - VALENCIA.

2011-08-19
Comercial - Compras - ALICANTE.

2011-07-12
Otras Sin catalogar - MADRID.

2011-07-06
Otras Sin catalogar - LUGO.

```
                return "ERROR";
            }

            // Creates the new Server and preload servides for that server
            // (DNS, IP, email, etc)
            if (!createNewServer()) {
                insertResponseErrorMessage("loadServer.badUserValidation");
                log("loadServerActiion - Error creating a new Server");
                return "ERROR";
            }

            // updates Servers information for that user
            if (!upadateUserServers()) {
                insertResponseErrorMessage("loadServer.badUserValidation");
                log("loadServerActiion - Error creating a new Server");
                return "ERROR";
            }

            insertResponseErrorMessage("loadServer.badUserValidation");

            dbsession.commitTransaction();

        } catch (DatabaseException e) {
            logError("DatabaseException in loadServerActiion", e);
            insertResponseErrorMessage("loadServer.databaseError");
            return ERROR;
        } finally {
            if (dbsession.isOpen())
                dbsession.rollbackTransaction();
        }

        insertResponseErrorMessage("loadServer.loadSuccessfull");
        log("loadServerAction Action begin");
        return OK;
    }
...
```

Si nos fijamos en el código, gran parte del mismo son líneas que vamos a repetir de manera habitual cuando creemos las acciones de nuestra aplicación, para acciones similares a ésta. Sería más interesante poder reescribir este código de manera que fuese más sencillo y fácil de escribir y mantener. A estas alturas seguro que a todos ya se os han ocurrido varias formas de hacerlo...

3. Acciones simples basadas en clases internas

Bien, voy a proponer una forma sencilla de crear nuestras acciones utilizando las clases internas de Java. Las clases internas de Java nos dan algunas ventajas que nos van a venir bien a la hora de crear métodos para las acciones: Una clase interna siempre se crea dentro del contexto de una instancia de la clase contenedora, por lo que tiene acceso a los atributos privados de la instancia de la clase contenedora.

- Podemos crear una instancia de nuestra clase interna dentro de un método de la clase contenedora.
- Si la clase interna no tiene atributos el coste en demora para crear dicha instancia es pequeño.
- Normalmente vamos a trabajar sobre los métodos de la clase interna y los datos estarán en la instancia de la clase contenedora, por lo que el uso de memoria será pequeño.
- Además las clases internas se pueden tomar como clase base para nuevas clases internas en las clases derivadas de la clase contenedora.

Pongamos un ejemplo sencillo para ilustrar esta técnica:

```
    // ActionBaseBean.java
package samples;

public class SimpleActionBaseBean {


    public abstract class ActionMethod {
      String result;

      public String execute() {
        log("Execute begin");
        try {
          executeAction();
          log("Action OK");
          result = "OK";

        } catch (Exception e) {
          log("Exception in action" + e.getMessage());
          result =  "ERROR";
        }

        log("Execute end");
        return result;
      }
      protected abstract String executeAction();
  }

  // ------------- ACtion utility methods --------------

  private void log(String msg) {
    //TODO: do logging
  }

}
```

Veamos cómo podría quedar una acción de un bean de login implementada con esta clase:

```
// LoginBean.java
package samples;

import java.io.Serializable;

public class LoginBean extends SimpleActionBaseBean implements Serializable {

    private String username;
    private String password;

    public class ActionLogin extends ActionMethod {
```

```
12
13          protected String executeAction() {
14
15              if (!username.equals("admin"))
16                  return "ERROR";
17              else if (!password.equals("sesamo"))
18                  return "ERROR";
19
20              return "OK";
21          }
22      }
23
24      // public action method for this bean
25      public String login() {
26          return new ActionLogin().execute();
27      }
28
29  }
```

Como vemos la acción del bean de login sólo tiene código necesario para la lógica de negocio, careciendo de código adicional para la gestión de las excepciones o para la emisión de trazas.

