



UNIVERSIDAD AUTÓNOMA DEL ESTADO DE HIDALGO

INSTITUTO DE CIENCIAS BÁSICAS E INGENIERÍA

Centro de Investigación en Tecnologías de la Información y Sistemas

Desarrollo Ágil de Aplicaciones Web con Grails Framework.

Caso de estudio: PROMEP-UAEH

MONOGRAFÍA

Para obtener el título de

LICENCIADO EN SISTEMAS COMPUTACIONALES

Presenta:

P.L.S.C. ALEJANDRO GARCÍA GRANADOS

Asesor:

M. C. C. EDUARDO CORNEJO VELÁZQUEZ

Mineral de la Reforma, Hidalgo, enero 2012



UAEH

Programa de Mejoramiento del Profesorado

Este trabajo se desarrolló en el área de Sistemas de Información de la Dirección de apoyo PROMEP dependiente de la Coordinación de Investigación y Posgrado de la Universidad Autónoma del Estado de Hidalgo; bajo la dirección del M.C.C. Eduardo Cornejo Velázquez.

Las versiones preliminares de este documento y el trabajo práctico desarrollado han sido utilizados en el taller “Desarrollo ágil de aplicaciones con Groovy & Grails”, así como en la ponencia “Desarrollo Web con Grails Framework” durante la celebración del Congreso Universitario en Tecnologías de Información y Comunicaciones 2011. Por otro lado, alumnos de octavo semestre de la Licenciatura en Sistemas Computacionales han sido introducidos en el uso de Grails en la materia de Bases de Datos II usando este documento como referencia, teniendo gran aceptación entre los mismos. Y finalmente, alumnos que cursan las materias de Proyecto de Fin de Carrera I y II han comenzado a trabajar en su proyectos terminales empleando este documento como material de referencia al encontrarse desarrollando aplicaciones con Grails.

Agradecimientos

Antes que nadie, quiero agradecer infinitamente a mi familia por el apoyo y amor incondicional que siempre me han brindado. Oscar, Guillermina, Oscar y Claudia, este trabajo es para ustedes.

Asimismo, agradezco al M. C. C. Eduardo Cornejo Velázquez por sus enseñanzas, consejos y asesorías en el desarrollo de éste documento; pero sobre todo, su amistad.

Finalmente, quiero agradecer a Alejandra Gómez Varela, mi compañera de viaje, por la comprensión y sobre todo la inspiración para la realización de este proyecto.

Alejandro García Granados.

Contenido

Agradecimientos.....	III
Parte 1. Introducción.....	1
Capítulo 1. Introducción al documento.....	2
1.1 Los inicios en el desarrollo de aplicaciones web.....	2
1.2 Definición del Problema.....	4
1.3 Justificación.....	4
1.4 Objetivo General.....	5
1.5 Objetivos Específicos.....	5
1.6 Metodología.....	6
1.7 Herramientas tecnológicas.....	7
1.8 Descripción del trabajo.....	7
Capítulo 2. Requerimientos, análisis y diseño del Sistema de Control y Seguimiento de Indicadores PROMEP-UAEH (CSI).....	10
2.1 PROMEP.....	10
2.2 Requerimientos.....	11
2.2.1 Administración de correspondencia.....	12
2.2.2 Seguimiento de indicadores.....	15
2.2.3 Tablero de control.....	17
2.2.4 Seguimiento de evaluaciones de cuerpos académicos.....	19
2.2.5 Seguimiento de participación en convocatorias.....	20
2.3 Análisis.....	20
2.3.1 Modelado de entidades.....	21
2.3.2 Modelado de procesos.....	21
2.4 Diseño.....	22
2.4.1 Capa de lógica de negocios.....	22
2.4.2 Capa de control.....	23
2.4.3 Capa de presentación.....	23
2.4.4 Diagrama de la arquitectura de 3 capas.....	24
2.5 Resumen.....	25
Parte 2. Introducción a Grails.....	26
Capítulo 3. ¿Por qué Grails?.....	27
3.1 El inicio de todo: El lenguaje de programación Java.....	27
3.2 Java Enterprise Edition.....	29
3.3 La tecnología base: Servlets.....	30
3.4 Frameworks: encapsulado y abstracción de funcionalidad.....	31
3.5 Facilitando JEE: Spring Framework.....	32
3.6 Persistencia Orientada a Objetos: Hibernate Framework.....	33
3.7 La evolución natural: Lenguaje de programación Groovy.....	35

3.8 Descubriendo el grial: Grails.....	36
3.8.1 Convención sobre Configuración.....	37
3.8.2 Filosofía Ágil.....	38
3.8.3 Fundamentos sólidos.....	38
3.8.4 Plantillas y “Scaffolding”.....	38
3.8.5 Integración con Java.....	39
3.8.6 Wetware.....	39
3.8.7 Productividad.....	40
3.9 Resumen.....	40
Capítulo 4. Creación y arranque del sistema.....	41
4.1 Generación de la aplicación.....	41
4.2 Lógica de negocios: creación de las clases de dominio.....	45
4.3 Interacción con el mundo exterior: controladores y vistas.....	47
4.4 Persistencia: almacenamiento de datos en Grails.....	53
4.5 Resumen.....	60
Parte 3. Fundamentos Básicos de Grails.....	61
 Capítulo 5. Dominio de una aplicación web.....	63
5.1 Creación de una clase de dominio.....	63
5.2 Adición de atributos.....	64
5.2.1 Tipos de atributos.....	65
5.3 Consistencia e integridad: validación de datos.....	67
5.4 Estilizando la base de datos.....	70
5.5 Relaciones entre clases de dominio.....	74
5.5.1 Relación uno a uno.....	74
5.5.2 Relación muchos a uno.....	75
5.5.3 Relación uno a muchos.....	76
5.5.4 Relación muchos a muchos.....	78
5.6 Resumen.....	79
 Capítulo 6. Modelo Objeto-Relacional en Grails (GORM).....	80
6.1 Hibernate Framework y ORM.....	80
6.2 ORM en Grails.....	82
6.3 Métodos comunes a todas las clases de dominio.....	83
6.3.1 Create.....	83
6.3.2 Read.....	84
6.3.3 Update.....	85
6.3.4 Delete.....	85
6.4 Métodos dinámicos.....	86
6.5 Realización de consultas orientadas a objetos: Criteria.....	88
6.6 Hibernate Query Lenguaje (HQL).....	91
6.7 Resumen.....	92

Capítulo 7. Capa de presentación Web: Controladores y Groovy	
Server Pages.....	93
7.1 Generación de controladores.....	93
7.2 Análisis detallado de los controladores.....	94
7.2.1 index y list.....	94
7.2.2 create.....	96
7.2.3 save.....	97
7.2.4 show.....	99
7.2.5 edit.....	100
7.2.6 update.....	101
7.2.7 delete.....	104
7.3 Generación de las vistas.....	106
7.4 Plantillas usadas para la generación de controladores y vistas.....	107
7.5 Groovy Server Pages.....	109
7.6 Expresiones.....	109
7.7 Taglibs.....	111
7.7.1 Toma de decisiones.....	111
7.7.2 Iteraciones.....	112
7.7.3 Elementos HTML.....	114
7.7.3.1 Formularios.....	114
7.7.3.2 Campos de texto.....	116
7.7.3.3 Listas desplegables.....	117
7.7.3.4 Elementos de selección.....	118
7.7.3.5 Carga de archivos.....	119
7.7.3.6 Vínculos y recursos estáticos.....	120
7.7.4 Tablas.....	121
7.7.5 Manejo de errores en clases de dominio.....	122
7.7.6 Formato de datos.....	123
7.7.7 Uso de variables.....	124
7.8 Incorporando Web 2.0 mediante Ajax.....	125
7.8.1 g:remoteLink.....	127
7.8.2 g:remoteField.....	128
7.8.3 g:remoteFunction.....	129
7.8.4 g:formRemote.....	130
7.8.5 g:submitToRemote.....	130
7.9 Resumen.....	131
Parte 4. Tópicos avanzados de Grails.....	132
Capítulo 8. Desarrollo de API's tipo REST con Grails.....	133
8.1 ¿Que es un API?.....	133
8.2 Arquitectura REST.....	134
8.3 Implementación de un API tipo REST.....	135
8.4 Experimentando con el API.....	140
8.5 Seguridad en API's tipo REST.....	142
8.6 Ejemplo de la vida real: el API de Twitter.....	142

8.7 Resumen.....	144
Capítulo 9. Plugins.....	145
9.1 Introducción a los plugins de Grails.....	145
9.2 Creación dinámica de menús.....	146
9.3 Exportación de datos a diversos formatos.....	149
9.4 Uso de tablas dinámicas con JQGrid.....	152
9.5 Adición de Seguridad mediante Spring Security.....	155
9.6 Ingeniería inversa de bases de datos.....	160
9.7 Inclusión de otros plugins en el sistema.....	162
9.7.1 Uso de calendarios con el plugin Calendar.....	162
9.7.2 Despliegue de ayuda mediante Help-Ballons.....	163
9.7.3 Generación dinámica de gráficas con Google Chart y jQuery.....	164
9.8 Resumen.....	167
Conclusiones y perspectivas a futuro.....	168
Glosario de términos.....	170
Bibliografía.....	174
Anexo A: Instalación de Grails.....	179
<i>Prerrequisitos.....</i>	<i>179</i>
<i>Lista de pasos a seguir.....</i>	<i>179</i>
Anexo B: Diagrama de Entidades del Sistema (CSI) de PROMEP- UAEH.....	181

Índice de Figuras

Figura 1: Actividades del Director.....	14
Figura 2: Actividades de la Recepcionista.....	14
Figura 3: Actividades del Trabajador.....	15
Figura 4: Actividades del Administrador.....	15
Figura 5: Diagrama de Arquitectura de 3 capas utilizada por Grails.....	25
Figura 6: Página inicial de la aplicación.....	44
Figura 7: Página inicial de la aplicación con un vínculo al controlador.....	49
Figura 8: Pantalla principal del catálogo de Profesores.....	50
Figura 9: Página de creación de registros.....	50
Figura 10: Confirmación de la creación del registro.....	51
Figura 11: Formulario de edición de datos.....	102
Figura 12: Respuesta del servidor al invocar la acción list del API REST.....	138
Figura 13: Respuesta del servidor con el método show.....	139
Figura 14: Respuesta del servidor con el método show con un isbn inválido.....	140
Figura 15: Plugin Poster para la creación de peticiones HTTP.....	141
Figura 16: Página principal del API de Twitter.....	143
Figura 17: Respuesta del API de Twitter al solicitar el timeline público.....	144
Figura 18: Menú generado por el plugin "navigation".....	147
Figura 19: Aspecto del menú al hacer modificaciones.....	148
Figura 20: Barra de exportación de datos.....	151
Figura 21: Reporte generado por el plugin export.....	151
Figura 22: Tabla con JQGrid y Ajax.....	155
Figura 23: Solicitud de credenciales con Spring Security.....	158
Figura 24: Página de inicio de sesión de la aplicación Grails de PROMEP-UAEH...	159
Figura 25: Pantalla principal para un usuario tipo DES.....	160
Figura 26: Uso del plugin calendar.....	163
Figura 27: Uso del plugin help-ballons.....	164
Figura 28: Uso de Google Chart y jQuery para la visualización de gráficas.....	165
Figura 29: Uso de Google Chart para la generación de gráficas de indicadores.....	166

Índice de Tablas.

Tabla 1: Indicadores de desempeño.....	16
Tabla 2: Tipos de datos soportados por Grails.....	66
Tabla 3: Restricciones utilizadas en Grails.....	69
Tabla 4: Restricciones utilizadas en Grails (continuación).....	70
Tabla 5: Algunas convenciones utilizadas en la generación de la base de datos.....	71
Tabla 6: Algunas opciones de estilización de la base de datos en Grails.....	72
Tabla 7: Parámetros del método save.....	83
Tabla 8: Comparadores y Operadores soportados por los métodos dinámicos.....	87
Tabla 9: Comparadores y Operadores soportados por los métodos dinámicos (continuación).....	88
Tabla 10: Métodos soportados por Criteria.....	90
Tabla 11: Métodos en Grails para el uso de HQL.....	91

Parte 1 Introducción

En la Parte 1 del presente trabajo se introduce al lector en el panorama general de la temática planteada en el documento.

En el Capítulo 1 se da una introducción al contexto histórico del desarrollo de aplicaciones web, se explica la problemática asociada, se plantean los objetivos tanto general como específicos, justificación, metodología y herramientas tecnológicas. Al final se presenta un mapa general del contenido tratado a lo largo del documento.

En el Capítulo 2: Requerimientos, análisis y diseño del Sistema de Control y Seguimiento de Indicadores PROMEP-UAEH (CSI), se presentan las necesidades de la dirección de PROMEP-UAEH para la realización de una aplicación web. Se hace un análisis de estos requerimientos y posterior a éste se realiza un diseño preliminar de la aplicación.

Capítulo 1 Introducción al documento

Todo trabajo requiere de una explicación introductoria que empape al lector en el panorama general del mismo. En este capítulo se describen de forma general los principales lineamientos de este documento.

1.1 Los inicios en el desarrollo de aplicaciones web

La World Wide Web (WWW), o lo que se conoce comúnmente como Internet, apareció en los inicios de la década de los 90's. Las computadoras comenzaron a comunicarse entre sí desde cualquier parte del mundo; el intercambio de datos no solo podía hacerse a nivel local, sino que una persona que vivía en América podía enviar las fotos de su boda a un amigo radicado en Asia casi al mismo tiempo en el que las fotografías eran tomadas. Las posibilidades tecnológicas comenzaban a sobrepasar sus límites, generando así un nuevo campo en el ámbito de la informática: se iniciaba la red de redes.

A casi 20 años de surgimiento de Internet, es evidente su evolución, impacto e influencia en la vida de las personas. Antes era impensable hacer compras, transacciones bancarias, cursos académicos y foros de discusión a través de una red de computadoras. Hoy en día, las redes sociales como Twitter y Facebook gobiernan los hábitos diarios de los jóvenes (y no tan jóvenes) [PRE01]; sitios como Amazon, eBay y Mercado Libre propician la compra y venta de un sinnúmero de productos y servicios; dentro de pocos años toda la facturación se realizará de manera electrónica; el Cloud Computing (Cómputo en la Nube) está

incitando la migración de aplicaciones de escritorio y sistemas operativos a la red. Las tecnologías computacionales tienen una mayor aceptación conforme éstas se involucran en las actividades diarias de la gente.

La demanda de los usuarios de Internet ha crecido de forma exponencial en los últimos 10 años. En un mundo cuyo ritmo de trabajo es tan vertiginoso y cambiante, los sitios web necesitan generarse y ponerse en línea tan pronto como sea posible. Los desarrolladores de software han visto la necesidad de generar diversas herramientas de creación de aplicaciones informáticas, las cuales han cubierto en cierta medida la necesidad del desarrollo ágil de aplicaciones. Sin embargo, muchas de estas herramientas o frameworks involucran al programador en muchas tareas de configuración ajenas a las necesidades o requerimientos que su sitio web necesita, haciendo lento el proceso de desarrollo y retrasando la codificación de las partes propias y específicas de su aplicación.

La mayoría de las aplicaciones web requieren un proceso inicial de desarrollo bastante repetitivo, por lo que la automatización de dicho proceso puede ahorrar semanas e incluso meses de trabajo a todo un equipo de programadores, permitiéndoles concentrarse en otros aspectos que son pasados por alto durante la fase inicial del proyecto debido a cuestiones de tiempo o una mala planeación.

Este documento introduce al lector al desarrollo ágil de aplicaciones web mediante el uso de Grails [GRA01], una herramienta que agiliza muchas de las tareas de configuración y desarrollo de una aplicación

web. Grails está ganando muchos seguidores conforme evoluciona y demuestra su robustez, estabilidad y sobre todo, facilidad de uso.

1.2 Definición del Problema

La demanda de aplicaciones en Internet ha crecido de forma desmesurada en los últimos 10 años. Los sitios web necesitan ponerse en línea lo más pronto posible. La filosofía Ágil de Desarrollo de Software [AAL01] ha respondido con métodos, técnicas y herramientas que satisfacen tal demanda. Aunado a esto, la mayoría de las aplicaciones web actuales siguen un proceso similar al inicio de su implementación. La automatización de estos pasos permite al desarrollador concentrarse en los aspectos que son nuevos para su aplicación. Grails permite la agilización y automatización del desarrollo de aplicaciones web de forma rápida y transparente.

1.3 Justificación

Grails es una herramienta que permite el desarrollo ágil de aplicaciones y reduce el tiempo de entrega. La documentación existente para dicho framework está disponible en inglés y hay poca información en español. Asimismo, el acercamiento de las universidades en México con dicha herramienta es casi nulo, por lo que es importante contar con un documento de referencia para comenzar a utilizarla. Los estudiantes, programadores y desarrolladores de aplicaciones web se ven beneficiados con este documento con un alto nivel de productividad, estabilidad, robustez y fácil mantenimiento de sus sistemas desarrollados en Grails.

1.4 Objetivo General

Describir los fundamentos del desarrollo ágil de aplicaciones web mediante el uso de Grails Framework para implementar el Sistema de Control y Seguimiento de los Indicadores Estratégicos del Programa del Mejoramiento del Profesorado (PROMEPE).

1.5 Objetivos Específicos

- Describir los fundamentos de Grails mediante el análisis de la estructura de los proyectos y los componentes de los mismos para la implementación ágil de aplicaciones web robustas y portables.
- Comprender las ventajas y desventajas del desarrollo ágil de aplicaciones.
- Visualizar la aplicación de Grails mediante el estudio de un caso práctico implementado en la Dirección de PROMEP-UAEH.
- Modelar las entidades de un sistema mediante clases de dominio.
- Conocer el paradigma ORM (Object-Relational Mapping) y su implementación en Grails.
- Aplicar la tecnología GSP (Groovy Server Pages) para el desarrollo de páginas web dinámicas.
- Implementar API's tipo REST para la exposición y acceso remoto de datos.

- Utilizar complementos (plugins) para exponenciar la velocidad de desarrollo en una aplicación web.

1.6 Metodología

La metodología utilizada para la redacción del documento está basada en la serie de libros Deitel & Deitel [DEI07], denominada *Metodología de código activo*. Ésta consiste en explicar un poco de teoría acerca de un tópico y visualizar su aplicación por medio de un fragmento de código.

Para la implementación del caso práctico en la Dirección de PROMEP-UAEH, se usa una mezcla de diversas metodologías de desarrollo de software, entre las cuales destacan:

- SCRUM [SCR01], metodología basada en una técnica de rugby ampliamente utilizada en entornos de desarrollo ágil [PRE02].
- Modelo Incremental, otra metodología ágil que entrega artefactos parciales de software en lapsos determinados de tiempo [PRE03] [IAN05].
- Modelo en Cascada, una de las primeras metodologías empleadas en el desarrollo de software y que representa la base de la gran mayoría de ellas [PRE04].
- Desarrollo dirigido por características, el cual se basa en la modularidad de los requisitos del software que se pueden entregar en 2 semanas o menos [PRE05].

1.7 Herramientas tecnológicas

La herramienta tecnológica utilizada en este documento es Grails, un software gratuito y de código abierto que a su vez está basado en las siguientes tecnologías:

- Spring Framework [SPR01], diseñado para facilitar el uso de la plataforma Java Enterprise Edition (sección 3.5).
- Hibernate Framework [HIB01], creado para el uso de bases de datos relacionales mediante el paradigma de la Programación Orientada a Objetos (sección 3.6).
- Lenguaje de Programación Groovy [GRO01], un lenguaje dinámico basado en Java, Ruby, Python y Smalltalk (sección 3.7)

Al ser un framework basado en Groovy, Grails requiere la instalación del Kit de Desarrollo de Software de Java (JDK, por sus siglas en inglés). El JDK contiene la máquina virtual de Java (JVM) y todas las bibliotecas de clases necesarias para la correcta ejecución de aplicaciones. El *Anexo A: Instalación de Grails*, contiene las instrucciones necesarias para la descarga e instalación del JDK y de Grails.

