Controladores

Índice

En la sesión anterior, vimos como crear y modificar plantillas en nuestra aplicación, así como algunas etiquetas útiles que nos servirán a lo largo de todo el proceso de desarrollo de una aplicación con Grails. También vimos como implementar algunas acciones en los controladores.

En esta sesión empezaremos viendo una introducción a los controladores y posteriormente, veremos como mejorar su funcionalidad y mantenimiento. Posteriormente pasaremos a ver también un concepto conocido como *interceptadores de acciones*, que permite ejecutar una serie de métodos antes de que una determinada acción sea invocada por un controlador. Por último, echaremos un vistazo a los filtros, que son bastantes similares a los interceptadores de acciones con algunas pequeñas diferencias como la posibilidad de definir estos filtros para que puedan ser utilizados en diversos controladores.

1. Introducción

En una aplicación que siga el patrón Modelo Vista Controlador, los controladores son los componentes que se encargan de recibir las órdenes por parte del usuario, gestionar la ejecución de la lógica de negocio y posteriormente actualizar la vista para que el usuario pueda ver como ha quedado el modelo de datos tras las actualizaciones pertinentes.

Hablando en términos de aplicaciones web, los controladores se encargan de interceptar las peticiones HTTP del navegador y generar la respuesta correspondiente, bien sea html, xml, json o cualquier otro formato, bien desde el propio controlador o delegando el trabajo en una vista GSP.

La convención que utiliza Grails en el caso de los controladores es que un controlador es cualquier clase que se encuentre en el directorio *grails-app/controllers* de nuestro proyecto y su nombre termine por la palabra *Controller*.

En cada petición HTTP, Grails debe determinar que controlador debe ser invocado en función de las reglas establecidas en el archivo *grails-app/conf/UrlMappings.groovy*. La configuración por defecto que utiliza Grails es `/[controlador]/[accion]/[id]`. En la sesión 12 veremos como modificar este comportamiento por defecto.

1.1. Ámbitos

Para facilitar el almacenamiento de la información generada por la aplicación durante su utilización, los controladores disponen de una serie de objetos implícitos. Estos objetos son *mapas* y coinciden con los ámbitos que se pueden encontrar habitualmente en cualquier aplicación web.

- `servletContext` - contiene los datos del ámbito de la aplicación. Cualquier valor que almacenemos en este objeto estará disponible globalmente, desde cualquier controlador o acción

- `session` - permite asociar un estado a cada usuario, habitualmente mediante el envío de cookies. Los datos que guardaremos en este ámbito serán visibles únicamente para el usuario, mientras su sesión esté activa
- `request` - estos valores sólo estarán disponibles durante la ejecución de la solicitud actual
- `params` - contiene todos los parámetros de la petición actual, tanto los de la url como los del formulario. El mapa *params* puede ser modificado en cualquier momento para añadir valores o modificar los existentes
- `flash` - este ámbito es un almacén temporal para atributos que necesitaremos durante la petición actual y la siguiente, y que serán eliminados cuando ambas se hayan procesado. Este ámbito se suele utilizar para almacenar el código de error en caso de redirecciones, tal y como veremos en la sesión 8.

1.2. El método render()

Cuando una acción de un controlador que se está ejecutando no incluye ninguna llamada al método *render*, Grails buscará la vista predeterminada para dicha acción en el archivo *grails-app/views/[controlador]/[acción]*. Esto sucede en el método `login` de la clase *Usuario* donde no especificábamos ninguna llamada al método `render`.

En caso de que la acción devuelva un mapa con el modelo a mostrar, los campos definidos en el mapa estarán disponibles en la vista GSP como variables locales. Mientras que si la acción no devuelve nada, la página GSP tendrá acceso sólo a las variables locales definidas en el propio controlador.

En muchos casos esto será suficiente para producir la salida de nuestras páginas GSP, pero habrá otros casos en los que necesitaremos mayor control sobre esta salida producida. Para ello necesitamos el método `render()`, que es un método que se utiliza para enviar respuestas al cliente de varias formas. El método `render()` acepta varios parámetros en función de la salida que necesitemos.