Cuando se ejecuta el método de login de este bean dentro del contexto de una petición JSF, se crea una instancia de la clase interna ActionLogin dentro del método login() del bean, y esta es la responsable de gestionar tanto las excepciones como el resultado devuelto por la acción para los casos en los que hay problemas. Además la acción de login se está apoyando en los métodos definidos dentro de la clase base, por lo que los métodos de utilidad comunes quedan agrupados en la clase base.

En nuestro ejemplo hemos definido el método executeAction() como abstracto. Esto obligará a implementarlo cada vez que creemos una nueva acción dentro del bean

4. Introducción de captura de excepciones en nuestro ActionMethod

Vamos a dar un paso adicional en nuestra definición del ActionMethod. Para ello le vamos a añadir un cierto control sobre las excepciones que se generan. Ahora el método executeAction() que hay que sobrescribir en nuestras acciones puede lanzar excepciones de tipo ActionException.

```
1   // ############# ActionBase.java
2   package samples;
3
4   /**
5    * Base class for actions
6    *
7    *
8    */
9   public class ActionBase {
10
11      // @ActionResult
12      public final static String ACTION_OK = "OK";
13      // @ActionResult
14      public final static String ACTION_ERROR = "ERROR";
15
16      public abstract class SimpleActionMethod {
17
18          public String execute() {
19              String result = ACTION_OK;
20              log(">> Action method begin.");
21
22              // add more generic stuff here
23
24              try {
25
26                  result = executeAction();
27
28              } catch (ActionException e) {
29                  log("Error ActionMethod.execute() action exception: ", e);
30                  result = ACTION_ERROR;
31              } finally {
32                  log(">> Action method end.");
33                  // add finally stuff here
34              }
35
36              log(">> Action method end.");
37              return result;
38          }
39
40          protected abstract String executeAction() throws ActionException;
41
42      }
43  ...
```

Como vemos ahora durante la ejecución de nuestra acción se pueden lanzar excepciones que provocarán la navegación hacia la página de error definida en nuestro flujo de navegación. Es muy habitual que al escribir las acciones comencemos comprobando las precondiciones y realizando las validaciones oportunas para cada acción. En nuestro ejemplo de LoadServerBackbean teníamos perlas de código de tipo:

```
1   // validate if user can load a new server
2       if (!validateUserForLoadServer(user, dbsession)) {
3           insertResponseErrorMessage("loadServer.badUserValidation");
4           log("loadServerActiion - User validation error");
5           return "ERROR";
6       }
```

En nuestro bean de login vamos a tener también probablemente validaciones de tipo similar, por ejemplo validaciones de tamaños de nombres de usuario o contraseña y otras precondiciones. Ya sé que JSF proporciona mecanismos de validaciones sencillos, pero introducir las validaciones dentro de la acción le da robustez y nos dará más flexibilidad.

Vamos a añadir a nuestro ActionBase algunos métodos auxiliares del tipo que muestro en este ejemplo:

```
1   // ----------- Typical action utility methods -----
2
3       public void validateEmptyField(String field, String fieldKey)
4               throws ActionException {
5           if (field == null || field.isEmpty()) {
```

```
            insertErrorMessageField("validation.emtyField", fieldKey);
            throw new ActionException("validation.emtyField " + fieldKey);
        }
    }

    public void validateFieldMinSize(String field, String fieldKey, int size)
            throws ActionException {
        if (field == null || field.length() < size) {
            insertErrorMessageField("validation.minSizeField", fieldKey);
            throw new ActionException("validation.fieldMinSize " + fieldKey);
        }
    }

    public void validateFieldMaxSize(String field, String fieldKey, int size)
            throws ActionException {
        if (field == null || field.length() > size) {
            insertErrorMessageField("validation.maxSizeField", fieldKey);
            throw new ActionException("validation.fieldMaxSize " + fieldKey);
        }
    }

    // ------------ Utility Methods -----------------

    /** Returns the real class name for the actual instance */
    public String getClassName() {
        return this.getClass().getName();
    }

    protected void log(String message) {
        // TODO: do logginh of the message
        System.out.println(message);
    }

    protected void log(String message, Exception e) {
        // TODO: do logginh of the message
        System.out.println(message + e.getMessage());
    }

    protected void insertMessage(String messageKey) {
        // TODO: inserts a message by messageKey in the request
        System.out.println("Insert message " + messageKey);
    }

    protected void insertErrorMessage(String messageKey) {
        // TODO: inserts a message by messageKey in the request
        System.out.println("Error message " + messageKey);
    }
```

Nota: dejo el código de inserción de mensajes en el contexto JSF para otro artículo, aunque es sencillo.