1.8 Descripción del trabajo

El presente trabajo está dividido en 4 partes:

- Parte 1: Introducción.
- Parte 2: Introducción a Grails.

- Parte 3: Fundamentos Básicos de Grails.
- Parte 4: Tópicos avanzados de Grails.

La Parte 1 es una introducción al documento. Se plantea la problemática encontrada en el desarrollo de aplicaciones web y se propone una solución mediante el uso de Grails. Asimismo, se dan a conocer los requerimientos del sistema desarrollado en la Dirección de PROMEP-UAEH.

La Parte 2 introduce al lector a Grails. Se habla acerca de los frameworks antecesores que motivaron su creación, sus ventajas y desventajas respecto a otras herramientas de desarrollo. Asimismo, se inicia el desarrollo de la aplicación web requerida en PROMEP-UAEH para su administración interna. Esta aplicación es utilizada a lo largo de todo el documento como referencia para la explicación de los diversos temas tratados.

La Parte 3 cubre las bases de Grails: Clases de Dominio, que modelan las entidades obtenidas en el proceso de diseño de un proyecto de software; el paradigma Object-Relational Mapping en Grails (GORM, por sus siglas en inglés), una tecnología basada en Hibernate que permite manejar la persistencia de las clases de dominio; asimismo, se habla de Groovy Server Pages (GSP), una herramienta similar a Java Server Pages (JSP) pero con capacidades superiores y más sencillas de utilizar.

La Parte 4 abarca algunos temas avanzados en el desarrollo web con Grails. Se explica el tratamiento de API's de tipo REST con la

nomenclatura XML, tecnología utilizada para la conexión e interacción de diversas aplicaciones. Para cerrar el estudio de Grails, se introduce al lector a una de las características más poderosas: el uso de plugins. Dichos complementos tienen funcionalidades muy variadas que van desde la exportación de tablas a diversos formatos hasta la ingeniería inversa de bases de datos existentes desde diferentes DBMS. Asimismo, se describe la implementación final del caso práctico de la Dirección de PROMEP-UAEH introducido en el Capítulo 2. En dicha institución se generaron aplicaciones web y se integraron bases de datos existentes a Grails. En este caso de estudio se utiliza gran parte del contenido descrito en los capítulos anteriores.

Finalmente, se dan a conocer las conclusiones acerca del documento y las perspectivas a futuro acerca del desarrollo ágil de aplicaciones con Grails. Además, se incluye un anexo que describe paso a paso el proceso de descarga e instalación del JDK y de Grails y un anexo que muestra el diagrama de entidades generado para el ejemplo práctico.

Capítulo 2 Requerimientos, análisis y diseño del Sistema de Control y Seguimiento de Indicadores PROMEP-UAEH (CSI)

La dirección de PROMEP-UAEH maneja una gran cantidad de datos e información. En algunos procesos, el manejo de la información sin ayuda de la tecnología se vuelve una tarea difícil, por lo que es necesario llevar a cabo el desarrollo de aplicaciones que faciliten dichos procesos. En este capítulo se describen los requerimientos de éstos, así como el análisis y diseño correspondiente.

2.1 PROMEP

De acuerdo a [PRO01], el Programa de Mejoramiento del Profesorado (PROMEP) está dirigido a elevar permanentemente el nivel de habilitación del profesorado, con base en los perfiles adecuados para cada subsistema de educación superior. Se busca que al impulsar la superación permanente en los procesos de formación, dedicación y desempeño de los cuerpos académicos de las instituciones, se eleve la calidad de la educación superior .

El PROMEP responde a los propósitos del Programa Sectorial de Educación 2007-2012, que establece como uno de sus objetivos estratégicos: "Elevar la calidad de la educación para que los estudiantes mejoren su nivel de logro educativo, cuenten con medios para tener acceso a un mayor bienestar y contribuyan al desarrollo nacional", y como objetivo particular: "Fortalecer los procesos de

habilitación y mejoramiento del personal académico".

Para lograr su cometido:

- Otorga becas nacionales y para el extranjero a profesores de carrera de las universidades públicas, para la realización de estudios de posgrado en programas de reconocida calidad.
- Apoya la contratación de nuevos profesores de tiempo completo que ostenten el grado académico de maestría o de doctorado (preferentemente) y la reincorporación de exbecarios PROMEP a su institución después de haber terminado sus estudios en tiempo dotándolos con los elementos básicos para el trabajo académico.
- Reconoce con el Perfil Deseable a profesores que cumplen, con eficacia y equilibrio sus funciones de profesor de tiempo completo, como atender la generación y aplicación del conocimiento, ejercer la docencia y participar en actividades de tutorías y gestión académica.
- Apoya el fortalecimiento de Cuerpos Académicos, la integración de redes temáticas de colaboración de Cuerpos Académicos, incluyendo el apoyo para gastos de publicación y becas Post-Doctorales.

2.2 Requerimientos

Antes de escribir cualquier línea de código, antes de aventurarse a

diseñar una página web, una interfaz gráfica de escritorio o una base de datos, es indispensable saber qué es lo que se necesita. La dirección de PROMEP-UAEH presenta 3 requerimientos:

- La administración de la correspondencia entrante.
- El seguimiento de indicadores y evaluaciones de cuerpos académicos.
- La realización de un tablero de control de los indicadores vinculados a objetivos estratégicos de la institución.

Las siguientes secciones detallan los requerimientos de cada punto.

2.2.1 Administración de correspondencia

Diariamente, la dirección de PROMEP-UAEH recibe correspondencia remitida desde diversos puntos que van desde oficinas manejados dentro de la misma Universidad hasta paquetes enviados desde distintas zonas de la República Mexicana.

Este proceso requiere de la participación de varias personas y cada una de ellas desarrolla actividades que deben ser apoyadas para lo siguiente:

- Una persona con el rol de recepcionista recibe la correspondencia y la debe hacer llegar a su respectivo destinatario de forma eficiente y eficaz.
- El director del área debe tener un mejor control, seguimiento y supervisión de la correspondencia recibida y emitida.

- El personal debe dar atención, seguimiento y consulta de la correspondencia que le ha sido asignada.
- Para el caso de la correspondencia recibida por parte de instancias propias de la Universidad, los profesores y líderes de cuerpos académicos deben de conocer el estado en que se encuentran sus solicitudes presentadas.

Para sistematizar y automatizar este proceso se requiere desarrollar e implementar una nueva aplicación basada en Web accesible desde cualquier computadora con acceso a Internet. Dicha aplicación debe tener la siguiente funcionalidad:

- Que un usuario con el rol de recepcionista reciba la correspondencia y registre su alta en el sistema.
- Otro usuario con el rol de director debe asignar la correspondencia a su destinatario correspondiente.
- El usuario destinatario, que puede tener cualquier rol, recibe la correspondencia y decide qué hacer con ella:
 - Si ha recibido la correspondencia pero aún no la ha atendido, debe actualizar el estado de la correspondencia en el sistema como “En Espera”.
 - Si ha recibido y atendido la correspondencia, debe actualizar el estado de la correspondencia en el sistema como “Terminado”.

- El usuario administrador, el director y el recepcionista pueden consultar el listado de correspondencia de todos los usuarios, mientras que el resto únicamente puede visualizar su propia correspondencia.
- El usuario administrador puede realizar cualquier operación.

Las figuras 1, 2, 3 y 4 muestran los casos de uso de los roles requeridos en la aplicación y sus actividades correspondientes.

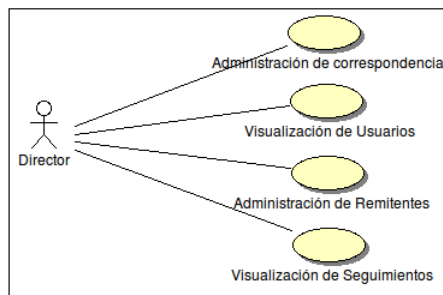


Figura 1: Actividades del Director.

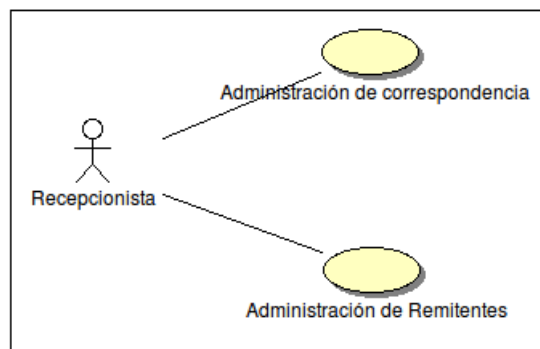


Figura 2: Actividades de la Recepcionista.

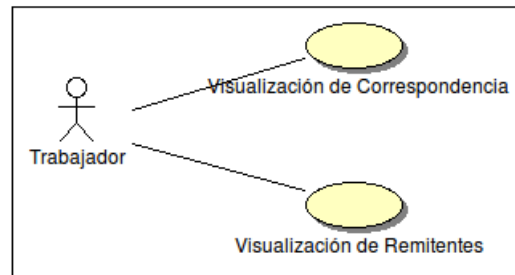


Figura 3: Actividades del Trabajador.

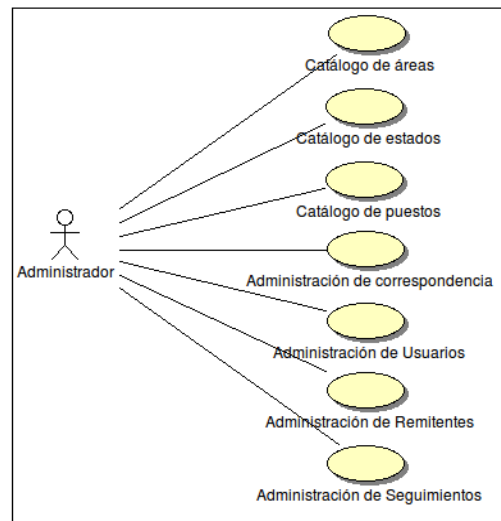


Figura 4: Actividades del Administrador.

2.2.2 Seguimiento de indicadores

Dentro de la dirección de PROMEP-UAEH se cuenta con el Sistema Institucional para el Seguimiento Académico y Científico de la UAEH (SISAC) [COR07], a través del cual se concentra, administra y

distribuye toda la información relacionada con la capacidad académica y producción científica y académica de los profesores y cuerpos académicos de la UAEH. Uno de los aspectos manejados dentro del SISAC es el uso de indicadores de desempeño, los cuales permiten visualizar de forma cuantitativa el comportamiento actual.

Los indicadores de desempeño que deben ser administrados se muestran en la Tabla 1.

Indicador	CÁLCULO
PTC con posgrado	$PTC \text{ con posgrado} = PTC \text{ con maestría} + PTC \text{ con doctorado}$
% PTC con posgrado	$\% PTC \text{ con posgrado} = PTC \text{ con posgrado} / \text{Total de PTC}$
PTC con doctorado	Total de PTC con doctorado
% PTC con doctorado	$\% PTC \text{ con doctorado} = PTC \text{ con doctorado} / \text{Total de PTC}$
PTC con perfil	$PTC \text{ con perfil} = \text{Total de PTC con perfil maestría} + \text{Total de PTC con perfil doctorado}$
% PTC con perfil	$\% PTC \text{ con perfil} = PTC \text{ con perfil} / \text{Total de PTC}$
PTC en SNI	Total de PTC en SNI
% PTC en SNI	$\% PTC \text{ en SNI} = PTC \text{ en SNI} / \text{Total de PTC}$
CA consolidados	Total de CA consolidados
% CA consolidados	$\% CA \text{ consolidados} = CA \text{ consolidados} / \text{Total de CA}$
CA en consolidación	Total de CA en consolidación
% CA en consolidación	$\% CA \text{ en consolidación} = CA \text{ en consolidación} / \text{Total de CA}$
PTC en CA	Total de PTC en CA
% PTC en CA	$\% PTC \text{ en CA} = PTC \text{ en CA} / \text{Total de PTC}$

Tabla 1: Indicadores de desempeño.

Básicamente, se requiere una aplicación en línea que permita realizar las siguientes tareas:

- Administración de catálogos de los siguientes aspectos:
 - Convocatorias.
 - Indicadores.
- Administración de los usuarios de la aplicación.
- Seguimiento de evaluaciones de cuerpos académicos.
- Seguimiento de participación en convocatorias.
- Seguimiento de los indicadores estratégicos.
- Tablero de Control de indicadores estratégicos.
- Generación de reportes en diversos formatos.

La administración de catálogos se refiere a las 4 operaciones básicas de una base de datos: altas, bajas, modificaciones y consultas (Create, Read, Update, Delete: CRUD).

Uno de los objetivos de la aplicación es la visualización de los indicadores a través de un tablero de control. Mientras que las pantallas administrativas sirven para manipular la información, el tablero de control muestra una serie de gráficas descriptivas relacionadas con dicha información.

2.2.3 Tablero de control

Los indicadores estratégicos manejados en la aplicación descrita en la sección 2.2.2 necesitan reflejar su comportamiento histórico de forma gráfica, para que así se pueda analizar el cambio de los mismos

a través del tiempo y así favorecer la toma de decisiones por parte de los directivos a nivel institucional, así como al interior de las Escuelas e Institutos.

De acuerdo a [WIK02] y [KAP99]:

“El tablero de control (TdeC) es una herramienta, del campo de la administración de empresas, aplicable a cualquier organización y nivel de la misma, cuyo objetivo y utilidad básica es diagnosticar adecuadamente una situación. Se lo define [sic] como el conjunto de indicadores cuyo seguimiento y evaluación periódica permitirá contar con un mayor conocimiento de la situación de su empresa o sector apoyándose en nuevas tecnologías informáticas”

Los indicadores estratégicos que deben ser incluidos en el tablero de control son los siguientes:

- Profesores de Tiempo Completo con Posgrado.
- Profesores de Tiempo Completo con Doctorado.
- Profesores de Tiempo Completo con Perfil.
- Profesores de Tiempo Completo en el Sistema Nacional de Investigadores.
- Cuerpos Académicos Consolidados.
- Cuerpos Académicos en Consolidación.
- Profesores de Tiempo Completo en Cuerpos Académicos.

Asimismo, las gráficas utilizadas deben reflejar los siguientes aspectos:

- Un semáforo que muestre el estado actual del indicador.
- Valores históricos del indicador en el período 2000-2010.
- Valores absolutos empleados para calcular el indicador en el período 2000-2010.
- Consultar el tablero de control a nivel institucional y de institutos.

2.2.4 Seguimiento de evaluaciones de cuerpos académicos.

El seguimiento de las evaluaciones de cuerpos académicos por parte de la coordinación académica del PROMEP nacional es necesario para administrar y distribuir los dictámenes emitidos por los comités evaluadores de los cuerpos académicos. Los directivos a nivel institucional y de los institutos deben tener acceso a dicha información para que sea considerada en los procesos de planeación y seguimiento de los cuerpos académicos.

La información que se necesita manejar es:

- Nombre del cuerpo académico.
- Año de evaluación.
- Grado de consolidación propuesto por la UAEH.
- Grado dictaminado por el comité de evaluación.

- Dictamen recibido.

2.2.5 Seguimiento de participación en convocatorias.

El seguimiento de la participación en las convocatorias de profesores y cuerpos académicos es necesaria para conocer la participación de profesores y cuerpos académicos en las convocatorias emitidas por el PROMEP nacional. Este seguimiento permite conocer los niveles de participación en cada convocatoria por año de participación.

Se requiere manejar la siguiente información:

- Nombre del profesor o cuerpo académico.
- Año de participación.
- Convocatoria en la que participa.
- Resultado de la participación.

Es necesario que se habilite la consulta por convocatoria y año, así como por tipo de participante (profesor o cuerpo académico). Además, se debe presentar un gráfico descriptivo que muestre el número de participantes por año en la convocatorias de PROMEP-UAEH.

2.3 Análisis

Una vez obtenidos y entendidos los requerimientos, se procede a realizar el análisis de los mismos. El análisis de un sistema responde a la pregunta *¿qué se necesita?*. Los pasos a seguir son los siguientes:

- **Modelado de entidades:** consiste en la identificación de objetos del mundo real involucrados en los procesos descritos en los requerimientos para su representación computacional.
- **Modelado de procesos:** consiste en la identificación de las diversas actividades que realizan las entidades tanto de forma individual como en conjunto para lograr el objetivo del sistema.

2.3.1 Modelado de entidades

Debido a que la dirección de PROMEP-UAEH ya cuenta con una base de datos, el modelado de entidades se resume en la adaptación de las tablas existentes a la aplicación. Con base en los requerimientos del sistema, se obtiene una lista de objetos y las relaciones existentes entre sí. Los sustantivos que aparezcan en la descripción son identificados como posibles entidades. El resultado de éste análisis arroja el diagrama mostrado en el *Anexo B: Diagrama de Entidades del Sistema (CSI) de PROMEP-UAEH*. Existen otras entidades involucradas, pero para los objetivos de las aplicaciones, únicamente se muestran las entidades relevantes.

2.3.2 Modelado de procesos

Dada la naturaleza ágil de la implementación del sistema, la descripción realizada en la sección 2.2 es suficiente para completar el modelado de procesos.

2.4 Diseño

Dado que el tema principal de este documento es el uso de Grails para el desarrollo de aplicaciones web, el diseño de las aplicaciones requeridas en la dirección de PROMEP-UAEH está basado en la arquitectura ofrecida por el framework. Dicha arquitectura utiliza 3 capas:

- Capa de lógica de negocios.
- Capa de control.
- Capa de presentación.

Las siguientes secciones explican a detalle cada una de ellas.

2.4.1 Capa de lógica de negocios

El núcleo de una aplicación que utiliza una arquitectura de 3 capas es la lógica de negocios. En ella se modelan las entidades del mundo real y se realizan las tareas necesarias para su manipulación, tal como su creación, almacenado y actualización. Dentro del entorno de Grails, esta capa está representada por las clases de dominio, sus métodos relacionados con la persistencia y el acceso a la base de datos. En algunas aplicaciones más robustas, se incluye la capa de servicios, la cual controla el acceso de cualquier programa a la lógica de negocios, facilitando el mantenimiento y promoviendo la reutilización de código.

Gracias a la facilidad que ofrece Grails para el acceso a cualquier

tipo de base de datos y a la existencia de un plugin de ingeniería inversa de bases de datos (sección 9.6), la generación del código correspondiente a la estructura básica de la lógica de negocios es una tarea que puede automatizarse.

2.4.2 Capa de control

Las aplicaciones web necesitan una forma de comunicación entre la lógica de negocios y el resultado final mostrado al usuario. En Grails, la capa de control es la encargada de lograr la interactividad entre las clases de dominio (o los servicios) y la capa de presentación a través de clases controladoras (o controladores, como se maneja en las secciones 7.1 y 7.2). La capa de control tiene la habilidad de recibir peticiones HTTP, procesarlas y emitir un resultado utilizando el mismo protocolo. El contenido puede abarcar desde simples caracteres hasta complejos archivos multimedia.

Tomando como base las clases de dominio generadas por Grails, el comando *grails generate-controller* (sección 7.1) crea por defecto el código necesario para la interacción entre clases de dominio y la capa de presentación, por lo que la codificación del esqueleto inicial de la capa de control, al igual que la capa de negocios, puede automatizarse.