- `text` - la respuesta enviada será en texto plano
- `builder` - se puede enviar un builder para generar la respuesta
- `view` - se indica la vista que queremos procesar para generar la respuesta
- `template` - se indica la plantilla que queremos procesar para generar la respuesta
- `plugin` - se indica el plugin donde buscar la plantilla, si ésta no pertenece a nuestra aplicación
- `bean` - un objeto con los datos para generar la respuesta
- `var` - el nombre de la variable con la que accederemos al *bean*. Si no se indica este parámetro, se utilizará la variable *it*
- `model` - un mapa con el modelo para usar en la vista
- `collection` - Una colección para procesar una plantilla con cada elemento
- `contentType` - el tipo mime de la respuesta
- `encoding` - el juego de caracteres de la respuesta
- `converter` - un objeto del tipo *grails.converters.** para generar la respuesta

Todos estos parámetros son opcionales y en función de los que proporcionemos, la salida será diferente. Veamos algunos ejemplos:

```
//Enviar texto
render "un texto devuelto"

//especificar el tipo mime y codificación
render (
        text:"<error>Ha habido un error</error>",
        contentType:"text/xml",
        encoding:"UTF-8"
)

//procesar una plantilla con un modelo
render (
        template:'listado',
        model:[lista:Usuario.list()]
)

//o una colección
render (
        template:'listado',
        collection:[u1,u2,u3]
)

//o con un objeto
render (
        template:'listado',
        bean:Usuario.get(2),
        var:'u'
)

//procesar una vista con un modelo
render (
        view:'listado',
        model:[lista:Usuario.list()]
)

//o con el propio controlador como modelo
render (view:'usuario')

//con un builder
render {
        div(id:'miDiv','Contenido del div')
}
render (contentType:'text/xml'){
        listado {
                Usuario.list().each {
                        usuario(
                                nombre:it.nombre,
                                apellidos:it.apellidos
                        )
                }
        }
}

//generando JSON
render (contentType:'text/json') {
```

```
        usuario(nombre:u.nombre, apellidos:u.apellidos)
}

//generar XML y JSON automáticamente
import grails.converters.*

render Usuario.list(params) as XML
render Usuario.get(params.id) as JSON
```

2. Mejorando los controladores

Hasta ahora, hemos utilizado el scaffolding dinámico para gestionar la información de las diferentes clases de nuestra aplicación y lo que es evidente es que si Grails permite este tipo de scaffolding al vuelo, también permitirá un scaffolding estático en el cual el código se genere de forma offline para que pueda ser modificado a nuestro antojo.

La primera clase que vamos a modificar va a ser la clase *Usuario* y en primer lugar, necesitamos generar sus vistas, que hasta ahora se estaban generando dinámicamente. Para ello, debemos ejecutar el comando `grails generate-views Usuario`. Este comando utiliza unas plantillas para crear las vistas de las cuatro operaciones básicas, lo que se resume en la creación en el directorio *grails-app/views/usuario* de cuatro archivos, uno para cada una de las vistas. Estos archivos son: *create.gsp*, *edit.gsp*, *list.gsp* y *show.gsp*. Si nos fijamos en nuestro proyecto en NetBeans, veremos que se han creado cuatro nuevos archivos en la categoría de *Views and Layouts*.

Ahora que ya tenemos las vistas generadas, necesitamos volver a generar el controlador de la clase *Usuario* para que en lugar de utilizar scaffolding dinámico a partir de ahora sea estático. Antes de ejecutar el comando `grails generate-controller Usuario`, hacemos una copia de seguridad del controlador de la clase *Usuario* para conservar los métodos `login()`, `handleLogin()` y `logout()`.

Una vez realizada dicha copia de seguridad y ejecutado el comando `grails generate-controller Usuario`, podemos volver a abrir el archivo *UsuarioController.groovy* y comprobaremos que su contenido se ha modificado bastante del que teníamos al principio y se han generado una serie de métodos. Si echamos un vistazo al nombre de estos métodos (`index(), list(), show(), delete(), edit(), update(), create() y save()`), nos daremos cuenta de que se corresponden con los métodos necesarios para realizar el scaffolding de forma estática.

El siguiente paso será copiar los métodos `login(), handleLogin() y logout()` de la copia de seguridad que hemos hecho del anterior controlador de la clase *Usuario*. Cuando ya hayamos copiado estos métodos, la copia de seguridad ya no nos sirve y la podremos eliminar. Ahora ya podemos empezar a mejorar el controlador de la clase *Usuario* y la primera mejora consistirá en conseguir que un usuario sólo pueda modificar sus datos.