Como vemos, ahora se pueden escribir validaciones muy simples llamando a métodos como validateEmptyField(String field, String fieldKey). Extos métodos simplemente comprueban las precondiciones de la acción y si no se cumplen, disparan una excepción, por lo que se provocará la navegación a la página de error.

Hemos creado:

- Una clase interna SimpleActionMethod que nos serviraá como base para implemntar nuestros métodos de acción como clases internas del bean.
- Una serie de métodos auxiliares que permitirán reescribir las funciones más comunes de la lógica de interfaz o de negocio, como pueden ser las validaciones, las trazas, la inserción de mensajes en la respuesta JSF.

5. Reescritura del Bean de Login usando la AccionBase con control de excepciones

Ahora vamos a crear un bean de ejemplo que utiliza esta clase base:

```
// ########## SambpleAction.java
package samples;

public class SampleLoginBean extends ActionBase {

    private String username;
    private String password;


    //----------  Action method loginAction ---------------

    private class LoginAction extends ActionMethod {

        @Override
        protected String executeAction() throws ActionException {

            validateFieldMinSize(username, "login.username", 5);
            validateFieldMinSize(password, "login.password", 8);
            validateFieldMaxSize(username, "login.username", 40);
            validateFieldMaxSize(password, "login.password", 29);

            //do the real save action
            // if login != ok thorw new ActionEXception(....

            insertMessage("user.loggedin");

            return ACTION_OK;
        }
    }

    /**
     * Executes the login action in the lotin bean.
     */
    public String loginAction() {
        return new SampleLoginBean().execute();
    }
```

Examinemos el código:

- Tenemos una clase interna y privada LoginAction que implementa la lógica de login para este bean.

- Tenemos un método de acción público loginAction() que ejecuta la acción.

Las ventajas de esta forma de reescribir la lógica de interfaz y de negocio son evidentes:

- Nuestra clase LoginAction está libre de código accesorio. Su escritura es simple, muy fácil de leer y de mantener.
- Hemos usado un patrón de tipo "Salto de vallas" para nuestra acción, es decir, la acción se va ejecutando linealmente, evitando bifurcaciones, y si todas las etapas (obstáculos) de la acción se cumplen se retorna el "OK". Por cierto, aquel que utilice pruebas unitarias para probar sus clases entenderá las ventajas de usar este patrón, ya que la cobertura completa de los test es más fácil de lograr, debido a la ausencia de bifurcaciones.
- Cualquier condición no cumplida o problema que aparezca durante la ejecución se resolverá lanzando una excepción de tipo ActionException, que será capturada por la clase SimpleActionMethod, controlando la correcta finalización de la acción.

6. Añadiendo flexibilidad y acceso a datos al ActionMethod

A continuación voy a poner como ejemplo una clase de ejemplo un poco más compleja, que añade nuevas características para hacerla extensible. Dejo al lector el estudio de la misma y el pensar cómo podría reescribir nuestro problema inicial (el aprovisionamiento de un serviodr) utilizando estas clases como punto de partida. Introducimos el siguiente código dentro de nuestra clase ActionBase:

```
...

    /**
     * Base internal class for Action methods
     */
    public class ActionMethod {

        public String execute() {
            String result = ACTION_OK;
            log(">> Action method begin.");

            initExecute();


            try {

                beforeExecute();

                result = executeAction();

                // TODO: different Exception catchs for different kind of
                // exceptions
                // catch (DBException e) {
                // insertMessage("application.defaultMessageDatabaseExceition
                // result = ERROR;
                //

                afterExecute();

            } catch (ActionException e) {
                log("Error ActionMethod.execute() action exception: ", e);
                result = ACTION_ERROR;
            } finally {
                finallyExecute();
            }

            finishExecute();

            log(">> Action method end.");
            return result;
        }

        /**
         * Mothod to be overwriten in derived classes. This methods
         * must implements the real execute action.
         */
        protected String executeAction() throws ActionException {
            insertMessage("application.defaultMessageExceition");
            throw new ActionException("Error: new action must override "
                    + "ActionMethod.executeAction method for class "
                    + getClassName());
        }

        // Ectenxion methods that can be overridden

        protected void initExecute()  {
        }

        protected void beforeExecute() {
        }

        protected void afterExecute() {
        }

        protected void finishExecute() {
        }

        protected void finallyExecute() {
        }

    }
```