2.4.3 Capa de presentación

En una aplicación web, el resultado final mostrado al usuario generalmente se representa por medio de interfaces gráficas de

usuario (GUI, por sus siglas en inglés) en un navegador web. La capa de presentación en Grails viene representada por la tecnología Groovy Server Pages o GSP (sección 7.5). La capa de control entrega un modelo que un archivo GSP puede manipular mediante código Groovy (sección 7.6) o a través del uso de taglibs (sección 7.7). Asimismo, la capa de presentación se representa por medio de archivos XML (Figura 17), texto en formato JSON, PDF, e inclusive audio y video.

Tomando como base las clases de dominio generadas por Grails, el comando *grails generate-views* (sección 7.3) crea por defecto el código HTML y GSP necesario para la capa de presentación, por lo que la codificación del esqueleto inicial de las interfaces gráficas de usuario, al igual que la capa de negocios y la de control, puede automatizarse.

2.4.4 Diagrama de la arquitectura de 3 capas

La Figura 5 muestra la interacción entre la capa de lógica de negocios, la capa de control y la capa de presentación. Cabe destacar que cada capa tiene conocimiento únicamente de la capa inmediata inferior. Esto quiere decir, por ejemplo, que los servicios sólo tienen acceso a las clases de dominio y que la base de datos no tiene conocimiento de las capas que acceden a ella.

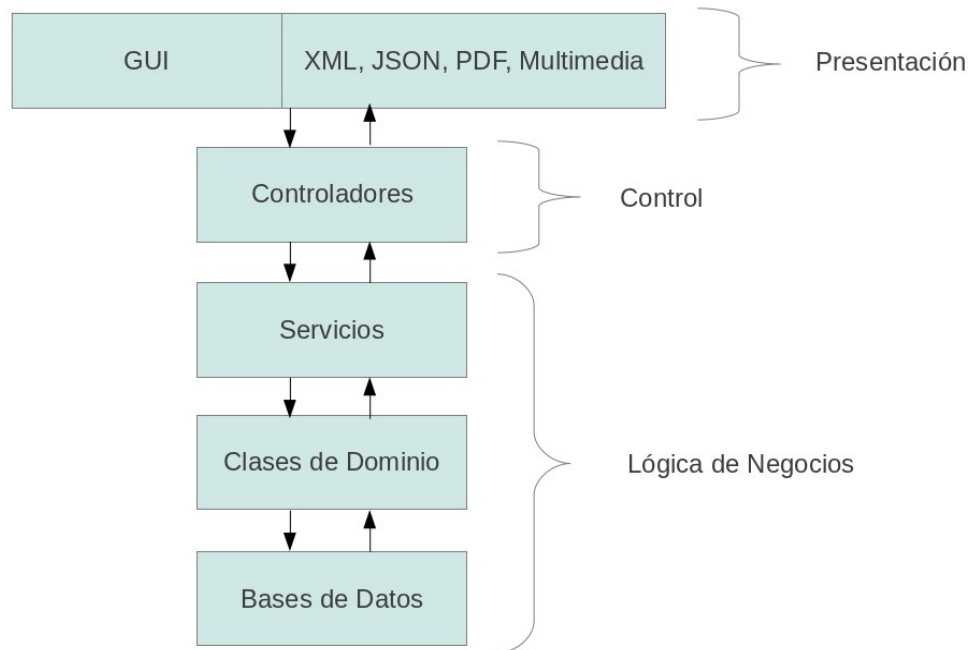


Figura 5: Diagrama de Arquitectura de 3 capas utilizada por Grails.

2.5 Resumen

La dirección de PROMEP-UAEH tiene necesidades que pueden solventarse mediante la creación de aplicaciones web. El análisis y diseño presentados en este capítulo sirven como base para la generación y puesta en marcha de dichas necesidades.

Parte 2 Introducción a Grails

Todo tiene un inicio. Todo surge por una razón. Cuando un nuevo paradigma llega a romper las reglas establecidas, viene con ideas novedosas que generalmente mejoran y facilitan las prácticas actuales. Grails no es la excepción. Grails ha llegado para romper el paradigma de que el desarrollo de aplicaciones web en Java es un proceso complicado.

A través del *Capítulo 3: ¿Por qué Grails?*, se da una introducción a los frameworks de Java utilizados desde el surgimiento de la plataforma Enterprise de dicho lenguaje. Se exploran las características principales de cada uno de ellos y al final, se explica con detalle qué es Grails, sus atributos, aplicaciones, ventajas y desventajas.

En el *Capítulo 4: Creación y arranque del sistema*, se inicia la creación de la aplicación web para la administración de PROMEP-UAEH. El desarrollo de dicha aplicación tiene como metas mostrar las capacidades de Grails y motivar al lector en el aprendizaje de las mismas.

Capítulo 3 ¿Por qué Grails?

Para entender y apreciar el origen, funcionamiento y poder de Grails, es necesario conocer un poco de historia acerca de las tecnologías de desarrollo de aplicaciones Web en las cuales está basado. Las siguientes secciones guían al lector a través de las principales herramientas utilizadas en dicho desarrollo.

3.1 El inicio de todo: El lenguaje de programación Java

Java surgió en el año 1995 en una presentación de Sun Microsystems en Sand Hill Road, Menlo Park, California. Su creador, James Gosling [JGO01], concibió la idea de generar un lenguaje de programación que fuera completamente orientado a objetos, portable entre diversas plataformas y que pudiera ejecutarse en diversos dispositivos electrónicos, principalmente móviles.

La sintaxis de Java se asemeja mucho a la de C y C++, lenguajes en los que está escrito. Java posee características que lo hacen único [DEI07]:

- No utiliza apuntadores, liberando al usuario de la administración directa de la memoria.
- Java es compilado en *bytecodes* en archivos con extensión *.class*. Dichos archivos son interpretados por un programa denominado Máquina Virtual de Java (JVM, por sus siglas en inglés). Ésta JVM está escrita de forma nativa para cada sistema operativo, lo que hace portables a las aplicaciones

escritas en Java: “*Write Once, Run Everywhere*” es uno de los lemas de Sun Microsystems para dicho lenguaje.

- Java no permite la sobrecarga de operadores tal como lo hace C++.
- Java no usa herencia múltiple como tal, sino que posee un mecanismo de implementación de varias clases conocidas como *interfaces*.
- Se utiliza un mecanismo automático de liberación de memoria conocido como *garbage collector*, el cual detecta referencias no utilizadas o nulas y libera sus recursos. Con este mecanismo, el programador evita problemas de fugas de memoria.
- El rendimiento de una aplicación de Java es competente con una aplicación escrita en C++. Se ha demostrado que Java es 1.1 veces más lento que C++¹, y en muchos casos, es más rápido que C++ [TSS04].

Java se divide en 3 partes de acuerdo al entorno de ejecución: Java Standard Edition (JSE), cuyo desarrollo se enfoca a aplicaciones de escritorio; Java Enterprise Edition (JEE), cuyo objetivo es hacia aplicaciones empresariales que manejan bases de datos, servicios web, mensajería asíncrona, correo electrónico, entre otras funcionalidades; y Java Micro Edition (JME), cuyas aplicaciones se ejecutan en dispositivos móviles como son celulares, PDA's,

¹ La version 1.0 de Java era de 20 a 40 veces más lenta que C++ [KGP05]. A partir de la versión 1.5, el rendimiento se logró optimizar hasta 1.1 veces. Al momento de escribir este documento, la version de Java es la 1.6.0.24.

Blackberries, Smartphones, entre otros.

Grails utiliza JEE para gran parte del desarrollo de aplicaciones web, así como JSE para la creación de módulos que se comunican con otras aplicaciones.

3.2 Java Enterprise Edition

Java Enterprise Edition (JEE) es una plataforma ampliamente utilizada para la programación de aplicaciones de servidor escritas en Java. Dicha plataforma difiere de la versión JSE en que la primera añade bibliotecas de clases que proporcionan funcionalidad para aplicaciones multicapa, distribuidas y tolerantes a fallas.

Las aplicaciones para JEE se ejecutan en un *servidor de aplicaciones*, el cual es un programa escrito en Java que permite el despliegue de páginas web, conexiones a bases de datos, administración de sesiones, entre otras cosas. Los archivos de la aplicación JEE se colocan en una ubicación propia del servidor de aplicaciones y éste se encarga de desplegarla. Los servidores de aplicaciones más populares son:

- Apache Tomcat [APT01].
- GlassFish [GLA01].
- Websphere [WEB01].
- Jetty [JET01].
- Weblogic [WLO01].

Algunas de las bibliotecas de clases de JEE son:

- JDBC (Java Database Connectivity) para tratamiento de bases de datos [JDB01].
- RMI (Remote Method Invocation) para el uso de procedimientos remotos [RMI01].
- JavaMail para correo electrónico [JMA01].
- JMS (Java Message Service) para el uso de colas de mensajes [JMS01].
- JAX-WS (Java API for XML Web Services) para la implementación de servicios Web [JAX01].
- JSP (JavaServer Pages) para la creación de páginas web dinámicas [JSP01].

Se recomienda consultar [JEE10] para un tratamiento más profundo de Java Enterprise Edition.

3.3 La tecnología base: Servlets

El desarrollo de aplicaciones web con JEE se basa en la programación de clases que se encargan de gestionar las peticiones HTTP hechas al servidor. Dichas clases son denominadas *Servlets*. Los servlets son una tecnología moderna capaz de administrar solicitudes y respuestas mediante el protocolo HTTP. Los servlets son puestos en marcha mediante servidores de aplicaciones y archivos de configuración que indican los servlets a utilizar para la gestión de la aplicación web.

Como los servlets son clases de Java, tienen la facultad de procesar cualquier tipo de lógica de negocios y compartir la entrada y salida de dicha lógica con un sitio web. Esta característica es la esencia de Java Enterprise Edition: recibir datos de entrada mediante HTTP, procesar dichos datos mediante la lógica de negocios implementada con Java y enviar una respuesta generada por dicha lógica nuevamente por medio de HTTP.

Un servlet escrito de forma manual puede ser una tarea difícil. Los frameworks para aplicaciones web facilitan dicha tarea abstrayendo gran parte de la funcionalidad de ésta tecnología.

3.4 Frameworks: encapsulado y abstracción de funcionalidad

Con la llegada de JEE, los servlets y los servidores de aplicaciones, el desarrollo web con Java comenzó a difundirse entre la comunidad de desarrolladores. Con el paso del tiempo, éstos comenzaron a percatarse de que la codificación de aplicaciones web involucraba la repetición pasiva de muchos pasos, especialmente en la configuración inicial. Asimismo, la experiencia adquirida proporcionó a los programadores una serie de patrones de diseño que podían implementarse con el fin de mejorar y acelerar la implementación de software. Aunado a esto, la velocidad de desarrollo de aplicaciones web comenzó a incrementarse cuando la Internet fue explotada no solo con fines informativos, sino con fines lucrativos, comerciales, académicos, sociales, etc.

Teniendo en cuenta que la velocidad de desarrollo podía

incrementarse mediante la automatización de ciertos procesos y la aplicación de patrones de diseño, comenzaron a surgir los *frameworks de aplicaciones*. Un framework es un conjunto de herramientas diseñado para acelerar, mejorar y simplificar el desarrollo de software. Pueden ser de propósito general o propios de un área. Los frameworks de propósito general son la solución a la creación de la estructura general de programas, mientras que los frameworks propios de un área se enfocan en uno o varios módulos.

Desde el surgimiento de Java, el número de frameworks ha crecido de forma desmesurada. Dos de ellos han adquirido bastante popularidad debido a su estabilidad, robustez y facilidad de uso en el desarrollo de aplicaciones web: Spring, que es un framework de propósito general para el desarrollo de aplicaciones web y de escritorio², y Hibernate, un framework diseñado para manejar bases de datos con el paradigma de Programación Orientado a Objetos. Las siguientes secciones explican dichas herramientas.

3.5 Facilitando JEE: Spring Framework

El desarrollo de aplicaciones web con JEE contenía detalles que lo hacían lento y repetitivo. Teniendo esto en cuenta, Rod Johnson, un desarrollador férreo de JEE desde 1996, publicó en su libro *Expert One-on-One J2EE Design and Development* [SPR02] una serie de herramientas que facilitaban, aceleraban y simplificaban el desarrollo de aplicaciones web en JEE. Este conjunto de herramientas fueron la

² Al momento de escribir este documento, Spring Source estaba desarrollando Spring Mobile, un módulo para la creación de aplicaciones dirigidas hacia dispositivos móviles.

base de lo que más adelante sería Spring Framework.

Spring es un conjunto de herramientas que promueven la separación de los módulos de una aplicación mediante la *Inyección de Dependencias* [DIN04], un concepto relativamente nuevo que propone el bajo acoplamiento entre componentes, así como su reutilización dentro del mismo programa. Por otra parte, Spring Framework abstrae mucha de la funcionalidad de JEE mediante la automatización de ésta o proporcionando versiones más sencillas.

En los últimos años, Spring ha ganado popularidad entre los desarrolladores de software y es cada vez más utilizado en la creación de aplicaciones web. Asimismo, Spring no se ha limitado al desarrollo Web en Java, sino que ha implementado varios proyectos relacionados con otras herramientas, lenguajes y entornos. Ejemplos de esto son:

- Spring BlazeDS Integration para el desarrollo con Adobe Flex [FLX01].
- Spring Python [PYT01].
- Spring .NET [SDN01].
- Spring Mobile para dispositivos móviles [SMO02].

Para una mejor referencia, se puede consultar [SPR01].

3.6 Persistencia Orientada a Objetos: Hibernate Framework

En JEE, el uso de conexiones a bases de datos es una tarea muy común. Esto se logra utilizando el API JDBC. Su funcionalidad básica consiste en realizar una conexión mediante un controlador (driver) y

por medio del mismo hacer peticiones al gestor de base de datos mediante SQL (Structured Query Language). Una vez hechas las peticiones, se traducen y se procesan los resultados mediante Java.

Existen varios inconvenientes al utilizar JDBC. En primer lugar, la gran variedad de gestores de bases de datos provoca que cada uno de ellos contenga versiones diferentes de SQL, lo cual aminora la portabilidad de una aplicación y dificulta su mantenimiento. Asimismo, la extracción de los datos obtenidos mediante las peticiones es un proceso redundante y dependiente del tipo de gestor utilizado. Por ejemplo, mientras que MySQL [MSQ01] representa números enteros mediante *integer*, SQL Server [MSS01] lo hace con *numeric*.

Hibernate es un framework enfocado al uso de bases de datos desarrollado por JBoss que elimina los problemas antes mencionados. Utiliza el paradigma *Object-Relational Mapping* (ORM, por sus siglas en inglés), el cual le permite a un programa manipular una base de datos relacional con el paradigma de la programación orientada a objetos. Con ello, a nivel de la aplicación, las tablas se vuelven objetos, los registros se convierten en atributos del objeto, las llaves foráneas se transforman en asociaciones entre objetos y las consultas se traducen en llamadas a métodos.

Hibernate libera al desarrollador de la escritura del 95% de sentencias SQL, lo que aumenta la velocidad de desarrollo y portabilidad entre gestores de bases de datos. Asimismo, Hibernate proporciona su propio lenguaje de consultas denominado HQL

(Hibernate Query Language), el cual es muy parecido a SQL pero con la diferencia de que es completamente orientado a objetos.

El sitio oficial de Hibernate [HIB01] es una excelente referencia para un tratamiento a detalle de este framework.

3.7 La evolución natural: Lenguaje de programación Groovy

Java ha mostrado ser un lenguaje de programación poderoso y de alto nivel. Sin embargo, posee características que en algunas ocasiones vuelve redundante la codificación. Algunos lenguajes orientados a objetos, como Python, SmallTalk y Ruby, poseen características dinámicas que Java no tiene, lo que ocasiona que muchos desarrolladores opten por cambiar de lenguaje.

Groovy surgió en el año 2003 gracias a la iniciativa de James Strachan y Bob McWhirter . La intención de dicho lenguaje no es ser el sucesor de Java, ya que toda la funcionalidad de éste se encuentra disponible en Groovy. Asimismo, Groovy no está escrito sobre Java (asi como Java está escrito en C y C++), sino que es otro lenguaje de programación para la JVM, lo que lo hace completamente compatible con aplicaciones Java existentes y viceversa.

Groovy posee características de Java, Ruby, Python y SmallTalk en un sólo lenguaje, lo que lo convierte en un lenguaje poderoso y fácil de usar. La curva de aprendizaje de Groovy para los desarrolladores de Java es relativamente pequeña, lo cual les permite crear aplicaciones en Groovy de forma inmediata. Los desarrolladores de Grails eligieron

a Groovy como su lenguaje base debido a su poder y dinamismo.

Se recomienda consultar la página oficial de Groovy [GRO01] para un tratamiento más profundo acerca de este lenguaje.

3.8 Descubriendo el grial: Grails

El uso compartido de Spring y Hibernate es muy común en aplicaciones de JEE, ya que esto acelera y facilita el desarrollo de aplicaciones web. Ambos frameworks utilizan XML para sus módulos de configuración³. Para algunos desarrolladores, utilizar esta nomenclatura puede ser una tarea complicada, ya que involucra la repetición de código cuyo objetivo podría lograrse de una forma más simple.

Como ya se mencionó con anterioridad, Hibernate utiliza llamadas a métodos para la realización de consultas. En la gran mayoría de aplicaciones, se utilizan 4 métodos que coinciden con las operaciones CRUD de una base de datos: *Create* (crear), *Read* (leer), *Update* (actualizar) y *Delete* (eliminar). La codificación de estos métodos se vuelve muy repetitiva, lo cual abre la pauta para su automatización.

Grails surge en el año 2006 como una respuesta a la necesidad de agilizar, automatizar y simplificar el desarrollo de aplicaciones Web. Grails está basado en Spring, Hibernate y Groovy, y toma las mejores prácticas de cada uno de ellos para formar un marco de trabajo estable, robusto, sencillo de usar y de fácil mantenimiento.

El libro *Grails in Action* [GRA09] maneja 7 “Grandes Ideas” que

³ En versiones recientes de Spring y Hibernate, se promueve el uso de anotaciones en lugar de XML.

hacen único a Grails sobre otras herramientas. Las siguientes secciones explican estos puntos.

3.8.1 Convención sobre Configuración

Grails posee una estructura especial para la ubicación de cada uno de sus elementos, clases y archivos de configuración:

- Las clases de dominio se encuentran en una carpeta llamada *domain*.
- Los controladores están en la carpeta *controllers*.
- Las páginas web se localizan en la carpeta *views*.
- Los archivos de configuración se ubican en la carpeta *conf*.

El nombre de los controladores, vistas y URL's generadas por Grails dependen del nombre de la clase de dominio correspondiente. Por ejemplo, si se tiene una clase llamada *Profesor*, el controlador se nombra *ProfesorController* y las vistas se ubican en *views/profesor*, Asimismo, las URL's para acceder a las vistas se basan en el nombre de éstas. Ejemplo de ello es la vista *list*, cuya vista se encuentra en *views/profesor/list.gsp* y su correspondiente URL sería *http://nombre_aplicacion/profesor/list*.

Como puede observarse se siguen ciertas convenciones sobre las clases de dominio, controladores, vistas y URL's. Dichas convenciones ahorran al desarrollador la tarea de configurar la correspondencia entre estos módulos mediante XML. A esta característica de Grails se le conoce como *Convención sobre Configuración*.

3.8.2 Filosofía Ágil

La principal razón por lo que Grails vio la luz fue la necesidad de desarrollar aplicaciones rápidamente. A través de la Convención sobre Configuración, se acelera el proceso inicial de codificación. Por otra parte, Grails permite hacer cambios y visualizarlos en tiempo real sin necesidad de reiniciar el servidor de aplicaciones (acción muy común con otras herramientas). Una aplicación Grails puede generarse y ejecutarse con solo 2 comandos, y en cuestión de minutos, se pueden desarrollar funcionalidades sorprendentes.