Para solucionar este problema de autenticación, cada vez que alguien intente editar o eliminar un usuario, se comprobará que ese *alguien* coincide con el usuario en cuestión.

En caso contrario, se mostrará un mensaje de error indicándole que sólo puede editar su propia información. En primer lugar nos centraremos en la posibilidad de editar los datos de un usuario y para ello, modificaremos el método `edit()`. El contenido actual de este método es el siguiente:

```
def edit = {
    def usuarioInstance = Usuario.get( params.id )

    if(!usuarioInstance) {
        flash.message = "Usuario not found with id ${params.id}"
        redirect(action:list)
    }
    else {
        return [ usuarioInstance : usuarioInstance ]
    }
}
```

Como puedes comprobar, en primer lugar se identifica que usuario se desea modificar por medio de `Usuario.get(params.id)`. Si el usuario no existe, se mostrará un mensaje de error y en caso contrario se devuelve la instancia generada. Pues bien, antes de todo este código vamos a comprobar que el *ID* pasado por parámetro coincide con el que tenemos almacenado en la variable *session* (está ahí desde que el usuario se identificó en el sistema). Si no coincide, mostraremos un mensaje de error, invocaremos el método `list()` de la clase Usuario y haremos un `return` para abandonar el método actual. En caso contrario, se continuará con la ejecución normal del método. El nuevo método `edit()` quedaría así:

```
def edit = {
        if (session?.usuario?.id as String != params.id){
                flash.message = "Sólo puedes editar tu información"
                redirect(action:list)
                return
        }

        def usuarioInstance = Usuario.get( params.id )

        if(!usuarioInstance) {
            flash.message = "Usuario not found with id
${params.id}"
            redirect(action:list)
        }
        else {
            return [ usuarioInstance : usuarioInstance ]
        }
}
```

Si intentamos ahora modificar los datos de un usuario diferente al cual nos hemos identificado, el sistema nos mostrará el mensaje de error que hemos especificado. Esta misma comprobación la debemos implementar en las funciones `update()` y `delete()`.

Nuestra aplicación acaba de aumentar su seguridad, sin embargo, todavía queda mucho por hacer. Si recordamos la especificación de la aplicación, decíamos que los *administradores* eran los únicos encargados de la creación de usuarios, con lo que

debemos hacer algo para comprobar que sólo los administradores tengan la posibilidad de crear nuevos usuarios. Para conseguir esto, en el método `create()` debemos comprobar que el usuario que intenta crear un nuevo usuario es del tipo *administrador*. Siguiendo la misma metodología que en el método `edit()`, el método `create()` quedaría así:

```
def create = {
    if (session?.usuario?.tipo != "administrador"){
        flash.message = "Sólo los administradores pueden crear
usuarios"
        redirect(action:list)
        return
    }

    def usuarioInstance = new Usuario()
    usuarioInstance.properties = params
    return ['usuarioInstance':usuarioInstance]
}
```

Si ahora intentamos crear un usuario sin ser *administradores*, el sistema nos indicará que no podemos, puesto que no tenemos los privilegios suficientes.

De igual forma que sólo los administradores pueden crear nuevos usuarios, los datos de los usuarios sólo podrán ser editados por los propios usuarios y por los administradores, con lo que los métodos `edit()`, `update()` y `delete()` anteriores, deben ser modificados para controlar esta restricción.

```
def edit = {
        if ((session?.usuario?.id as String != params.id) &&
(session?.usuario?.tipo != "administrador")){
                flash.message = "Sólo puedes editar tu información"
                redirect(action:list)
                return
        }

        def usuarioInstance = Usuario.get( params.id )

        if(!usuarioInstance) {
            flash.message = "Usuario not found with id
${params.id}"
            redirect(action:list)
        }
        else {
            return [ usuarioInstance : usuarioInstance ]
        }
}
```

Veamos algo sobre la variable *param*. Esta variable es de tipo *mapa* y recoge los valores pasados en la petición. Para acceder a las propiedades de este mapa podemos utilizar el operador punto (.id) o bien el operador corchete ['id']. El operador corchete nos servirá para aquellos casos en los que la propiedad a analizar tenga caracteres extraños como un punto (.) o una barra (/) o en aquellas situaciones en que se deba componer el nombre de la propiedad.