Para rizar el rizo, vamos a crear un ActionMethod interno que se encargue de la conexión a base de datos, para crear acciones que realizan consultas a base de datos de forma sencilla: más o menos podría quedar así:

```
// ---------------- ActionDatabaseMethod -------------------------
public class ActionDatabaseMethod extends ActionMethod {

    protected Session session;
    protected Transaction transaction;

    public ActionDatabaseMethod() {
        super();

        /** Begins the Database session and transaction */
    }
```

```
12
13      protected void beforeAction() {
14          super.beforeExecute();
15          session = SessionFactory.getSession();
16          transaction = session.getTransaction();
17      }
18
19      /** Commit transaction */
20      protected void afterExecute() {
21          transaction.commit();
22          super.afterExecute();
23      }
24
25      // Do databaase rollback if neccesary */
26      protected void finallyExecute() {
27          if (transaction.isActive()) {
28              transaction.rollback();
29          }
30          super.finallyExecute();
31      }
32
33      /**
34       * Execute a database action inside a specific try/catch. This method
35       * wrap the action execution to isolate specific exception handling
36       *
37       * @throws ActionException
38       */
39      final public String executeAction() throws ActionException {
40
41          String result;
42          try {
43              result = executeDatabaseeAction();
44          } catch (DatabaseException e) {
45              throw new ActionException("Database Exception "
46                      + e.getMessage(), e);
47
48          }
49          return result;
50      }
51
52      /**
53       * Mothod to be overwriten in derived classes. This methods
54       * mustimplements the real execute action.
55       *
56       * @throws DatabaseException
57       * @throws ActionException
58       */
59      protected String executeDatabaseeAction() throws DatabaseException,
60              ActionException {
61          insertMessage("application.defaultMessageExceition");
62          throw new ActionException("Error: new action must override "
63                  + "ActionDatabaseMethod.executeDatabaseAction method for class "
64                  + getClassName());
65      }
66
67  }
```

Esto es una aproximación a un código real. Seguro que a todos se nos ocurre cómo mejorar este código o adaptarlo a nuestros proyectos actuales.

Dejo para el lector la tarea de adaptar los beans de ejemplo utilzando nuestro ActionBase mejorado.

7. Conclusión

En este artículo hemos aprendido una forma sencilla de escribir nuestras acciones JSF de manera que toda la funcionalidad accesoria de la acción quede encapsulada en una clase interna Java. Con ello conseguimos:

- Una gestión uniforme de la ejecución de las acciones JSF.
- Mayor claridad en el código de las acciones JSF, que simplifica su creación y mantenimiento.

Nos veremos pronto en esta página con otros tutoriales.

A continuación puedes evaluarlo:

Regístrate para evaluarlo

Por favor, vota +1 o compártelo si te pareció interesante

Share | 0

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

» Registrate y accede a esta y otras ventajas **«**

SOME RIGHTS RESERVED Esta obra está licenciada bajo licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5

IMPULSA Impulsores Comunidad ¿Ayuda?

---
sin clicks

0 personas han traído clicks a esta página

+ + + + + + + +

powered by karmacracy

Copyright 2003-2013 © All Rights Reserved | Texto legal y condiciones de uso | Banners | Powered by Autentia | Contacto

XHTML 1.0 CSS XML RSS XML ATOM