3.8.3 Fundamentos sólidos

Grails está implementado sobre las mejores herramientas de desarrollo existentes para lograr sus objetivos. Como ya se ha mencionado con anterioridad, Grails está fuertemente basado en Spring, Hibernate y Groovy, tecnologías ampliamente aceptadas y reconocidas por su estabilidad y madurez. Por lo tanto, Grails tiene un soporte sólido que lo respalda como una excelente herramienta de desarrollo web.

3.8.4 Plantillas y “Scaffolding”

Para la generación automática de código, Grails maneja el concepto de “scaffolding”. Basado en la estructura de las clases de dominio, ésta técnica genera de forma dinámica el código necesario para los controladores y vistas sin necesidad de escribir código para ello y sin necesidad de la existencia de sus archivos correspondientes. Grails utiliza una serie de plantillas para la generación de éste código,

creando controladores y vistas consistentes y en tiempo real.

Naturalmente, no todas las aplicaciones web requieren la funcionalidad que las plantillas ofrecen. Grails ofrece la opción de modificarlas de acuerdo a las necesidades del desarrollador, permitiéndole agregar mayor funcionalidad. Una vez que las plantillas son modificadas, los artefactos correspondientes (controladores y vistas) pueden ser generados con las necesidades específicas de la aplicación ya incluidas. Esto tiene aplicación en páginas web que necesitan consistencia y uniformidad tanto en sus funciones como en su apariencia.

3.8.5 Integración con Java

Probablemente se tengan módulos desarrollados en Java y exista la necesidad de integrarlos con Grails. Groovy es completamente compatible con Java y viceversa, lo cual permite utilizar cualquier clase, API y biblioteca de clases de Java en Grails⁴. Todo el conocimiento adquirido con Java puede ser aplicado con Groovy y Grails. Asimismo, Grails permite crear clases en Java desde cero como si fuera una aplicación 100% hecha en dicho lenguaje. Como se dijo antes, Groovy no pretende ser la competencia ni mucho menos el sucesor de Java.

3.8.6 Wetware

A lo largo de casi 5 años, Grails ha ganado muchos seguidores, lo

⁴ Para que las clases Groovy puedan ser utilizadas en una clase Java, se requiere agregar un archivo JAR en el CLASSPATH de la aplicación. Dicho JAR contiene la infraestructura básica de Groovy.

que ha incrementado el número de recursos dedicados al soporte y mantenimiento de ésta herramienta. Sitios Web, foros de discusión, libros, podcasts y sobre todo el desarrollo exponencial de plugins son algunos de las herramientas disponibles para novatos y expertos en el desarrollo de aplicaciones web con Grails. El recurso más importante es el sitio web oficial [GRA01], donde el lector puede encontrar el software, la documentación, foros, enlaces a otros sitios, plugins, entre otras cosas.

3.8.7 Productividad

Sin duda, una de las consecuencias más evidentes del uso de Grails es el nivel de productividad que se obtiene. Una aplicación que podría llevar semanas o incluso meses queda lista en cuestión de horas o días con Grails. Esto permite a los desarrolladores enfocarse en otros aspectos como son la lógica y reglas de negocio, estética, la presentación, documentación, capacitación, etc.

3.9 Resumen

Grails es una herramienta poderosa basada en el lenguaje Groovy que agiliza y simplifica el desarrollo de aplicaciones web en JEE. El hecho de que esté basada sobre frameworks maduros y estables como son Spring y Hibernate lo hace sumamente confiable y fácil de usar. Su dinamismo e intuición lo convierten en una excelente opción para la filosofía ágil de creación de programas.

Capítulo 4 Creación y arranque del sistema

En este capítulo se inicia la codificación del sistema CSI de PROMEP-UAEH. Gracias a la facilidad de implementación que ofrece Grails, es posible poner en marcha el sistema de forma inmediata. En capítulos posteriores, conforme se desarrollan los tópicos concernientes a Grails, cada uno es ejemplificado con un módulo del sistema.

4.1 Generación de la aplicación

Si aún no se tiene instalado Grails, se sugiere consultar el Anexo A: Instalación de Grails, el cual contiene las instrucciones necesarias para su instalación en diversos sistemas operativos.

Para generar una aplicación con Grails, se escribe el siguiente comando:

```
> grails create-app {nombre de la aplicación}
```

Para el caso del sistema CSI, el comando se ejecuta de la siguiente forma:

```
> grails create-app BDPromep
```

Esto va a generar un directorio llamado *BDPromep*, y dentro de éste se encuentran todas los archivos y directorios necesarios para la aplicación. La estructura de directorios obtenida es la siguiente:

```
BDPromep/  
|-- grails-app  
| |-- conf  
| |-- controllers  
| |-- domain  
| |-- i18n  
| |-- services  
| |-- taglib  
| |-- utils  
| `-- views  
|-- lib  
|-- scripts  
|-- src  
|-- test  
`-- web-app
```

- **grails-app/conf:** contiene los archivos de configuración de bases de datos, propiedades, URL's, acciones a ejecutar en el arranque de la aplicación, entre otros.
- **grails-app/controllers:** aquí se colocan los archivos fuente correspondientes a los controladores de la aplicación.
- **grails-app/domain:** en esta carpeta se encuentran las clases de dominio.
- **grails-app/i18n:** aquí están los archivos de mensajes de internacionalización de la aplicación que le permiten adaptar el idioma de acuerdo al lenguaje del navegador.
- **grails-app/services:** las clases que modelan los servicios

generados por Grails se colocan en este directorio.

- **grails-app/taglib:** las clases que modelan los taglibs para Groovy Server Pages (GSP) se alojan en esta carpeta.
- **grails-app/utils:** aquí se colocan las clases utilitarias específicas de Grails⁵.
- **grails-app/views:** los archivos para las páginas web GSP se encuentran aquí.
- **lib:** en esta carpeta se colocan los archivos JAR que la aplicación requiera y que Grails no incluya por defecto.
- **scripts:** aquí se colocan las clases creadas por el desarrollador que pueden ejecutarse como scripts.
- **src:** si es necesario generar código Java o Groovy además del generado por Grails, se debe colocar en éste directorio.
- **test:** las pruebas unitarias y de integración se ponen en esta carpeta.
- **web-app:** aquí se ubican todos los archivos de configuración, JavaScript, imágenes y recursos estáticos propios de la aplicación web.

Para la puesta en marcha de la aplicación, se escribe el siguiente comando:

⁵ En muchas aplicaciones de Java se manejan clases utilitarias, las cuales funcionan como clases auxiliares que proporcionan funcionalidad propia de la aplicación que se utiliza en varios módulos de la misma. El directorio `grails-app/utils` de Grails se utiliza para colocar códecs específicos de la infraestructura de Grails, no de la aplicación desarrollada en sí. Por esta razón, rara vez es utilizado.

```
> grails run-app
```

Si todo marcha bien, al final de su ejecución, Grails muestra la siguiente salida:

```
Server running. Browse to http://localhost:8080/BDPromep
```

Para visualizar la aplicación, se debe abrir un navegador web y dirigirse a la dirección <http://localhost:8080/BDPromep>. El navegador muestra la página de la Figura 6.

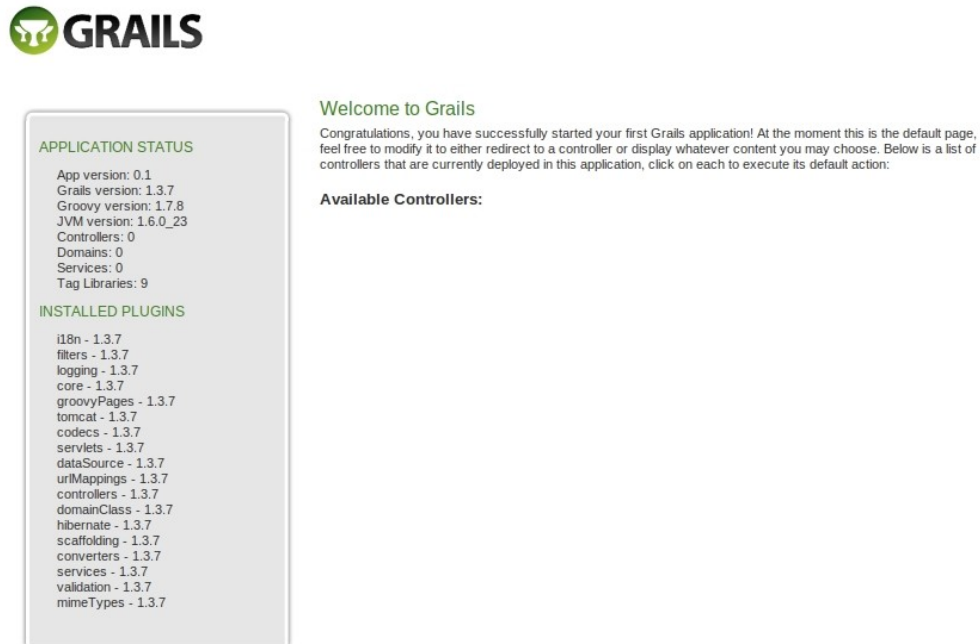


Figura 6: Página inicial de la aplicación.

El panel izquierdo de la página de la Figura 6 despliega información acerca de las versiones de la JVM (Java Virtual Machine), de la aplicación, de los diferentes plugins instalados, de Groovy y Grails, entre otros datos.

Grails ha generado y desplegado de forma automática toda la infraestructura necesaria para una aplicación web. No hay necesidad de instalar un servidor de aplicaciones, no hay necesidad de escribir archivos XML como ocurriría con cualquier otro framework de desarrollo web y no hay necesidad de escribir HTML para la página inicial. Grails automatiza todos esos pasos.

4.2 Lógica de negocios: creación de las clases de dominio

La generación de las clases de dominio de una aplicación web en Grails se encarga de codificar las entidades obtenidas en la fase de análisis y diseño. Dicha codificación se realiza mediante archivos *.groovy*. Para la creación de una clase de dominio, se utiliza el comando *grails create-domain-class*:

```
grails create-domain-class {paquete de la clase}{Nombre de la clase}
```

Para ejemplificar el uso de este comando, se genera la clase *Profesor* en el paquete *mx.edu.uaeh.promep*:

```
grails create-domain-class mx.edu.uaeh.promep.Profesor
```

Si no se escribe paquete alguno, Grails genera un paquete con el nombre de la aplicación web⁶, lo cual, en este caso, sería *bdpromep.Profesor*. Nótese la conversión a minúsculas del nombre de la aplicación. Esto con el fin de cumplir con las convenciones de nombramiento de paquetes en Java.

La ejecución del comando genera 2 archivos:

- **BDPromep/grails-**

⁶ En versiones anteriores de Grails no se generaba un paquete por defecto.

app/domain/mx/edu/uaeh/promep/Profesor.groovy, el cual corresponde a la clase de dominio.

- **BDPromep/grails-**

app/test/unit/mx/edu/uaeh/promep/ProfesorTests.groovy, en el cual se realizan las pruebas unitarias de la clase *Profesor*.

Al abrir el archivo *Profesor.groovy* en un editor de texto, se puede observar el siguiente contenido:

```
1. package mx.edu.uaeh.promep
2.
3. class Profesor {
4.
5.     static constraints = {
6.     }
7. }
```

Existe un paquete, el nombre de la clase y un *closure* donde se especifican las restricciones de los atributos de la clase. Las restricciones indican los límites de cada atributo, tales como su tamaño, su obligatoriedad, si deben seguir un patrón, entre otros. Este tema se estudia a detalle en el Capítulo 5.

Para modelar la clase de acuerdo al diagrama de clases del Anexo B, se modifica la clase como se muestra a continuación:

```
1. package mx.edu.uaeh.promep
2.
3. class Profesor {
4.
5.     String folio
6.     String paterno
```

```
7.     String materno
8.     String nombre
9.     String curp
10.    Character sexo
11.    String email
12.    String telefonoTrabajo
13.    String telefonoCasa
14.    Boolean plantilla
15.    Integer anioPlantilla
16.
17.    static constraints = {
18.        folio nullable: true, maxSize: 15
19.        paterno nullable: true, maxSize: 60
20.        materno nullable: true, maxSize: 60
21.        nombre nullable: true, maxSize: 60
22.        curp nullable: true, maxSize: 20
23.        sexo nullable: true, maxSize: 1
24.        email nullable: true, maxSize: 100
25.        telefonoTrabajo nullable: true, maxSize: 50
26.        telefonoCasa nullable: true, maxSize: 50
27.        anioPlantilla nullable: true
28.    }
29. }
```

Se han agregado los atributos y sus restricciones. Gracias a la naturaleza dinámica de Groovy, no hay necesidad de colocar los clásicos métodos *setters* y *getters* para el acceso y manipulación de los atributos de la clase.

4.3 Interacción con el mundo exterior: controladores y vistas

En la práctica, Grails utiliza el patrón de diseño MVC (Modelo-

Vista-Controlador) [MVC01], el cual consiste en separar la lógica de negocios de las interfaces gráficas de usuario y lograr una comunicación (no una mezcla) entre ellas por medio de controladores. La clase de dominio creada en la sección 4.2 corresponde a la lógica de negocios. Para la creación de controladores y vistas en Grails se utilizan los siguientes comandos:

- **create-controller:** Genera una clase controladora sin código. Se utiliza para la creación de controladores cuyo código es escrito de forma manual.
- **generate-controller:** Genera una clase controladora con todo el código necesario para administrar las peticiones hechas por el cliente web. Esto se explica a detalle en el Capítulo 7.
- **generate-views:** Crea todos los archivos GSP necesarios para las vistas.
- **generate-all:** Genera tanto controladores como vistas, equivale a ejecutar los comandos *generate-controller* y *generate-views* en un solo paso.

Estos comandos reciben como argumento la clase de dominio a la cual se le generarán los artefactos. A manera de ejemplo, la creación del controlador para la clase *Profesor* se realiza de la siguiente manera:

```
grails create-controller mx.edu.uaeh.promep.Profesor
```

Puede observarse que se debe especificar la ruta completa de la clase, incluyendo los paquetes. El controlador generado es nombrado

ProfesorController, y su contenido es el siguiente:

```
1. package mx.edu.uaeh.promep
2.
3. class ProfesorController {
4.
5.     def index = { }
6. }
```

En la sección 3.8.4 se habló acerca de la técnica de *scaffolding*, la cual genera de forma dinámica el código de los controladores y las vistas sin necesidad de que exista. Para lograr dicha propiedad, se modifica el controlador de la siguiente manera:

```
1. package mx.edu.uaeh.promep
2.
3. class ProfesorController {
4.     static scaffold = true
5. }
```

En la ventana del navegador mostrada en la Figura 6, se debe actualizar la pantalla (generalmente con la tecla F5) y se debe visualizar un vínculo al controlador justo debajo del título *Available Controllers*, tal como lo muestra la Figura 7.

Welcome to Grails

Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:

Available Controllers:

- [mx.edu.uaeh.promep.ProfesorController](#)

Figura 7: Página inicial de la aplicación con un vínculo al controlador.

Al hacer clic en el vínculo, la aplicación se dirige hacia la pantalla principal del catálogo de Profesores, como se muestra en la Figura 8.



Figura 8: Pantalla principal del catálogo de Profesores.

Al acceder al vínculo *New Profesor*, Grails muestra un formulario semejante al de la Figura 9.

The screenshot shows a form titled "Create Profesor" in a green font. The form contains several input fields for data entry: "Folio", "Paterno", "Materno", "Nombre", "Curp", "Sexo", "Email", "Telefono Trabajo", "Telefono Casa", and "Año Plantilla". Each of these fields is represented by a white rectangular box. Below the "Año Plantilla" field, there is a "Plantilla" label followed by a small square checkbox. At the bottom of the form, there is a "Create" button with a small icon to its left.

Figura 9: Página de creación de registros.

Al introducir datos en el formulario de registros, la aplicación dirige al usuario a la pantalla mostrada en la Figura 10.

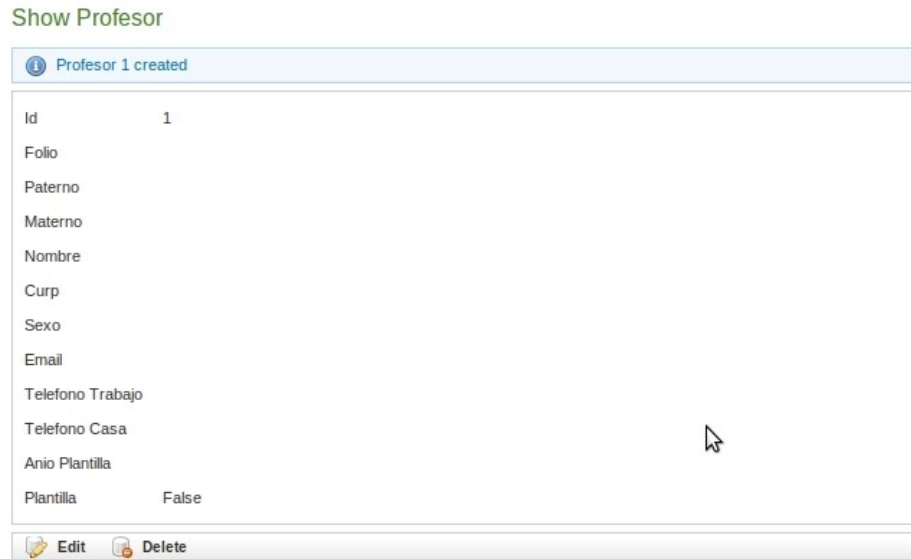


Figura 10: Confirmación de la creación del registro.

La aplicación muestra una notificación de que el registro fue insertado exitosamente. Asimismo, puede observarse que se despliegan cada uno de los campos insertados⁷. Grails genera un ID numérico auto-incremental para cada clase de dominio. Las opciones de edición y borrado del registro aparecen en la parte inferior.

Todas las páginas web y las acciones de creación, visualización, actualización y borrado de registros se generan con la creación del controlador y su variable estática *scaffold*. Como puede observarse, Grails automatiza el desarrollo de catálogos CRUD. Naturalmente, se pueden obtener los archivos correspondientes a los controladores y

⁷ En la imagen, se omiten los datos introducidos por confidencialidad de datos.

las vistas mediante los comandos *generate-controller*, *generate-views* y *generate-all*, los cuales son completamente editables de acuerdo a las necesidades de la aplicación. Para generar todo en un solo paso, se sugiere ejecutar la siguiente sentencia:

```
grails generate-all mx.edu.uaeh.promep.Profesor
```

Al momento de la ejecución, Grails encuentra que el controlador para la clase *Profesor* ya existe y pregunta al usuario si se debe sobrescribir dicho archivo:

```
File ../ProfesorController.groovy already exists. Overwrite? [y,n,a]
Tests ProfesorControllerTests.groovy already exists. Overwrite? [y/n] (y, Y,
n, N)
```

Cabe mencionar que existe un tiempo límite para escribir la respuesta a dicho comando. La estructura de los archivos generados (sin incluir los archivos de pruebas) es la siguiente:

```
BDPromep/
|-- grails-app
|-- controllers
|   |-- mx/edu/uaeh/promep
|       |-- ProfesorController.groovy
|-- views
    |-- profesor
        |-- create.gsp
        |-- edit.gsp
        |-- list.gsp
        |-- show.gsp
```

El principio *Convención sobre Configuración* mencionado en la sección 3.8.1 se hace presente. Los nombres de los directorios y

archivos de las vistas corresponden con las URL's enviadas al servidor. Asimismo, como se describe en el Capítulo 7, los nombres también corresponden con las acciones del controlador.