Pero, ¿cómo se pasan los parámetros al controlador? Si nos vamos al listado de usuarios y

hacemos clic en cualquiera de los usuarios para ver su información, podemos ver la siguiente dirección web: *http://localhost:8080/biblioteca/usuario/show/1*. Esta dirección url podemos dividirla en seis partes:

- *http://localhost*: dominio
- *8080*: puerto
- *biblioteca*, nombre de la aplicación
- *usuario*, nombre del controlador
- *show*, nombre de la acción ejecutada por el controlador
- *1*, el parámetro ID

Si nos fijamos a partir de ahora en las direcciones url utilizadas por nuestra aplicación, veremos como todas siguen el mismo convenio. Fijémonos ahora en la sentencia `return [usuarioInstance: usuarioInstance]`. Esta sentencia devuelve un mapa de valores, en el que la entrada anterior al símbolo : se refiere a la clave y la segunda se refiere al valor. Desde los archivos correspondientes a la vista, como por ejemplo en el archivo *show.gsp*, podemos ver como se accede a esta información de la siguiente forma:

```
<tr class="prop">
    <td valign="top" class="name">Login:</td>

    <td valign="top"
class="value">${fieldValue(bean:usuarioInstance,
field:'login')}</td>

</tr>
```

Como podemos ver, se hace referencia a la variable *usuarioInstance*, que es la misma que devuelve el método `show()` del controlador y en este caso a la propiedad *login*. El método `fieldValue()` no sólo se puede utilizar como tal pasándole el *bean* utilizado y el campo a devolver, sino también como una etiqueta `<g:fieldValue bean="${usuarioInstance}" field="login" />` que puede ser incluida en cualquier página GSP.

Sigamos analizando los controladores generados automáticamente por el scaffolding estático de Grails. Si ahora prestamos atención al método `save()` del controlador de la clase *Usuario* (utilizado para persistir la información en la base de datos de los nuevos usuarios creados), nos daremos cuenta de lo potente que es Grails a la hora de pasar información entre la vista y el controlador.

```
def save = {
    def usuarioInstance = new Usuario(params)
    if(!usuarioInstance.hasErrors() && usuarioInstance.save()) {
        flash.message = "Usuario ${usuarioInstance.id} created"
        redirect(action:show,id:usuarioInstance.id)
    }
    else {
        render(view:'create',model:[usuarioInstance:usuarioInstance])
    }
}
```

Simplemente con una sola línea de código, `def usuarioInstance = new Usuario(params)` hemos creado el nuevo usuario, aunque éste todavía no ha sido almacenado en la base de datos. La vista *create.gsp* es la encargada de generar el formulario de entrada de los datos del nuevo usuario (*login, password, nombre, apellidos y tipo*). Cuando el usuario envía este formulario, todos los datos del nuevo usuario se pasan al controlador a través de la variable *params*, que posteriormente es utilizado para crear el nuevo usuario a través del constructor de la clase *Usuario*. Esta forma de trabajar nos ahorra mucho tiempo y esfuerzos y si alguna vez has trabajado con otros frameworks o lenguajes de programación, podrás constatarlo.

Si todo ha ido bien, lo cual se comprueba chequeando la presencia de errores mediante `hasErrors()` e intentando salvar el nuevo usuario en la base de datos mediante el método de GORM `save()`, se mostrará un mensaje indicando que el nuevo usuario se ha generado correctamente. En caso de que se produzca algún error, se renderizará la vista *create.gsp* a la cual se le pasará como modelo los datos del nuevo usuario para que se corrijan los datos erróneos.

Pero sigamos mejorando nuestra aplicación. En cuanto a los usuarios, quizás lo único que nos resta por controlar será evitar que los usuarios que no sean administradores no puedan modificar la propiedad *tipo* para que un usuario malintencionado no consiga elevar sus privilegios y tirarnos abajo el sistema. Para ello, vamos a añadir una comprobación a la vista *edit.gsp* para que sólo se muestre la posibilidad de editar la propiedad *tipo* de un usuario en caso de que el usuario que intente modificarlo sea un administrador.

```
<g:if test="${session?.usuario?.tipo == 'administrador' }">
        <tr class="prop">
            <td valign="top" class="name">
                <label for="tipo">Tipo:</label>
            </td>
            <td valign="top" class="value
${hasErrors(bean:usuarioInstance,field:'tipo','errors')}">
                <g:select id="tipo" name="tipo"
from="${usuarioInstance.constraints.tipo.inList}"
value="${usuarioInstance.tipo}" ></g:select>
            </td>
        </tr>
</g:if>
```

3. Interceptadores de acciones

Como cualquier aplicación en fase de desarrollo, los fallos y problemas son comunes y necesitamos establecer sistemas para detectar y solucionarlos en el menor tiempo posible. Cuando se produce uno de estos fallos, es necesario conocer el controlador que causó el fallo así como los datos de entrada y la acción que se ejecutó. Por este motivo, Grails añade lo que se conoce como *Interceptadores de acciones* que no son más que métodos que se ejecutan antes y después de cualquier acción de un controlador. Una de las utilidades de los interceptadores de acciones se refiere a la generación de archivos de *log*

con todas las acciones ejecutadas en nuestra aplicación, así como los datos de entrada.

Grails nos ofrece la posibilidad de utilizar los interceptadores antes y después de las acciones. Los métodos encargados son `beforeInterceptor()` y `afterInterceptor()`. En nuestra aplicación, vamos a utilizar estos métodos para la generación de un archivo de *log* con todas las acciones ejecutadas en ella. En concreto, escribiremos en nuestro *log* el nombre del usuario que realiza la acción, el nombre del controlador y la acción invocada, así como los parámetros.

En primer lugar, vamos a añadir los métodos `beforeInterceptor()` y `afterInterceptor()` al controlador de la clase *Usuario*. De esta forma, podremos conocer todo lo que se ha realizado en cuanto a la gestión de los usuarios.

```
def beforeInterceptor = {
    log.trace("${session?.usuario?.login} Empieza la acción
${controllerName} Controlador.${actionName}() : parámetros
$params")
}
```

De igual forma, el método `afterInterceptor()` será muy similar al método `beforeInterceptor()` con la salvedad de que ahora se le pasará el *modelo* para que se pueda imprimir en el *log*.

```
def afterInterceptor = { model ->
    log.trace("${session?.usuario?.login} Termina la acción
${controllerName} Controlador.${actionName}() : devuelve $model")
}
```

Los métodos `beforeInterceptor()` y `afterInterceptor()` son invocados antes y después de acción en un controlador. Sin embargo, es posible que no queramos que estos métodos se ejecutan antes y después de todas las acciones que consideremos intrascendentes. Si en nuestro ejemplo, consideramos innecesario realizar un log sobre la acción `list()`, podríamos utilizar el siguiente código.

```
def beforeInterceptor = [action:this.&beforeAudit,except:['list']]

def afterInterceptor = [action:{model ->this.&afterAudit(model)},
except:['list']]

def beforeAudit = {
    log.trace("${session?.usuario?.login} Empieza la acción
${controllerName} Controlador.${actionName}() : parámetros
$params")
}

def afterAudit = { model ->
    log.trace("${session?.usuario?.login} Termina la acción
${controllerName} Controlador.${actionName}() : devuelve $model")
}
```

Para poder generar estos archivos de log necesarios para realizar un completo análisis de nuestra aplicación, necesitamos configurar *log4j*, una librería de Apache ampliamente utilizada para este tipo de tareas. Esta librería permite establecer una serie de niveles de

granularidad de los mensajes emitidos por el sistema. Con log4j se acabaron los típicos `println()` que tantos problemas pueden provocar.

Log4j presenta seis niveles de detalle:

- *Fatal:* se utiliza para mensajes críticos del sistema, generalmente después de guardar el mensaje el programa abortará.
- *Error:* se utiliza en mensajes de error de la aplicación que se desea guardar. Estos eventos afectan al programa pero lo dejan seguir funcionando, como por ejemplo que algún parámetro de configuración no es correcto y se carga el parámetro por defecto.
- *Warn:* se utiliza para mensajes de alerta sobre eventos que se desea mantener constancia, pero que no afectan al correcto funcionamiento del programa.
- *Info:* se utiliza para mensajes similares al modo *verbose* en otras aplicaciones.
- *Debug:* se utiliza para escribir mensajes de depuración. Este nivel no debe estar activado cuando la aplicación se encuentre en producción.
- *Trace:* se utiliza para mostrar mensajes con un mayor nivel de detalle que debug.