4.4 Persistencia: almacenamiento de datos en Grails

Grails permite la creación de catálogos CRUD de forma rápida. Por defecto, Grails utiliza la tecnología HSQLDB [HSQ01] para generar una base de datos *in memory* (en memoria), lo que significa que mientras el servidor de aplicaciones no se apague o reinicie, los datos almacenados en memoria se conservan. El uso de esta tecnología permite al desarrollador hacer pruebas de su aplicación sin necesidad de realizar la conexión con una base de datos real.

Grails utiliza 3 entornos: desarrollo (*development*), pruebas (*test*) y producción (*production*). La ejecución, empaquetado de WAR (el archivo entregable), conexión a base de datos, entre otras acciones, pueden ser configuradas para cada uno de estos entornos. Por defecto, Grails tiene establecido el entorno de desarrollo.

Echando un vistazo al archivo de configuración *BDPromep/grails-app/conf/DataSource.groovy*, se tiene lo siguiente:

```
1. dataSource {
2.     pooled = true
3.     driverClassName = "org.hsqldb.jdbcDriver"
4.     username = "sa"
5.     password = ""
6. }
7. hibernate {
8.     cache.use_second_level_cache = true
```

```
9.     cache.use_query_cache = true
10.    cache.provider_class = 'net.sf.ehcache.hibernate.EhCacheProvider'
11. }
12. // environment specific settings
13. environments {
14.     development {
15.         dataSource {
16.             dbCreate = "create-drop" // one of...
17.             url = "jdbc:hsqldb:mem:devDB"
18.         }
19.     }
20.     test {
21.         dataSource {
22.             dbCreate = "update"
23.             url = "jdbc:hsqldb:mem:testDb"
24.         }
25.     }
26.     production {
27.         dataSource {
28.             dbCreate = "update"
29.             url = "jdbc:hsqldb:file:prodDb;shutdown=true"
30.         }
31.     }
32. }
```

En JDBC, se requieren 4 parámetros básicos para la conexión a una base de datos:

- **Driver:** es el nombre de una clase controladora que permite la conexión a un gestor de bases de datos. Cada fabricante provee un JAR que debe ser agregado al *CLASSPATH* de una aplicación Java y que contiene dicha clase.
- **Url:** contiene la ubicación de la base de datos.

- **Username:** se coloca el nombre de usuario.
- **Password:** se coloca la contraseña del usuario.

El primer DSL (*Domain Specific Language*) del archivo *DataSource.groovy* nombrado *dataSource*, contiene 3 de los 4 parámetros principales de conexión a una base de datos: *driver*, *username* y *password*. El segundo DSL llamado *hibernate* contiene las propiedades que de forma manual se tendrían que configurar en el archivo *hibernate.cfg.xml*⁸. El tercer DSL contiene los tres entornos de Grails antes mencionados. En cada uno de ellos se puede configurar un DSL de tipo *dataSource*. Por defecto, en cada uno de ellos se incluyen 2 propiedades:

- **dbCreate:** se refiere al tratamiento de la base de datos al momento de iniciar y apagar el servidor de aplicaciones. Esta propiedad puede tener 3 valores: *create*, que genera la base de datos desde cero (si ya existe, la destruye); *create-drop*, que realiza lo mismo que *create* pero al apagar el servidor destruye la base de datos, y *update*, que únicamente realiza cambios sin alterar la base de datos existente (si esta no existe, se genera desde cero). Por ejemplo, si se agrega un atributo a una clase, se agrega un campo a la tabla correspondiente.
- **Url:** como ya se había mencionado anteriormente, contiene la ubicación de la base de datos.

⁸ Se sugiere consultar la documentación oficial de Hibernate para un tratamiento más profundo de todas las propiedades disponibles. Dicha documentación se encuentra en [HDC01]

Como un ejemplo de la portabilidad lograda por Hibernate y Grails hacia diferentes gestores de bases de datos, se va a cambiar el almacenamiento de datos de HSQLDB a MySQL. Para ello, se debe instalar el JAR que contiene el controlador de MySQL. Esto se logra modificando el archivo *grails-app/conf/BuildConfig.groovy*. El archivo original contiene lo siguiente:

```
1. grails.project.class.dir = "target/classes"
2. grails.project.test.class.dir = "target/test-classes"
3. grails.project.test.reports.dir = "target/test-reports"
4. //grails.project.war.file = "target/${appName}-${appVersion}.war"
5. grails.project.dependency.resolution = {
6.     // inherit Grails' default dependencies
7.     inherits("global") {
8.         // uncomment to disable ehcache
9.         // excludes 'ehcache'
10.    }
11.    log "warn" // log level of Ivy resolver, either 'error',...
12.    repositories {
13.        grailsPlugins()
14.        grailsHome()
15.        grailsCentral()
16.
17.        // uncomment the below to enable remote...
18.        // from public Maven repositories
19.        //mavenLocal()
20.        //mavenCentral()
21.        //mavenRepo "http://snapshots.repository.codehaus.org"
22.        //mavenRepo "http://repository.codehaus.org"
23.        //mavenRepo "http://download.java.net/maven/2/"
24.        //mavenRepo "http://repository.jboss.com/maven2/"
25.    }
```

```

26.     dependencies {
27.         // specify dependencies here under either 'build',...
28.         // runtime 'mysql:mysql-connector-java:5.1.13'
29.     }
30. }

```

Básicamente, este archivo se encarga de gestionar las dependencias de terceros que pueden ser agregadas por el desarrollador de acuerdo a sus necesidades. Se puede observar que el último DSL llamado *dependencies* tiene una dependencia en tiempo de ejecución correspondiente al JAR de MySQL. Se asume que el lector tiene instalado dicho gestor de bases de datos y que tiene conocimientos básicos de su uso. Para obtener dicho JAR, se sugiere detener la aplicación escribiendo *Ctrl+c* en la ventana de comandos donde se está ejecutando el servidor Tomcat. La consola muestra el siguiente mensaje:

```

Application context shutting down...
Application context shutdown.

```

Posteriormente, se modifica el archivo *BuildConfig.groovy* para habilitar los repositorios de Maven y descargar el JAR de MySQL (líneas 19 a 24 y 28):

```

19.     mavenLocal()
20.     mavenCentral()
21.     mavenRepo "http://snapshots.repository.codehaus.org"
22.     mavenRepo "http://repository.codehaus.org"
23.     mavenRepo "http://download.java.net/maven/2/"
24.     mavenRepo "http://repository.jboss.com/maven2/"
...
28.     runtime 'mysql:mysql-connector-java:5.1.13'

```

Como segundo paso, se debe modificar el archivo

DataSource.groovy para colocar los 4 parámetros necesarios para realizar la conexión:

```
1. dataSource {
2.     pooled = true
3.     driverClassName = "com.mysql.jdbc.Driver"
4.     username = "usuario"
5.     password = "contrasenia"
6. }
...
12. // environment specific settings
13. environments {
14.     development {
15.         dataSource {
16.             dbCreate = "update" // one of...
17.             url = "jdbc:mysql://localhost:3306/promep"
18.         }
19.     }
20.     test {
21.         dataSource {
22.             dbCreate = "update"
23.             url = "jdbc:mysql://localhost:3306/promep"
24.         }
25.     }
26.     production {
27.         dataSource {
28.             dbCreate = "update"
29.             url = "jdbc:mysql://localhost:3306/promep"
30.         }
31.     }
32. }
```

Finalmente, se debe crear una base de datos en MySQL llamada *promep* y se inicia la aplicación con el comando *grails run-app*. La

consola debe mostrar la descarga del JAR de MySQL y posteriormente el arranque de la aplicación. Si todo marcha bien, al consultar las tablas en la base de datos *promep* se obtiene un resultado similar al siguiente:

```
mysql> show tables;
+-----+
| Tables_in_promep |
+-----+
| profesor          |
+-----+
1 row in set (0.00 sec)

mysql> desc profesor;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id             | bigint(20)    | NO   | PRI | NULL    | auto_increment |
| version       | bigint(20)    | NO   |     | NULL    |                |
| anio_plantilla | int(11)       | YES  |     | NULL    |                |
| curp          | varchar(20)   | YES  |     | NULL    |                |
| email         | varchar(100)  | YES  |     | NULL    |                |
| folio         | varchar(15)   | YES  |     | NULL    |                |
| materno       | varchar(60)   | YES  |     | NULL    |                |
| nombre        | varchar(60)   | YES  |     | NULL    |                |
| paterno       | varchar(60)   | YES  |     | NULL    |                |
| plantilla     | bit(1)        | NO   |     | NULL    |                |
| sexo          | char(1)       | YES  |     | NULL    |                |
| telefono_casa | varchar(50)   | YES  |     | NULL    |                |
| telefono_trabajo | varchar(50)  | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
13 rows in set (0.00 sec)
```

Para cada clase de dominio, Hibernate y Grails han creado una

tabla. Cada atributo de las clases se corresponde con un campo de la tabla. Las restricciones establecidas en cada atributo se reflejan en el tamaño y obligatoriedad de cada campo. Gracias a la transparencia de Hibernate en el manejo de diferentes gestores de bases de datos, la aplicación generada con Grails no tiene conocimiento del gestor utilizado, lo que aumenta su portabilidad y facilidad de mantenimiento.

4.5 Resumen

A lo largo de éste capítulo se han demostrado las capacidades básicas de Grails. Con escribir unos cuantos comandos y algunas líneas de código, se ha implementado una aplicación web completamente funcional, de fácil mantenimiento y sencilla de usar. El desarrollo de aplicaciones web con Grails se convierte un una experiencia ágil e interesante. En la siguiente parte de este documento, Fundamentos Básicos de Grails, se describen con mayor detalle los tópicos tratados en éste capítulo, además de incluir temas que sientan las bases para el desarrollo de aplicaciones reales.

Parte 3 Fundamentos Básicos de Grails

Tal como se explicó en la sección 2.4, las aplicaciones generadas por Grails consisten de 3 capas: la capa de lógica de negocios, la capa de controladores y la capa de presentación. En la Parte 3 de este documento se detallan cada una de estas capas.

En el *Capítulo 5: Dominio de una aplicación web*, se muestran los fundamentos del modelado de entidades, tales como la selección adecuada del tipo de atributos, métodos *setter* y *getter*, validación de datos, personalización del mapeo para la base de datos y las relaciones existentes entre entidades (uno a uno, uno a muchos y muchos a muchos).

Una vez entendidos los fundamentos de las entidades, en el *Capítulo 6: Modelo Objeto-Relacional en Grails (GORM)*, se explica a detalle el tratamiento de Grails para la persistencia de objetos en una base de datos. Se muestran los métodos comunes a todos los objetos y se hace hincapié en la utilización de métodos dinámicos, Criterias y HQL para la realización de consultas.

Para la parte de interacción con el usuario final, en el *Capítulo 7: Capa de presentación Web: Controladores y Groovy Server Pages*, se presenta al lector la forma en que Grails implementa el patrón de diseño *Modelo-Vista-Controlador* (MVC) para lograr, a través de controladores, la interacción del dominio con la interfaz gráfica de usuario representada por las páginas web. Dichas páginas trabajan bajo la tecnología de Groovy Server Pages, la cual contiene

funcionalidades poderosas que van desde la emulación de instrucciones de condición, ciclos e impresión de variables con formato hasta la implementación de llamadas asíncronas Ajax con diversas bibliotecas con JavaScript.

Capítulo 5 Dominio de una aplicación web

Toda aplicación computacional posee una parte que modela aspectos del mundo real. Con este modelado se realizan diversas operaciones que dan vida a un programa. En una aplicación web, la *lógica de negocios* desempeña este papel. En Grails, las *clases de dominio* se encargan de una parte importante de la lógica de negocios abstrayendo las entidades que sirven como base para la creación de una base de datos. En este capítulo se tratan a detalle dichas clases.

5.1 Creación de una clase de dominio

Tal como se muestra en la sección 4.2, la generación de una clase de dominio en Grails se logra mediante el comando *create-domain-class*. La sintaxis de este comando es la siguiente:

```
> grails create-domain-class {paquete}{clase}
```

Ejemplos de su ejecución son los comandos escritos en el Capítulo 4:

```
> grails create-domain-class mx.edu.uaeh.promep.Profesor
```

Los archivos generados son 2: uno representa la clase en sí y se coloca en la carpeta *grails-app/domain*. El segundo archivo es una clase para la escritura de pruebas unitarias, ubicado en *test/unit*. El nombre utilizado para la clase de pruebas es *{paquete}{nombreClase}Tests.groovy*. Si no se especifica un paquete, Grails genera por omisión un paquete con el nombre del proyecto en minúsculas para ambos archivos (sección 4.2).

5.2 Adición de atributos

El contenido de la clase de dominio por defecto es similar al siguiente:

```
1. package {paquete}
2.
3. class {NombreClase} {
4.
5.     static constraints = {
6.     }
7. }
```

Una parte medular de las clases de dominio son los atributos. Los atributos de una clase representan aquellas características propias de la entidad a modelar. Para ejemplificar el uso de atributos de una clase, se retoma la clase *Profesor* del Capítulo 4:

```
1. package mx.edu.uaeh.promep
2.
3. class Profesor {
4.
5.     String folio
6.     String paterno
7.     String materno
8.     String nombre
9.     String curp
10.    Character sexo
11.    String email
12.    String telefonoTrabajo
13.    String telefonoCasa
14.    Boolean plantilla
15.    Integer anioPlantilla
16.
17.    static constraints = {
```

```
18.         folio nullable: true, maxSize: 15
19.         paterno nullable: true, maxSize: 60
20.         materno nullable: true, maxSize: 60
21.         nombre nullable: true, maxSize: 60
22.         curp nullable: true, maxSize: 20
23.         sexo nullable: true, maxSize: 1
24.         email nullable: true, maxSize: 100
25.         telefonoTrabajo nullable: true, maxSize: 50
26.         telefonoCasa nullable: true, maxSize: 50
27.         anioPlantilla nullable: true
28.     }
29. }
```

Las características de un profesor modeladas en esta clase son su nombre, apellidos, sexo, CURP, entre otros. Naturalmente, las profesores tienen más características que las mencionadas, pero para el modelado de la aplicación, éstos atributos satisfacen los requerimientos de la misma.

Cada uno de los atributos se caracteriza por un *tipo de dato*. Los tipos de datos indican la naturaleza del atributo. En el mundo real, existen datos numéricos, alfabéticos, lógicos, cronológicos, etc. A continuación, se hace un análisis detallado de los tipos de datos soportados por Grails.

5.2.1 Tipos de atributos

Al hacer el modelado de atributos de una clase, es preciso utilizar el tipo de datos que mejor se adapte a las necesidades de la aplicación. La Tabla 2 muestra los tipos de datos soportados por Grails, así como sus rangos.

Tipo de dato	Clase	Rango	Valor por defecto
Byte	Numérico entero	-128 a 127	0
Short	Numérico entero	-32,768 a 32,767	0
Integer	Numérico entero	-2,147,483,648 a 2,147,483,647	0
Long	Numérico entero	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0L
Float	Numérico flotante	Precisión simple de acuerdo al estándar IEEE 754 ⁹	0.0f
Double	Numérico flotante	Precisión doble de acuerdo al estándar IEEE 754	0.0d
Character	Alfanumérico	0 a 65535	'\u0000'
String	Alfanumérico	Tanto como la memoria de la JVM lo permita	null
Boolean	Lógico	True, false	false.
Date	Cronológico	A partir del 1 de enero de 1970 a las 00:00:00 GMT	null
Object	Definida por el usuario	Definido por el usuario	null

Tabla 2: Tipos de datos soportados por Grails.

Es importante conocer los rangos de los tipos de datos para lograr un mejor modelado y un mayor rendimiento de la aplicación. Por ejemplo, si se necesita representar la edad de una persona¹⁰, es impráctico utilizar un tipo de dato *Integer*. En aplicaciones donde se maneja una cantidad pequeña de datos, el uso de *Byte* e *Integer* de forma indistinta no se refleja en el rendimiento, pero en aplicaciones de gran escala donde se utilizan millones de datos, el almacenamiento de 200,000 registros de forma masiva llevaría 4 veces más tiempo con un dato *Integer* que con un dato de tipo *Byte*.

⁹ Para un mayor tratamiento acerca de la precisión de números flotantes, se sugiere visitar [IEEE01]

¹⁰ La edad es tomada únicamente como un ejemplo ilustrativo. Generalmente la edad de una persona nunca es almacenada sino calculada. Lo ideal es almacenar la fecha de nacimiento.

Al final de la tabla se define un tipo de dato *Object*, En Grails, este tipo de dato se utiliza para la creación de asociaciones entre objetos. Las clases *java.util.Date* y *java.lang.String* son consideradas objetos pero no están definidos por el usuario debido a que están incluidas en el API de Java. A estas clases se les da un tratamiento especial en Java, Hibernate y Grails por ser tipos de datos ampliamente utilizados. En el caso de la clase *Date*, Hibernate se encarga de hacer la equivalencia correspondiente sin necesidad de algún tipo de programación adicional, independientemente del tipo de gestor de bases de datos utilizado.

Se recomienda utilizar los *wrappers* (clases envolventes) de Java [JAV02] para datos primitivos en lugar de los datos primitivos puros, ya que en la práctica se presentan problemas con las bases de datos al utilizar referencias nulas que son difíciles de detectar.

5.3 Consistencia e integridad: validación de datos

En el archivo de la clase de dominio generado por Grails, se incluye un *closure* llamado *constraints*. Dentro de este *closure* se escriben las restricciones necesarias para los atributos. El uso de estas restricciones garantiza la integridad y consistencia de los datos ingresados a la aplicación.

Dentro de la aplicación, las restricciones se ven reflejadas en 2 partes:

- En la generación de la base de datos (sección 4.4),

- En la validación de campos al momento de crear un registro en la interfaz gráfica.

Para el caso de la clase *Profesor*, se tienen las siguientes restricciones:

```
...
17.     static constraints = {
18.         folio nullable: true, maxSize: 15
19.         paterno nullable: true, maxSize: 60
20.         materno nullable: true, maxSize: 60
21.         nombre nullable: true, maxSize: 60
22.         curp nullable: true, maxSize: 20
23.         sexo nullable: true, maxSize: 1
24.         email nullable: true, maxSize: 100
25.         telefonoTrabajo nullable: true, maxSize: 50
26.         telefonoCasa nullable: true, maxSize: 50
27.         anioPlantilla nullable: true
28.     }
...
```

La primera restricción a considerar es acerca del tratamiento de datos con valor nulo. En muchas bases de datos, la obligatoriedad de campos se maneja aplicando la restricción de *valor no nulo*. Esto significa que un registro no puede ser introducido si un campo marcado como no nulo tiene la condición contraria (es decir, es nulo). Grails permite la gestión de ésta restricción mediante la sentencia *nullable:{true|false}*.

Algunas cadenas de caracteres pueden ser marcadas como no nulas, pero esto permite la introducción de cadenas vacías. Para evitar esto, Grails utiliza la sentencia *blank:{true|false}*. Asimismo, la

longitud de las cadenas puede controlarse mediante la sentencia `size:{min}..{max}`.

La Tabla 3 y la Tabla 4 muestran todas las restricciones permitidas en Grails.

Nombre	Descripción	Ejemplo de uso
blank	Verifica que una cadena no esté vacía	nombre blank:false
creditCard	Verifica que una cadena sea un número de tarjeta de crédito válido.	numeroTarjeta creditCard:true
email	Verifica que una cadena sea una dirección de correo electrónico válida.	miMail email:true
inList	Verifica que el valor de un atributo esté dentro de una colección de valores permitidos.	colorPrimario inlist:["rojo", "amarillo", "azul"]
matches	Verifica que una cadena empate con una expresión regular dada.	apellido matches:"[a-zA-Z]+"
max	Verifica que un valor no supere el valor proporcionado.	precio max:1000
maxSize	Verifica que el tamaño de un valor no supere el valor proporcionado.	curp maxSize:18
min	Verifica que un valor no se encuentre por debajo del valor proporcionado.	precio min:10
minSize	Verifica que el tamaño de un valor no se encuentre por debajo del valor proporcionado.	curp minSize:18
notEqual	Verifica que un atributo no sea igual al valor proporcionado.	jedi notEqual:"Palpatine"
nullable	Habilita el uso de valores nulos. Por defecto, Grails no los permite.	password nullable:false
range	Acota el valor de una propiedad a los valores proporcionados	adulto range:18..60
scale	Establece el número de decimales para valores flotantes	moneda scale:2
size	Acota el tamaño de una propiedad entre los valores proporcionados	isbn size:13..13

Tabla 3: Restricciones utilizadas en Grails.

Nombre	Descripción	Ejemplo de uso
unique	Verifica que la propiedad no se repita en otros registros.	username unique:true
url	Verifica que una cadena sea una URL válida.	direccionWeb url:true
validator	Permite la creación de validaciones personalizadas.	numeroPar validator: { return (it % 2) == 0 }

Tabla 4: Restricciones utilizadas en Grails (continuación).

La sintaxis de validación permite colocar más de una sentencia por atributo, las cuales van separadas por comas. Asimismo, las restricciones pueden encerrarse entre paréntesis para una mejor legibilidad. Por ejemplo:

```
atributo( size:1..50, nullable:false, blank:false )
```

La documentación oficial de Grails¹¹ utiliza paréntesis para ilustrar los ejemplos de cada una de las validaciones.

5.4 Estilizando la base de datos

Tal y como se estudió en la sección 4.4, al momento de ejecutar la aplicación, Grails genera o actualiza la base de datos en base a las clases de dominio existentes. Para determinar el nombre de las tablas y campos, Grails utiliza las siguientes convenciones:

- Cada clase de dominio es representada por una tabla.
- Cada atributo de la clase de dominio es representado por un campo.

¹¹ Es altamente recomendable que el usuario se familiarice con la documentación de Grails y la utilice como referencia principal. La versión en línea más reciente se encuentra en [GRA02]. Cada distribución de Grails contiene la documentación en la carpeta *doc/index.html*.

- Para todos los nombres, se utilizan únicamente letras minúsculas.
- En caso de que el nombre de una clase o atributo consista de 2 o más palabras, cada una de ellas es separada por un guión bajo.
- Por cada tabla se generan los campos *id* y *version* de forma automática. El campo *id* funciona como llave primaria, mientras que *version* lleva la cuenta de las actualizaciones hechas al registro.

La Tabla 5 muestra algunos ejemplos de las convenciones antes mencionadas.

Tipo	Nombre en Grails	Nombre en la base de datos
Clase (Tabla)	Persona	persona
Clase (Tabla)	AccionRealizada	accion_realizada
Atributo (Campo)	nombre	nombre
Atributo (Campo)	apellidoPaterno	apellido_paterno

Tabla 5: Algunas convenciones utilizadas en la generación de la base de datos.

Es muy frecuente que las convenciones proporcionadas por Grails no satisfagan las necesidades de la aplicación. En ocasiones, se debe desarrollar una aplicación cuya base de datos ya existe, y generalmente ésta no sigue las convenciones antes mencionadas.

Grails proporciona la opción de estilizar la base de datos a través de un *closure* denominado *mapping*. Este *closure* utiliza gran parte de las convenciones utilizadas por Hibernate para realizar el mapeo ORM

entre clases y tablas.

La Tabla 6 muestra algunas de las opciones de estilización de bases de datos.

Nombre	Descripción	Ejemplo de uso
id	Personaliza la forma en que la llave primaria es generada.	id name:'titulo'
table	Personaliza el nombre de la tabla correspondiente a la clase.	table name:'catalogo_libros', schema:'dbo'
column	Personaliza la definición de una columna.	moneda column: "currency", sqlType: "char", length: 3
order	Personaliza el orden por defecto para el atributo.	order 'desc'
sort	Establece el atributo por defecto a tomar en cuenta para la ordenación de los resultados de las consultas.	sort 'fechaNacimiento'
version	Personaliza el atributo version de la tabla correspondiente.	version false

Tabla 6: Algunas opciones de estilización de la base de datos en Grails.

En la sección 4.2, se modeló un atributo llamado *folio* para la clase *Profesor*. El atributo *folio* es único e irrepetible, por lo que es necesario establecerlo como la llave primaria de la tabla de profesores.

Para realizar un cambio en la base de datos, es recomendable detener la aplicación utilizando *Ctrl+C* como se describe en la sección 4.4. Hecho esto, se procede a editar el archivo ubicado en *grails-app/domain/mx/edu/uaeh/promep/Profesor.groovy* para agregar la estilización con el *closure mapping*:

```

1. package mx.edu.uaeh.promep
2.
3. class Profesor {

```

```

4.
5.     // Atributos...
6.     static constraints = {
7.         //Restricciones...
8.     }
9.
10.    static mapping = {
11.        id name: 'folio', generator:'assigned'
12.    }
13. }

```

Por defecto, el nombre de la llave primaria es *id* y el tipo de generación es *increment*. En el *closure* se sobrescribe el nombre del campo que corresponde al id a *folio* y se especifica su tipo de generación a *assigned* (línea 11), lo que significa que es responsabilidad del desarrollador la generación y asignación de la llave primaria.

Como siguiente paso, se debe borrar la tabla *profesor*. En MySQL, esto se logra ejecutando el siguiente comando:

```
mysql> drop table profesor;
```

Ahora, se debe poner en marcha la aplicación con `grails run-app`. Grails y Hibernate generan nuevamente la tabla *profesor*. Cuando se revisa en MySQL, se obtiene la siguiente estructura que ya contiene el folio como llave primaria:

```
mysql> desc profesor;
```

Field	Type	Null	Key	Default	Extra
folio	varchar(15)	NO	PRI	NULL	
version	bigint(20)	NO		NULL	

anio_plantilla	int(11)	YES		NULL		
curp	varchar(20)	YES		NULL		
email	varchar(100)	YES		NULL		
materno	varchar(60)	YES		NULL		
nombre	varchar(60)	YES		NULL		
paterno	varchar(60)	YES		NULL		
plantilla	bit(1)	NO		NULL		
sexo	char(1)	YES		NULL		
telefono_casa	varchar(50)	YES		NULL		
telefono_trabajo	varchar(50)	YES		NULL		

-----+-----+-----+-----+-----+-----+
 12 rows in set (0.00 sec)

5.5 Relaciones entre clases de dominio

En la gran mayoría de aplicaciones web con acceso a bases de datos, las entidades están relacionadas entre sí. Grails soporta los 3 tipos de relaciones existentes en el modelo relacional de base de datos: *Uno a uno*, *uno a muchos* y *muchos a muchos*. Las siguientes secciones adaptadas de la documentación oficial de Grails [GRA02] tratan con detalle cada una de ellas.

5.5.1 Relación uno a uno

Una relación *uno a uno* en Grails se logra mediante la siguiente nomenclatura:

```

1. class Profesor {
2.     static hasOne = [direccion:Direccion]
3. }
4. class direccion {
5.     Profesor profesor
6. }
```

Una profesor tiene una y sólo una dirección. La propiedad *hasOne* establece la relación *uno a uno* entre la clase *Profesor* y la clase *Direccion*. En la base de datos, esto se reflejará con una llave foránea en la tabla *direccion* en una columna llamada *profesor_id*.

Para fortalecer la relación *uno a uno*, es conveniente agregar una restricción de tipo *unique* (línea 4):

```
1. class Profesor {
2.     static hasOne = [direccion:Direccion]
3.     static constraints = {
4.         direccion unique: true
5.     }
6. }
7. class Direccion {
8.     Profesor profesor
9. }
```

5.5.2 Relación muchos a uno

Una relación muchos a uno es la más sencilla de implementar, y es definida agregando una propiedad que sea del tipo de otra clase. Por ejemplo:

```
1. class Profesor {
2.     Grupo grupo
3. }
4. class Grupo {
5. }
```

Con esto se logra una relación *muchos a uno unidireccional*, lo que significa que la clase *Profesor* puede acceder a su atributo *grupo* pero *Grupo* no puede acceder a la clase *Profesor* que la contiene.

Para lograr que la relación sea bidireccional, basta con agregar el atributo *belongsTo* (línea 5):

```
1. class Profesor {
2.     Grupo grupo
3. }
4. class Grupo {
5.     static belongsTo = [profesor:Profesor]
6. }
```

Con ello se establece que la clase *Grupo* pertenece a la clase *Profesor*. El resultado de esto es que se puede asociar una instancia de *Grupo* a una instancia de la clase *Profesor*, y cuando se ejecuten acciones de guardar y borrar en ésta última, éstas se ven reflejadas en *Grupo*. En otras palabras, las acciones de guardar y borrar se hacen en cascada (de *Profesor* a *Grupo*):

```
1. new Profesor(grupo:new Grupo()).save()
2. -----
3. def p = Profesor.get(1)
4. p.delete() // Profesor y Grupo son borradas
```

El ejemplo anterior guarda y borra tanto *Profesor* como *Grupo*. Cabe destacar que aunque se establece una relación bidireccional, la acción inversa no es permitida:

```
new Grupo(profesor:new Profesor()).save() // error
```

5.5.3 Relación uno a muchos

Una relación *uno a muchos* se establece cuando una clase, por ejemplo *Profesor*, tiene muchas instancias de otra clase, por ejemplo *Plazas*. En Grails, dicha relación se establece con la sentencia *hasMany* (línea 2):

```
1. class Profesor {
2.     static hasMany = [plazas:Plaza]
3.     String nombre
4. }
5. class Plaza {
6.     String nombre
7. }
```

Lo que se logra es una relación *uno a muchos unidireccional*, que significa que la clase *Profesor* puede acceder a su colección de *plazas* pero *Plazas* no puede acceder a la clase *Profesor* que la contiene.

De forma automática, Grails genera una propiedad de tipo *java.util.Set* dentro de la clase *Profesor* cuyo nombre es *plazas*. Como ejemplo de uso se tiene la siguiente pieza de código:

```
1. def p = Profesor.get(1)
2. p.plazas.each {
3.     println it.nombre
4. }
```

El comportamiento en cascada antes mencionado se hace presente, con excepción de la acción de eliminar, la cual no se realiza a menos que una sentencia *belongsTo* sea especificada:

```
1. class Profesor {
2.     static hasMany = [plazas:Plaza]
3.     String nombre
4. }
5. class Plaza {
6.     static belongsTo = [profesor:Profesor]
7.     String nombre
8. }
```

belongsTo también establece la relación bidireccional entre *Profesor*

y *Plaza*.

5.5.4 Relación muchos a muchos

El tratamiento de Grails para las relaciones muchos a muchos establece que en ambas clases debe existir una sentencia *hasMany* y que una de las clases debe ser la *propietaria*:

```
1. class Profesor {
2.     static belongsTo = Instituto
3.     static hasMany = [institutos:Instituto]
4.     String nombre
5. }
6. class Instituto {
7.     static hasMany = [profesores:Profesor]
8.     String nombre
9. }
```

La sentencia *belongsTo* establece que la clase *Instituto* es la clase propietaria y que se hace responsable de la persistencia de la clase *Profesor*. Esta última no puede gestionar la persistencia de *Instituto*. La siguiente sección de código guarda ambas clases, ya que es la clase *Instituto* quien realiza la acción *save*:

```
1. new Instituto(nombre:"ICBI")
2.     .addToProfesores(new Profesor(nombre:"Alejandro"))
3.     .addToProfesores(new Profesor(nombre:"Eduardo"))
4.     .save()
```

En la siguiente sección de código, únicamente la clase *Profesor* es guardada:

```
1. new Profesor(nombre:"Alejandro")
2.     .addToInstitutos(new Instituto(nombre:"ICBI"))
```

```
2.     .addToInstitutos(new Instituto(nombre:"ICSA"))
4.     .save()
```

Hasta el momento, la característica de *scaffolding* en Grails no soporta las relaciones *muchos a muchos*. El desarrollador debe hacerse cargo de esto de forma manual.

5.6 Resumen

En este capítulo se trataron a detalle las características que tienen las clases de dominio en Grails. La selección adecuada del tipo de dato para cada atributo, la validación de los mismos, la consistencia y estilización de la base de datos, así como el tratamiento de los diferentes tipos de relaciones entre clases son tópicos cruciales para el diseño óptimo de la lógica de negocios.

En el siguiente capítulo se explora la persistencia de datos con Grails y ORM (GORM).

Capítulo 6 Modelo Objeto-Relacional en Grails (GORM)

Hoy en día, son contados los sitios web que no cuentan con una base de datos para el almacenamiento de la información que manejan. La persistencia de datos es una de las características principales de los programas de computadora, ya que permite el uso y manipulación de éstos a corto y largo plazo. Las aplicaciones web desarrolladas con Grails no son la excepción en el uso de esta tecnología. Por ello, en las siguientes secciones se examina a profundidad el almacenamiento de datos con GORM.

6.1 Hibernate Framework y ORM

Tal como se mencionó en la sección 3.6, Hibernate es un framework diseñado para facilitar el uso de bases de datos con Java. El objetivo de Hibernate es liberar al desarrollador de la escritura del 95% de sentencias SQL manejando el paradigma de la programación orientada a objetos. Esto se logra mediante la tecnología ORM (Object-Relational Mapping, Mapeo Objeto-Relacional) y la abstracción del gestor de base de datos de la aplicación.

ORM utiliza mapeos de las clases de dominio de una aplicación haciéndolos corresponder con una base de datos de tipo relacional. Con ello, cada clase es representada por una tabla, cada atributo es representado por un campo y cada instancia se representa con un registro.

De forma manual, el desarrollador tiene que escribir los archivos de mapeo (cuya terminación es *.hbm.xml) o las anotaciones

correspondientes en las clases de dominio para hacer la correspondencia Objeto-Relacional. Grails hace el mapeo de forma automática utilizando convenciones (sección 5.4, Tabla 5) y permite la estilización de los atributos de cada tabla (sección 5.4, Tabla 6). En caso de que ya se tengan los archivos de mapeo, Grails permite su reutilización.

A manera de ejemplo, si se tuviera que hacer manualmente el archivo de mapeo de una clase hipotética llamada *Persona*, se obtendría lo siguiente:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3. 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4. <hibernate-mapping>
5.   <class name="clasehipotetica.Persona" table="PERSONA">
6.     <id column="id" name="id" type="long">
7.       <generator class="native"/>
8.     </id>
9.     <property column="nombre" name="nombre" length="100"
10.       not-null="true" type="string"/>
11.    <property column="apellido_paterno" name="apellidoPaterno"
12.       length="100" not-null="true" type="string"/>
13.    <property column="apellido_materno" name="apellidoMaterno"
14.       length="100" not-null="true" type="string"/>
15.    <property column="fecha_nacimiento" name="fechaNacimiento"
16.       not-null="true" type="timestamp"/>
17.    <property column="domicilio" name="domicilio" length="100"
18.       not-null="true" type="string"/>
19.   </class>
20. </hibernate-mapping>
```

Puede notarse que la nomenclatura utilizada es XML, la cual en

ocasiones resulta incómoda para la lectura y mantenimiento de aplicaciones. Mediante la característica *Convención sobre Configuración* (sección 3.8.1), Grails elimina la necesidad de escribir el archivo de mapeo.

Para lograr la abstracción e independencia de la aplicación del gestor de base de datos utilizado, Hibernate proporciona un dialecto para cada tipo de gestor. Al momento de hacer la conexión, se identifica y aplica el lenguaje SQL propio de dicho gestor, liberando al desarrollador de preocuparse por la portabilidad. Hibernate soporta alrededor de 40 tipos diferentes de gestores de bases de datos.

6.2 ORM en Grails

Como se mencionó en la sección 3.8.3, Grails basa toda su persistencia de datos en Hibernate. Esto quiere decir que todo el conocimiento sobre este framework adquirido con anterioridad es aplicado en Grails, a la vez que éste último agrega funcionalidades y facilita el uso de ORM incluso si el desarrollador no tiene experiencia alguna en el uso de Hibernate.

Al crearse una clase de dominio con Grails, se le agregan métodos de persistencia de forma dinámica e implícita. La funcionalidad de los métodos van desde las 4 operaciones básicas *CRUD* (*create*, *read*, *update* y *delete*), consultas con métodos dinámicos, consultas estáticas mediante *Criteria*, hasta consultas con *HQL* (Hibernate Query Language). El resto del capítulo detalla dichas tecnologías.

6.3 Métodos comunes a todas las clases de dominio

En toda aplicación web con conexión a base de datos, las operaciones *CRUD* son las primeras en codificarse y las más utilizadas. El uso en Grails de éstas operaciones es bastante sencillo.

6.3.1 Create

Para generar un registro en la base de datos, basta con tener la instancia de una clase de dominio y llamar al método *save*:

```
1. Profesor profesor = new Profesor(
2.     nombre: 'Alejandro',
3.     paterno: 'García',
4.     materno: 'Granados',
5.     curp: 'GAGA870301HDFRRL09',
6.     sexo: 'M')
6. profesor.save()
```

El método *save* puede recibir parámetros. La Tabla 7 describe cada uno de ellos.

Nombre	Descripción	Ejemplo de uso
validate	Bandera que determina si se lleva a cabo la validación de restricciones de la instancia.	profesor.save(validate:true)
flush	Bandera que determina que si el objeto se guarda inmediatamente o no.	profesor.save(flush:true)
failOnError	Bandera que determina si se arroja una excepción si la validación de la instancia falla.	profesor.save(failOnError:true)
deepValidate	Bandera que determina si la validación es en cascada en las asociaciones de la instancia.	profesor.save(deepValidate:true)

Tabla 7: Parámetros del método *save*.

El método *save* no guarda una instancia inmediatamente a menos que la bandera *flush* esté activada. Por defecto, el método regresa la

instancia guardada. Pero si ocurre algún error, devuelve nulo.

6.3.2 Read

La obtención de registros de la base de datos con Grails se logra mediante varios métodos. Uno de ellos es el método *get*:

```
Profesor profesor = Profesor.get( 23 )
```

El método *get* recibe como parámetro el id del registro deseado. Si se desean obtener instancias de sólo lectura, se utiliza el método *read*:

```
Profesor profesor = Profesor.read( 23 )
```

Un método interesante que permite obtener varias instancias es *getAll*:

```
def listaProfesores = Profesor.getAll( 2, 1, 3 )
def listaProfesores = Profesor.getAll( [ 1, 2, 3 ] )
def listaProfesores = Profesor.getAll()
```

El método *getAll* puede utilizarse de 3 formas. La primera de ellas recibe los id's de las instancias deseadas separados por comas. La particularidad de éste método es que sus argumentos son de longitud variable, es decir, puede recibir cualquier número de id's. La segunda forma recibe una lista, y la tercera, al no recibir parámetros, devuelve todas las instancias de la base de datos.

Un método muy utilizado en Grails en la capa de controladores es *list*, el cual devuelve una lista de todos los objetos de la clase de dominio.