En log4j los mensajes son enviados a una o varias salidas de destino, lo que se denomina un *appender*. Por defecto existen varios *appender* disponibles y configurados, pero también podemos crear nuestros propios appenders. Los típicos appender son aquellos en los que la salida es redirigida a un fichero de texto con extensión .log (`FileAppender`, `RollingFileAppender`), aunque también es posible almacenar estos registros en un servidor remoto gracias a `SocketAppender`, a una dirección de correo electrónico con `SMTPAppender` e incluso a una base de datos con `JDBCAppender`.

En nuestro ejemplo vamos a volcar la información generada en un fichero llamada *miLog.log* que se ubicará en la raíz de la aplicación. Para eso debemos configurar log4j para que funcione correctamente en nuestra aplicación y eso es algo que debemos hacer en el archivo *grails-app/conf/Config.groovy*, donde si echamos un vistazo rápido podremos comprobar como el sistema ya ha generado un pequeño ejemplo para nosotros, que ahora debemos modificar.

Como puedes ver, la apariencia de la configuración de log4j en Grails es el típico closure similar a los que vimos cuando hablamos sobre Groovy. En el ejemplo que viene por defecto en el archivo de configuración *Config.groovy*, debemos añadir un appender de tipo *file* para que vuelque la salida a un fichero pasado también por parámetro. Con esto, la parte de configuración de log4j quedaría así:

```
// log4j configuration
log4j = {
    appenders {
        file name:'file', file:'mylog.log'
    }
    root {
        info 'file'
        additivity = true
    }

    trace "grails.app.controller.UsuarioController"
```

```
    error  'org.codehaus.groovy.grails.web.servlet',  //
controllers
               'org.codehaus.groovy.grails.web.pages', //  GSP
               'org.codehaus.groovy.grails.web.sitemesh', //
layouts
               'org.codehaus.groovy.grails."web.mapping.filter', //
URL mapping
               'org.codehaus.groovy.grails."web.mapping', // URL
mapping
               'org.codehaus.groovy.grails.commons', // core /
classloading
               'org.codehaus.groovy.grails.plugins', // plugins
               'org.codehaus.groovy.grails.orm.hibernate', //
hibernate integration
               'org.springframework',
               'org.hibernate'

    warn   'org.mortbay.log'
}
```

Además del appender que le indica al sistema que la información será volcada a un fichero, le hemos indicado también que el controlador *UsuarioController* va a tener un nivel de granularidad de nivel *trace*, puesto que en los métodos `beforeAudit()` y `afterAudit()` ya le indicamos este nivel.

Si ejecutamos nuestra aplicación y realizamos algunas operaciones sobre los usuarios, podemos comprobar como el fichero *miLog.log* va añadiendo todas las operaciones relacionadas con el controlador *Usuario*.

4. Filtros

Los filtros son similares a los *interceptadores de acciones* en el sentido de que se ejecutan antes y después de una acción. Sin embargo, la diferencia entre ellos es que los filtros son más flexibles ya que, a diferencia de los interceptadores, éstos pueden ser utilizados en múltiples controladores.

Vamos a ver algún ejemplo donde podemos utilizar los filtros. En la primera parte de esta sesión veíamos como añadir algo de seguridad a nuestra aplicación comprobando que determinadas acciones (crear, editar, y eliminar usuarios), sólo podrían ser realizadas bien por los administradores o por el propio usuario. Gracias a los filtros vamos a centralizar estas comprobaciones para que sólo se realicen desde una determinada función. Por ejemplo, en las acciones `edit()`, `update()` y `delete()` comprobábamos que sólo el propio usuario podría realizar estas acciones y lo hacíamos de la siguiente forma:

```
if (session?.usuario?.id as String != params.id){
    flash.message = "Sólo puedes editar tu información"
    redirect(action:list)
    return
}
```

Esta lógica se repite en los tres métodos comentados, así que vamos a utilizar un filtro

para centralizar su tratamiento. En Grails, los filtros se generan creando una nueva clase que termine con la palabra *Filters*. Estas nuevas clases se deben crear en el directorio *grails-app/conf*. Cada filtro implementa la lógica a ejecutar antes y después de las acciones así como sobre que acciones se debe hacer. En nuestro caso, debemos crear un archivo nuevo en el directorio *grails-app/conf* llamado *UsuarioFilters.groovy* y un ejemplo de código podría ser el siguiente:

```
class UsuarioFilters {
    def filters = {
        chequeoModificacionUsuario(controller: 'usuario', action:
'*') {
            ...
        }
        otroFiltro(uri: '/usuario/*') {
        }
    }
}
```

Tal y como vemos en el código de ejemplo, se han definido dos *filtros* para la clase *Usuario*. De momento sólo se ha especificado el ámbito de aplicación de estos filtros. Para el primer filtro definido el ámbito de aplicación es el controlador *Usuario* y las acciones donde se ejecutará este filtro serán todas, gracias al símbolo *****. En nuestro caso, las acciones donde necesitamos ejecutar este filtro son 3, `edit()`, `update()` y `delete()`, pero esto lo vamos a controlar dentro de la propia lógica del filtro, tal y como veremos a continuación.

Para el segundo filtro se ha escogido otra forma de declarar el ámbito de aplicación de los filtros, que es especificando la dirección *uri* empleada, en el ejemplo, se referiría a cualquier acción ejecutada dentro del controlador de la clase *Usuario*, con lo que ambos filtros se ejecutarían en todas las acciones de la clase *Usuario*.

Lo único que nos restaría por indicarle al primer filtro es cuando queremos que se ejecuten estos métodos. Los filtros en Grails pueden ser ejecutados antes (`before()`) o después de la acción (`after()`) o incluso después de que la vista se haya renderizado (`afterView()`). En nuestro caso necesitamos que el filtro se ejecute antes de que comience la acción y podríamos tener algo así:

```
class UsuarioFilters {
    def filters = {
        chequeoModificacionUsuario(controller: 'usuario', action:
'*') {
            before = {
                if (actionName == 'edit' || actionName == 'update'
|| actionName == 'delete') {
                    if (session?.usuario?.id as String !=
params?.id) {
                        flash.message = "Sólo puedes editar tu
información"
                        redirect(controller:'usuario', action:
'list')
                        return false
                    }
```

```
                }
            }
        }
    }
}
```

Si ahora eliminamos las comprobaciones introducidas anteriormente en el controlador de la clase *Usuario*, veremos como un usuario sólo podrá editar y eliminarse a si mismo.

Ahora bien, también debemos añadir la posibilidad a los administradores de realizar estas acciones, con lo que debemos modificar este filtro para añadir esa característica a la aplicación. El filtro final quedaría así:

```
class UsuarioFilters {
    def filters = {
        chequeoModificacionUsuario(controller: 'usuario', action:
'*') {
            before = {
                if (actionName == 'edit' || actionName == 'update'
|| actionName == 'delete') {
                    if ((session?.usuario?.id as String !=
params?.id) && (session?.usuario?.tipo != "administrador")) {
                        flash.message = "Sólo puedes editar tu
información"
                        redirect(controller:'usuario', action:
'list')
                        return false
                    }
                }
            }
        }
    }
}
```

Con estos filtros, el código de nuestros controladores queda más limpio y hemos conseguido centralizar la lógica que gestiona las comprobaciones de usuario en un sólo fichero y una sola función.

Otra característica de nuestra aplicación era que sólo los administradores de la aplicación pueden crear usuarios, cosa que ya habíamos añadido a nuestro controlador, pero que ahora vamos a cambiar para hacerlo utilizando los filtros, que para algo están. Para esto, vamos a crear un nuevo filtro en la clase *UsuarioFilters* que se encargue de esta comprobación.

```
chequeoCreacionUsuario(controller: 'usuario', action: '*') {
    before = {
        if (actionName == 'create' || actionName == 'save'){
            if (session?.usuario?.tipo != "administrador") {
                flash.message = "Sólo los administradores pueden
crear usuarios"
                redirect(controller:'usuario', action: 'list')
                return false
            }
        }
    }
```

```
}
```

Ahora podemos eliminar esta comprobación que antes se realizaba directamente en los métodos `create()` y `save()` del controlador de la clase *Usuario* y veremos como sólo los administradores son capaces de crear usuarios.