```
1. def listaProfesores = Profesor.list()
2. def listaProfesores = Profesor.list( params )
```

Existen 2 versiones de este método. La primera devuelve todas las

instancias existentes, mientras que la segunda recibe un mapa con diversos parámetros de paginación y ordenamiento:

- *max*, que limita el máximo de registros,
- *offset*, que determina el registro de partida para la consulta,
- *order*, que indica el orden de los registros (ascendente o descendente),
- *sort*, que indica el atributo tomado como referencia para el ordenamiento.

6.3.3 Update

El método *save* es utilizado en Grails para crear y actualizar instancias de clases de dominio. Por lo tanto, las reglas descritas en la sección 6.3.1 aplican para la actualización de registros.

6.3.4 Delete

Para eliminar una instancia en Grails, se utiliza el método *delete*:

```
1. def profesor = Profesor.get( 1 )
2. profesor.delete()
```

Para que el método funcione, la instancia debe haberse obtenido de la base de datos, es decir, tiene que ser persistente. En caso contrario se obtiene un error.

El método *delete* recibe un parámetro, *flush*, que funciona tal y como se describe en la Tabla 7, sólo que en vez de ser guardada o actualizada, la instancia se elimina inmediatamente.

6.4 Métodos dinámicos

Además de las operaciones CRUD, una aplicación con acceso a base de datos realiza consultas. Grails ofrece la utilización de las mismas mediante una tecnología denominada *Dynamic Finders* (buscadores dinámicos o métodos dinámicos). La característica principal de estos métodos es que el usuario no los implementa de forma directa, sino que con base a las características de la clase de dominio éstos pueden ser invocados. Para ilustrar su uso, se considera la estructura de la clase *Profesor*:

```
1. class Profesor {
2.
3.     String nombre
4.     String paterno
5.     String materno
6.     String curp
7. ...
```

Los siguientes métodos dinámicos pueden ser invocados:

```
1. def p = Profesor.findByNombre( "Anakin" )
2. p = Profesor.findByNombreAndPaterno( "James", "Gosling" )
3. p = Profesor.findAllByNombreLike( "%Roberto%" )
4. p = Profesor.findAllByNombreIlike( "%Roberto%" )
5. p = Profesor.findAllByNombreNotEqual( "Alicia" )
6. p = Profesor.countByCurpIsNull()
7. p = Profesor.countByPaternoOrMaterno(
8.     "Gómez", "Hernández" )
```

Grails utiliza un mecanismo interno que en tiempo de ejecución ejecuta la consulta formada por el método dinámico, lo que significa que ningún método es implementado en sí, sino que todos son ejecutados en una sola consulta al estilo de Hibernate. La sintaxis

general de un método dinámico es la siguiente:

```
[nombreClase].(findBy|findAllBy|countBy)([atr][comp][op])?[atr][comp]
```

Donde:

- **nombreClase:** la clase de dominio que llama al método dinámico
- **findBy|findAllBy|countBy:** el tipo de consulta a realizar.
- **atr:** el nombre del atributo de la clase de dominio
- **comp:** el tipo de comparador a utilizar.
- **op:** el tipo de operador a utilizar.

La Tabla 8 y la Tabla 9 muestran los tipos de comparadores y operadores disponibles en los métodos dinámicos.

Nombre	Tipo	Tipo de consulta soportado
LessThan	Comparador	countBy, findAllBy, findBy
LessThanEquals	Comparador	countBy, findAllBy, findBy
GreaterThan	Comparador	countBy, findAllBy, findBy
GreaterThanEquals	Comparador	countBy, findAllBy, findBy
Between	Comparador	countBy, findAllBy, findBy
Like	Comparador	countBy, findAllBy, findBy
ILike	Comparador	countBy, findAllBy, findBy
IsNull	Comparador	countBy, findAllBy, findBy
IsNotNull	Comparador	countBy, findAllBy, findBy
Not	Comparador	countBy, findAllBy, findBy
Equal	Comparador	countBy

Tabla 8: Comparadores y Operadores soportados por los métodos dinámicos.

Nombre	Tipo	Tipo de consulta soportado
NotEqual	Comparador	countBy, findAllBy, findBy
And	Operador	countBy, findAllBy, findBy
Or	Operador	countBy, findAllBy, findBy
InList	Comparador	findAllBy, findBy

Tabla 9: Comparadores y Operadores soportados por los métodos dinámicos (continuación).

La sentencia *Not* es utilizada para hacer consultas sobre datos de tipo booleano. Por ejemplo, si existiera un atributo *suscrito* de tipo booleano en la clase *Profesor*, un método dinámico quedaría de la siguiente manera:

```
1. p = Profesor.findSuscritoByNombre( "Roberto" )
2. p = Profesor.findNotSuscritoByNombre( "Roberto" )
```

Una limitación de los métodos dinámicos es que solo pueden utilizarse 2 atributos de la clase de dominio para construirse. Para la realización de consultas que involucren más de 2 atributos, se utiliza la tecnología de *Criteria* o *HQL*.

6.5 Realización de consultas orientadas a objetos: Criteria

Criteria es una tecnología de realización de consultas orientada a objetos, lo cual significa que se utilizan métodos en lugar de sentencias introducidas mediante cadenas. Un ejemplo tomado de la documentación oficial de Grails es la siguiente:

```
1. def c = Profesor.createCriteria()
2. def results = c {
3.     between("anioPlantilla", 500, 1000)
4.     eq("nombre", "Alejandro")
5.     or {
```

```
6.         like("paterno", "Fred%")
7.         like("paterno", "Barney%")
8.     }
9.     maxResults(10)
10.    order("materno", "desc")
11. }
```

Primero se debe crear un objeto de tipo *Criteria*, el cual se obtiene llamando al método *createCriteria* o *withCriteria* de la clase de dominio. Posteriormente las restricciones se escriben mediante un DSL. Las restricciones corresponden al API de Hibernate, específicamente de la clase *Restrictions* [HIB02]. En el ejemplo mostrado se buscan los registros que cumplan con las siguientes restricciones:

- Su atributo *anioPlantilla* este entre 500 y 1000.
- Su atributo *nombre* sea igual a “Alejandro”.
- Su atributo *parterno* comience con “Fred” o “Barney”.
- El número máximo de registros es 10.
- Los resultados se ordenan en forma descendente tomando el atributo *materno* como referencia.

Por defecto, todas las restricciones son utilizadas con el operador lógico AND. Si se desea utilizar otro tipo de operador lógico, se deben agrupar. El ejemplo anterior muestra la agrupación de 2 restricciones *like* mediante el operador lógico OR.

Puede utilizarse *Criteria* con asociaciones entre objetos. Suponiendo que se tiene la siguiente clase de dominio:

```

1. class Profesor {
2.     ...
3.     static hasMany = [asistencias:Asistencia]
4.     ...
5. }

```

Para utilizar el atributo *transactions* en el DSL de *Criteria*, se debe utilizar un nodo con el nombre del atributo:

```

1. def c = Profesor.createCriteria()
2. def now = new Date()
3. def results = c.list {
4.     asistencias {
5.         between('date',now-10, now)
6.     }
7. }

```

Este ejemplo busca todas las instancias de *Profesor* que tengan asistencias realizadas durante los últimos 10 días. Puede notarse el uso del método *list* en el objeto *c*. *Criteria* soporta 5 métodos diferentes, siendo *list* el método utilizado por defecto en caso de no especificar alguno. La Tabla 10 describe éstos métodos.

Método	Descripción
list	Regresa todos los resultados que coincidan con las restricciones. Método utilizado por defecto.
get	Regresa una lista con un solo resultado.
scroll	Regresa una lista desplazable.
count	Regresa el número de resultados.
listDistinct	Regresa un lista con resultados no repetidos.

Tabla 10: Métodos soportados por *Criteria*.

Para un tratamiento detallado del uso de *Criteria*, se sugiere consultar el API de Hibernate y la documentación oficial de Grails.

6.6 Hibernate Query Lenguaje (HQL)

Para la realización de consultas similares a SQL, Hibernate proporciona su propio lenguaje denominado *HQL* (Hibernate Query Language) el cual es muy parecido a SQL, con la diferencia de que HQL es completamente orientado a objetos. Una excelente introducción a HQL puede ser encontrada en su documentación oficial [HIB03].

Grails proporciona 3 métodos para la realización de consultas HQL. La Tabla 11 describe cada uno de ellos.

Método	Descripción
findAll	Regresa todos los resultados que coincidan con la consulta.
find	Regresa una lista con un solo resultado.
executeQuery	Permite la ejecución de una sentencia HQL que no necesariamente regresa instancias de una clase de dominio.

Tabla 11: Métodos en Grails para el uso de HQL.

Ejemplos de consultas HQL son los siguientes:

```
1. def resultados = Profesor.findAll(
2.     "from Profesor as p where p.nombre like 'Ale%'")
```

HQL con el uso de parámetros posicionales:

```
1. def resultados = Profesor.findAll(
2.     "from Profesor as p where p.nombre like ?", ["Ale%"])
3. def grupo = Grupo.findByNombre("9-4")
4. def profesores = Profesor.findAll(
5.     "from Profesor as p where p.grupo = ?", [grupo])
```

HQL con el uso de parámetros nombrados:

```
1. def resultados = Profesor.findAll(
2.     "from Profesor as p where p.nombre like :search or p.nombre " +
3.     "like :search", [search:"Ale%"])
```

```
4. def grupo = Grupo.findByNombre("9-4")
5. def profesores = Profesor.findAll(
6.     "from Profesor as p where p.grupo = :grupo", [grupo: grupo])
```

Una característica de HQL que es muy utilizada en Grails es la paginación y el ordenamiento de resultados. Los métodos *findAll* y *executeQuery* soportan un parámetro extra para introducir paginación mediante *max* y *offset*:

```
1. def resultados = Profesor.findAll(
2.     "from Profesor as p where p.nombre like 'Ale%' " +
3.     "order by p.nombre asc", [max:10, offset:20])
4. results = Profesor.executeQuery(
5.     "select distinct p.nombre from Profesor p where p.nombre = :nombre",
6.     [nombre:'Alejandro'], [max:10, offset:5] )
```

6.7 Resumen

A lo largo de este capítulo se mostraron las principales características de GORM. La persistencia de datos en una aplicación web es una de los módulos más utilizados, por lo que Grails le dedica una parte considerable de su entorno. En el siguiente capítulo se habla acerca de la capa de presentación y su interacción con la lógica de negocios, además de aplicar lo aprendido de las clases de dominio y GORM.

Capítulo 7 Capa de presentación Web: Controladores y Groovy Server Pages

Para lograr la interacción entre una aplicación web y los usuarios finales, se utilizan Interfaces Gráficas de Usuario (GUI, por sus siglas en inglés). Las primeras GUI contenían elementos simples y poco estéticos, pero con el advenimiento de Web 2.0 todo cambió de forma radical. Es común encontrar GUI's con elementos ricos en contenido multimedia, animaciones, interactividad asíncrona y sobre todo, intuitivas al usuario. Ya no basta con saber HTML para desarrollar GUI's de alta calidad. Con el surgimiento de HTML 5, Ajax y JavaScript se postulan como las tecnologías de presentación de mayor uso en la Web.

Grails no es la excepción al uso de éstas tecnologías. En éste capítulo se trata a detalle la conexión de la lógica de negocios y la capa de presentación mediante controladores, así como el uso de HTML, JavaScript y Ajax para el desarrollo de las vistas.

7.1 Generación de controladores

En la sección 4.3 se utilizó el comando *create-controller* para la creación de controladores. A estos se les agregó la propiedad *scaffold = true*. Esto genera de forma dinámica el código necesario para los controladores y las vistas. Ciertamente, son contadas las aplicaciones que satisfagan sus requerimientos de GUI con los controladores y vistas que Grails ofrece, por lo que es necesario generar el código para modificarlo.

Para la generación de los archivos correspondientes, se utiliza el comando *generate-controller*:

```
grails generate-controller mx.edu.uaeh.promep.Profesor
grails generate-controller mx.edu.uaeh.promep.Grupo
grails generate-controller mx.edu.uaeh.promep.Plaza
```

La ejecución de estos comandos generan los siguientes archivos (se omiten los archivos de prueba):

```
BDPromep/
|-- grails-app
`-- controllers
    |-- mx/edu/uaeh/promep
        |-- ProfesorController.groovy
        |-- GrupoController.groovy
        `-- PlazaController.groovy
```

Se utiliza el principio de *Convención sobre Configuración* (sección 3.8.1) para la creación de los controladores.

7.2 Análisis detallado de los controladores

Los archivos de los controladores generados poseen closures que permiten la interacción de las interfaces gráficas de usuario con la lógica de negocios. A continuación, se describen detalladamente cada uno de estos closures. Se toma como ejemplo el controlador de la clase *Profesor*.

7.2.1 *index y list*

Todo controlador tiene una acción por defecto, la cual es nombrada

index. Cuando se hace una petición al servidor con la siguiente URL:

<http://localhost:8080/BDPromep/profesor>

o bien

<http://localhost:8080/BDPromep/profesor/index>

El closure *index* es invocado. Este closure tiene la siguiente forma:

```
1. def index = {
2.     redirect(action: "list", params: params)
3. }
```

Puede observarse que *index* redirecciona a *list* enviando los parámetros que el servidor envía originalmente a *index*. El closure *list* tiene la siguiente forma:

```
1. def list = {
2.     params.max = Math.min(params.max ? params.int('max') : 10, 100)
3.     [profesorInstanceList: Profesor.list(params),
4.      profesorInstanceTotal: Profesor.count()]
5. }
```

La variable *params* es una variable de tipo *java.util.Map* que contiene los nombres de los parámetros como llave y sus valores correspondientes como valor. Estos parámetros provienen de los valores enviados junto con la URL. Por ejemplo:

<http://localhost:8080/BDPromep/profesor/list?max=10&offset=10>

En esta URL los parámetros enviados son *max* y *offset*, ambos con un valor de 10.

En la línea 2, si *params* contiene la llave *max*, se conserva su valor, en caso contrario, se establece en 10. El valor máximo permitido es

100, esto con el fin de evitar desbordamientos de la memoria en la aplicación web.

Las líneas 3 y 4 representan un mapa en Groovy que contiene las variables de salida dirigidas a la vista (en este caso, *views/profesor/list.gsp*). Este mapa se conoce como el modelo (M) en el patrón de arquitectura MVC mencionado en la sección 4.3. Se envía la lista de personas parametrizada (en caso de existir parámetros) y el total de instancias existentes, todo esto a través de las variables *profesorInstanceList* y *profesorInstanceTotal*, respectivamente. La vista se encarga de renderizar el contenido de estas variables.

7.2.2 create

Cuando se hace una petición al servidor con la siguiente URL:

<http://localhost:8080/BDPromep/profesor/create>

El closure *create* es invocado. Este closure tiene la siguiente forma:

```
1. def create = {
2.     def profesorInstance = new Profesor()
3.     profesorInstance.properties = params
4.     return [profesorInstance: profesorInstance]
5. }
```

Se define una variable llamada *profesorInstance* de tipo *Profesor* (línea 2). Los atributos de esta instancia son inicializados con los parámetros recibidos en la petición HTTP. La primera vez que se llama a *create* no tiene mucho sentido utilizar los parámetros simplemente porque no existen, pero es importante cuando se realiza la validación de datos en el closure *save*, como se verá en breve. La instancia

inicializada es enviada a la vista correspondiente (línea 4): *views/profesor/create.gsp*, la cual renderiza el contenido de *profesorInstance*.

7.2.3 *save*

Cuando se hace una petición al servidor con la siguiente URL:

<http://localhost:8080/BDPromep/profesor/create>

se obtiene una GUI similar a la mostrada en la Figura 9. Dentro del código HTML, existe un formulario que está definido de la siguiente forma:

```
<form action="/BDPromep/profesor/save" method="post" >
```

Al momento de hacer clic en *Crear*, el closure *save* es invocado. Este closure tiene la siguiente forma:

```
1. def save = {
2.     def profesorInstance = new Profesor(params)
3.     if (profesorInstance.save(flush: true)) {
4.         flash.message = "${message(code: 'default.created.message',
5.             args: [message(code: 'profesor.label', default: 'Profesor'),
6.                 profesorInstance.id])}"
7.         redirect(action: "show", id: profesorInstance.id)
8.     }
9.     else {
10.        render(view: "create",
11.            model: [profesorInstance: profesorInstance])
12.    }
13. }
```

Nuevamente, se define una variable llamada *profesorInstance* de tipo *Profesor* inicializada con los parámetros de entrada (línea 2).

Grails intenta guardar el registro en la base de datos (línea 3). Al momento de invocar el método *save*, se realiza la validación de datos especificada en el closure *constraints* de la clase de dominio. Si no hay errores, se envía un mensaje de notificación al usuario mediante la variable *flash.message* (líneas 4, 5 y 6). El mensaje enviado está especificado en los archivos *i18n/messages_XX.properties* con la etiqueta *default.created.message*. En el archivo *messages.properties*, se encuentra lo siguiente:

```
default.created.message={0} {1} created
```

El mensaje recibe 2 parámetros. En el closure *save* se envía el mensaje con el código *profesor.label*, que en caso de no existir, el mensaje por defecto es *Profesor*. El segundo parámetro es el ID del registro recién creado. Un ejemplo del mensaje formado puede apreciarse en la Figura 10.

La variable *flash* es una variable especial. De acuerdo a la documentación de Grails:

“El objeto flash es esencialmente un mapa que puede utilizarse para guardar parejas key-value. Estos valores son almacenados de forma transparente dentro de la sesión y borradas al final de la siguiente petición”

En el closure *save* se crea una variable *message* y se almacena en *flash*. Posteriormente se redirecciona al closure *show* y se manda el *id* como modelo (línea 7). Puede notarse que no se envía la variable *flash*, pero existe debido a que sobrevive a la sesión actual (*save*) y a

la siguiente (*show*). Es por eso que al renderizar el archivo *show.gsp* en la Figura 10 el mensaje aparece.

Si la validación de datos no es exitosa, se invoca al método *render*. Se muestra la vista con el nombre de *create* y se envía la instancia de la clase como modelo (líneas 11 y 12).

7.2.4 *show*

Cuando se hace una petición al servidor con la siguiente URL:

<http://localhost:8080/BDPromep/profesor/show/1>

El closure *show* es invocado. Este closure tiene la siguiente forma:

```
1. def show = {
2.     def profesorInstance = Profesor.get(params.id)
3.     if (!profesorInstance) {
4.         flash.message = "${message(code: 'default.not.found.message',
5.             args: [message(code: 'profesor.label',
6.                 default: 'Profesor'), params.id])}"
7.         redirect(action: "list")
8.     }
9.     else {
10.        [profesorInstance: profesorInstance]
11.    }
12. }
```

En la línea 2 se define una variable llamada *profesorInstance* de tipo *Profesor*, cuya instancia se construye con una llamada al método *get* de la clase *Profesor*. El parámetro que recibe este método proviene de un parámetro especial contenido en la URL. Este parámetro recibe el nombre de *id* y es el valor final de la URL justo después de la

palabra *show* (en el ejemplo mostrado es 1).

En la línea 3, si la instancia no es encontrada en la base de datos, se envía un mensaje en la variable *flash* con el código *default.not.found.message* con diversos argumentos (líneas 4, 5 y 6) y se redirecciona a la acción *list* (línea 7). Si la instancia es encontrada, se envía como modelo al archivo *show.gsp* y se renderiza.

7.2.5 *edit*

La parte inferior de la Figura 10 muestra 2 botones: *Edit* y *Delete*. Estos botones permiten editar y eliminar el registro respectivamente. Analizando el código fuente de la página web se encuentra lo siguiente:

```
1. <form action="/BDPromep/profesor/index" method="post" >
2.     <input type="hidden" name="id" value="1" id="id" />
3.     <span class="button">
4.         <input type="submit" name="_action_edit" value="Edit"
5.             class="edit" />
6.     </span>
7.     <span class="button">
8.         <input type="submit" name="_action_delete" value="Delete"
9.             class="delete"
10.            onclick="return confirm('Are you sure?');" />
11.     </span>
12. </form>
```

De forma interna, Grails invoca a las acciones *edit* y *delete* si se hace clic en los botones respectivos. El closure para *edit* tiene la siguiente forma:

```
1. def edit = {
```

```
2.     def profesorInstance = Profesor.get(params.id)
3.     if (!profesorInstance) {
4.         flash.message = "${message(code: 'default.not.found.message',
5.             args: [message(code: 'profesor.label',
6.                 default: 'Profesor'), params.id])}"
7.         redirect(action: "list")
8.     }
9.     else {
10.        return [profesorInstance: profesorInstance]
11.    }
12. }
```

Como puede observarse, es el mismo código para *edit* y para *show*. La principal diferencia radica en la página final que se renderiza si la instancia de la clase es encontrada en la base de datos, la cual depende del nombre de la acción ejecutada. Asimismo, el id utilizado para la búsqueda del registro se obtiene mediante parámetros normales en *edit* (Nótese la entrada de tipo *hidden* en el código HTML en la línea 2) y no mediante el parámetro especial utilizado en *show*.

7.2.6 update

El formulario mostrado en la Figura 11 tiene 2 botones en la parte inferior: *Update* y *Delete*.

Edit Profesor

Folio	<input type="text"/>
Paterno	<input type="text"/>
Materno	<input type="text"/>
Nombre	<input type="text"/>
Curp	<input type="text"/>
Sexo	<input type="text"/>
Email	<input type="text"/>
Telefono Trabajo	<input type="text"/>
Telefono Casa	<input type="text"/>
Anio Plantilla	<input type="text"/>
Plantilla	<input type="checkbox"/>

Figura 11: Formulario de edición de datos.

Al hacer clic en *Update*, se llama al closure con el mismo nombre. El código del controlador para esta acción es el siguiente:

```
1. def update = {
2.     def profesorInstance = Profesor.get(params.id)
3.     if (profesorInstance) {
4.         if (params.version) {
5.             def version = params.version.toLong()
6.             if (profesorInstance.version > version) {
7.
8.                 profesorInstance.errors.rejectValue("version",
9.                 "default.optimistic.locking.failure",
10.                [message(code: 'profesor.label',
11.                default: 'Profesor')] as Object[],
12.                "Another user has updated this Profesor while "+
13.                "you were editing"
14.            )

```

```
15.         render(view: "edit",
16.               model: [profesorInstance: profesorInstance])
17.         return
18.     }
19. }
20. profesorInstance.properties = params
21. if (!profesorInstance.hasErrors() &&
22.     profesorInstance.save(flush: true)) {
23.     flash.message = "${message(code: 'default.updated.message',
24.                               args: [message(code: 'profesor.label',
25.                                               default: 'Profesor'),
26.                                       profesorInstance.id])}"
27.     redirect(action: "show", id: profesorInstance.id)
28. }
29. else {
30.     render(view: "edit",
31.           model: [profesorInstance: profesorInstance])
32. }
33. }
34. else {
35.     flash.message = "${message(code: 'default.not.found.message',
36.                               args: [message(code: 'profesor.label', default: 'Profesor'),
37.                                       params.id])}"
38.     redirect(action: "list")
39. }
40. }
```

Se define una variable llamada *profesorInstance* de tipo *Profesor*, cuya instancia se construye con una llamada al método *get* de la clase *Profesor* (línea 2). El parámetro que recibe este método proviene de un campo de entrada de tipo *hidden* contenido en los parámetros de entrada. Si la instancia es encontrada (línea 3), se verifica que alguien más no haya modificado el registro a actualizar a través del atributo

version (líneas 4 y 5). Si el valor contenido en la instancia es mayor que el valor obtenido en los parámetros de entrada (línea 6), el registro no es actualizado y se envía un mensaje de error al usuario a través de la variable *errors* de la instancia (líneas 8 a 17).

Si la versión es correcta, se procede a inicializar los atributos de la instancia con los parámetros de entrada recibidos mediante el formulario (línea 20). A partir de este punto se procede de la misma forma descrita en la sección 7.2.3. Se realiza validación de datos y con base al resultado obtenido se envía a la vista correspondiente. Si la instancia no es encontrada, se notifica al usuario con un mensaje alojado en la variable *flash*.

7.2.7 delete

El segundo botón del formulario mostrado en la Figura 10 invoca a la acción *delete*. La estructura del closure correspondiente a esta acción es la siguiente:

```
1. def delete = {
2.     def profesorInstance = Profesor.get(params.id)
3.     if (profesorInstance) {
4.         try {
5.             profesorInstance.delete(flush: true)
6.             flash.message = "${message(code: 'default.deleted.message',
7.                 args: [message(code: 'profesor.label',
8.                     default: 'Profesor'),
9.                     params.id])}"
10.            redirect(action: "list")
11.        }
12.        catch (org.springframework.dao.
```

```
13.         DataIntegrityViolationException e) {
14.         flash.message = "${message(
15.             code: 'default.not.deleted.message',
16.             args: [message(code: 'profesor.label',
17.                 default: 'Profesor'),
18.                 params.id]}}"
19.         redirect(action: "show", id: params.id)
20.     }
21. }
22. else {
23.     flash.message = "${message(code: 'default.not.found.message',
24.         args: [message(code: 'profesor.label', default: 'Profesor'),
25.             params.id]}}"
26.     redirect(action: "list")
27. }
28. }
```

Se define una variable llamada *profesorInstance* de tipo *Profesor*, cuya instancia se construye con una llamada al método *get* de la clase *Profesor* (línea 2). El parámetro que recibe este método proviene de un campo de entrada de tipo *hidden* contenido en los parámetros de entrada. Si la instancia es encontrada, se procede a su eliminación (línea 3). Si ésta es exitosa, se envía un mensaje con la variable *flash* y se redirecciona a *list* (líneas 6 a 10). En caso de que ocurra algún error en la eliminación, se envía un mensaje con la variable *flash* y se redirecciona a *show* (líneas 14 a 19). Si la instancia no es encontrada, se notifica al usuario con un mensaje y se redirecciona a *list* (líneas 23 a 26).

7.3 Generación de las vistas

Para la generación de los archivos correspondientes, se utiliza el comando *generate-views*:

```
grails generate-views mx.edu.uaeh.promep.Profesor
grails generate-views mx.edu.uaeh.promep.Grupo
grails generate-views mx.edu.uaeh.promep.Plaza
```

La ejecución de estos comandos generan los siguientes archivos (se omiten los archivos de prueba):

```
BDPromep/
|-- grails-app
    |-- views
        |-- profesor
            |-- create.gsp
            |-- edit.gsp
            |-- list.gsp
            |-- show.gsp
        |-- grupo
            |-- create.gsp
            |-- edit.gsp
            |-- list.gsp
            |-- show.gsp
        |-- plaza
            |-- create.gsp
            |-- edit.gsp
            |-- list.gsp
            |-- show.gsp
```

Nuevamente, se utiliza el principio de *Convención sobre*

Configuración (sección 3.8.1) para la creación de las vistas.

7.4 Plantillas usadas para la generación de controladores y vistas

La técnica de *scaffolding* y los comandos *generate-** utilizan una serie de plantillas como base para la generación del código de los controladores y las vistas. Como se ha mencionado con anterioridad, es rara la aplicación web que utilice las vistas y controladores que Grails genera de forma automática, por lo que se hace necesario acceder al código (en caso de estar utilizando *scaffolding*) para modificarlo de acuerdo a las necesidades de la aplicación. Al utilizar los comandos *generate-**, se obtiene dicho código. Puede darse el caso de que existan muchas clases de dominio y por lo tanto que se generen muchos archivos fuente, y es bastante común que el código tenga similitudes entre sí, por lo que se hace tediosa la tarea de modificar cada uno de los archivos para lograr un mismo resultado.

Grails permite obtener el código fuente de las plantillas utilizadas en la generación de controladores y vistas. Si al momento de desarrollar una aplicación web se presenta la situación mencionada en el párrafo anterior, la modificación de las plantillas es una excelente opción para la automatización de código estilizado por el desarrollador.

El comando que permite obtener las plantillas es el siguiente:

```
> grails install-templates
```

La ejecución de dicho comando genera la siguiente estructura de

archivos y directorios:

```
BDPromep/src
|-- groovy
|-- java
`-- templates
    |-- artifacts
    |   |-- Controller.groovy
    |   |-- DomainClass.groovy
    |   |-- Filters.groovy
    |   |-- hibernate.cfg.xml
    |   |-- Script.groovy
    |   |-- Service.groovy
    |   |-- tagLib.groovy
    |   |-- Tests.groovy
    |   `-- WebTest.groovy
    |-- scaffolding
    |   |-- Controller.groovy
    |   |-- create.gsp
    |   |-- edit.gsp
    |   |-- list.gsp
    |   |-- renderEditor.template
    |   `-- show.gsp
    `-- war
        `-- web.xml
```

Las plantillas son instaladas en *src/templates*. Existen 3 tipos: *artifacts*, que contienen las plantillas utilizadas por diversos comandos de Grails para crear controladores, clases de dominio,

filtros, scripts, servicios, taglibs y tests; *scaffolding*, que contienen las plantillas de las vistas, generación de controladores y renderización de elementos en los GSP; y *war*, que contiene el archivo `web.xml`, elemento clave en las aplicaciones web hechas con Java Enterprise Edition. Se recomienda ampliamente explorar cada uno de estos archivos para conocer el funcionamiento de las plantillas y explotar al máximo esta funcionalidad.

7.5 Groovy Server Pages

Los archivos correspondientes a las vistas en Grails utilizan la tecnología denominada *Groovy Server Pages* o *GSP*. GSP es similar a *Java Server Pages* (JSP), con la diferencia de que GSP utiliza Groovy en lugar de Java como su lenguaje base y GSP es más poderoso e intuitivo.

Al igual que un JSP, un GSP permite la inclusión directa de código mediante la siguiente sintaxis:

```
<% now = new Date() %>
```

El código incrustado dentro de un JSP es denominado *scriptlet*. Sin embargo, actualmente se considera una mala práctica de programación utilizarlos. Como la necesidad de asociar la lógica de negocios no desaparece, se utiliza formas más elegantes para lograr este objetivo: *Expresiones* y *taglibs*.

7.6 Expresiones

Groovy permite el uso de expresiones para la renderización de

código Groovy en un archivo GSP. La sintaxis de una expresión es idéntica a la utilizada en JSP:

```
1. <html>
2.   <body>
3.     Hello ${params.name}
4.   </body>
5. </html>
```

Toda expresión está encerrada entre `${...}`. El uso de expresiones en las vistas generadas por Grails es muy extendido. Ejemplo de ello es el archivo `views/profesor/create.gsp`:

```
1. <g:if test="${flash.message}">
2.   <div class="message">${flash.message}</div>
3. </g:if>
4. <g:hasErrors bean="${profesorInstance}">
5.   <div class="errors">
6.     <g:renderErrors bean="${profesorInstance}" as="list" />
7.   </div>
8. </g:hasErrors>
```

En la sección 7.2.2 se menciona que algunas variables son renderizadas a la vista. Las expresiones permiten dicha renderización al manipular el valor que contienen. Las expresiones pueden renderizarse de forma directa o ser utilizadas por *taglibs*, los cuales se explican más adelante. Es común que el valor a renderizar en un GSP sea una llamada al método `toString` de la variable contenida en la expresión. Técnicamente, si el valor `toString` de una variable contiene código HTML, éste se renderiza en la página final, lo cual puede provocar resultados inesperados. Para evitarlo, se sugiere agregar la siguiente propiedad en el archivo `grails-app/conf/Config.groovy`:

```
grails.views.default.codec='html'
```

7.7 Taglibs

Una de las características más poderosas de GSP es su biblioteca de taglibs. Grails ofrece una amplia gama de funcionalidades que el desarrollador puede utilizar para diversas tareas, a la vez que puede desarrollar sus propios taglibs si los que están por defecto no satisfacen sus necesidades.

7.7.1 Toma de decisiones

Una tarea común en cualquier lenguaje de programación es la toma de decisiones. Un ejemplo de taglibs de toma de decisiones es el siguiente:

```
1. <g:if test="${name == 'Obi-Wan-Kenobi'}">
2.     Hello Jedi!
3. </g:if>
4. <g:elseif test="${name == 'Palpatine'}">
5.     Hello Sith!
6. </g:elseif>
7. <g:else>
8.     Hello unknown!
9. </g:else>
```

El taglib *g:if* permite evaluar una expresión a través de su atributo *test*. La expresión debe tener una sentencia en Groovy que devuelva un valor booleano. Si la expresión devuelve *true*, todo el código contenido dentro del taglib es renderizado. En caso contrario, simplemente se omite.

El taglib *g:elseif* funciona de la misma forma que *g:if*. Si el taglib

g:if que lo precede devuelve *false*, *g:elseif* es evaluado, en caso contrario no es tomado en cuenta. Es requisito que antes de *g:elseif* exista *g:if*. Algo similar ocurre con *g:else*, con la diferencia de que no existe expresión a evaluar. Es indispensable que antes de *g:else* exista un *g:if* o un *g:elseif*.

Un ejemplo del uso de *g:if* se encuentra en un fragmento de código en el archivo mencionado con anterioridad, *views/profesor/create.gsp*:

```
<g:if test="${flash.message}">
```

Existe un taglib que hace lo contrario de *g:if*: *g:unless*. Este taglib renderiza su contenido si la expresión contenida en su atributo *test* no se cumple:

```
1. <g:unless test="${name == 'Fred'}">
2.     Hello, my name is not Fred, is ${name}!
3. </g:unless>
```

7.7.2 Iteraciones

Otra de las funciones más utilizadas en los lenguajes de programación es el ciclo controlado por contador (*for*) o centinela (*while*). Grails permite ejecutar dichos ciclos mediante los taglibs *g:each* y *g:while*, respectivamente.

El taglib *g:each* funciona como un ciclo controlado por contador. En JSP y en GSP, la funcionalidad de un ciclo *for* se extiende para iterar a través de listas y colecciones de objetos:

```
1. <g:each in="${books}">
2.     <p>Title: ${it.title}</p>
```

```
3.     <p>Author: ${it.author}</p>
4. </g:each>
5. <g:each var="book" in="${books}">
6.     <p>Title: ${book.title}</p>
7.     <p>Author: ${book.author}</p>
8. </g:each>
9. <g:each var="book" in="${books}" status="i">
10.    <p>Iteration: ${i}</p>
11.    <p>Title: ${book.title}</p>
12.    <p>Author: ${book.author}</p>
13. </g:each>
```

El taglib *g:each* tiene 3 atributos. El atributo *in* (el único obligatorio) recibe una expresión que representa una variable. Dicha variable debe ser una lista o un rango en Groovy. El atributo *var* representa el nombre del item actual en la iteración. El atributo *status* lleva el conteo de las iteraciones realizadas, comenzando en cero. Un ejemplo de la utilización de *g:each* está en *views/profesor/list.gsp*:

```
<g:each in="${profesorInstanceList}" status="i" var="profesorInstance">
```

Existe un taglib especial, *g:eachError*, que permite iterar a través de los errores de una instancia de clase de dominio. Este taglib se explica a detalle en la sección 7.7.5:

```
1. <g:eachError bean="${book}">
2.    <li>${it}</li>
3. </g:eachError>
```

Los ciclos controlados por centinela se representan mediante el taglib *g:while*:

```
1. <g:while test="${i < 5}">
2.    <%i++%>
```

```
3.     <p>Current i = ${i}</p>
4. </g:while>
```

Este taglib, al igual que *g:if* y *g:elseif*, contiene un atributo que recibe la expresión a evaluar. Dicho taglib no es muy utilizado debido a que puede involucrar el uso de scriptlets.

7.7.3 Elementos HTML

Grails tiene soporte para todos los elementos GUI estándar utilizados en HTML. Las siguientes secciones describen cada uno de ellos.

7.7.3.1 Formularios

El uso de formularios en páginas web es una de las tareas más comunes. El taglib *g:form* permite crear un form que envía las consultas a un controlador y acción específica:

```
1. <g:form name="myForm" action="myaction" id="1">
2. ...
3. </g:form>
```

Este taglib genera el siguiente código HTML:

```
1. <form action="/shop/book/myaction/1" method="post" name="myForm"
2.     id="myForm" >
3. ...
4. </form>
```

Generalmente, todos los formularios son enviados al servidor mediante un botón de tipo *submit*. El taglib *g:submitButton* permite generar un botón de este tipo:

```
<g:submitButton name="update" value="Update" />
```

Este taglib no permite el uso de varios botones dentro de un formulario. Para lograr esto, se utiliza el taglib `g:actionSubmit`:

```
<g:actionSubmit value="${message(code: 'label.update')}}" action="Update" />
```

Un ejemplo del uso de estos taglibs que ejemplifica la diferencia de uso entre ambos se encuentra en los archivos `create.gsp` y `edit.gsp`. En `create.gsp`, se utiliza `g:submitButton` para enviar la información, mientras que en `edit.gsp` se debe utilizar `g:actionSubmit` porque se tienen 2 acciones: `Update` y `Delete`:

En `create.gsp`:

```
1. <div class="buttons">
2.     <span class="button"><g:submitButton name="create" class="save"
3.         value="${message(code: 'default.button.create.label',
4.             default: 'Create')}}" />
5.     </span>
6. </div>
```

En `edit.gsp`:

```
1. <div class="buttons">
2.     <span class="button"><g:actionSubmit class="save" action="update"
3.         value="${message(code: 'default.button.update.label',
4.             default: 'Update')}}" />
5.     </span>
6.     <span class="button"><g:actionSubmit class="delete" action="delete"
7.         value="${message(code: 'default.button.delete.label',
8.             default: 'Delete')}}"
9.         onclick="return confirm('${message(code:
10.             'default.button.delete.confirm.message',
11.                 default: 'Are you sure?')})';" />
12.     </span>
13. </div>
```

7.7.3.2 Campos de texto

El taglib *g:textField* genera un campo de captura de texto:

```
<g:textField name="myField" value="${myValue}" />
```

El único atributo requerido es *name*, el atributo *value* es opcional. Todos los elementos definidos para el elemento HTML *input* [W3C01] de tipo *text* pueden ser incluidos:

```
<g:textField name="myField" value="${myValue}" size="40" maxLength="40" />
```

El taglib *g:textArea* genera un área de texto:

```
<g:textArea name="myField" value="myValue" rows="5" cols="40"/>
```

Los atributos funcionan de la misma forma que el taglib *g:textField*. Es interesante la forma en que Grails maneja ambos taglibs en la generación de las vistas. Cuando un campo de tipo `java.lang.String` tiene en sus restricciones una longitud menor o igual a 250, se utiliza *g:textField*, en caso contrario, se usa *g:textArea*.

Para el uso de campos de texto que involucren contraseñas, Grails proporciona el taglib *g:passwordField*:

```
<g:passwordField name="myPasswordField" value="${myPassword}" />
```

Nuevamente, los atributos funcionan de la misma forma que el taglib *g:textField*.

El uso de campos ocultos (`hidden`) es común cuando se necesita manipular un valor fijo que es proporcionado por el servidor pero no debe ser manipulado por el usuario. El taglib *g:hiddenField* permite el uso de dichos campos:

```
<g:hiddenField name="myField" value="myValue" />
```

Un ejemplo de uso de este campo se encuentra en

views/profesor/show.gsp:

```
<g:hiddenField name="id" value="{profesorInstance?.id}" />
```

El valor del campo oculto es enviado desde el servidor y es necesario para identificar cuál registro es enviado a las pantallas de edición y eliminación.

7.7.3.3 Listas desplegadas

Grails facilita el uso de listas desplegadas mediante el taglib *g:select*. Este taglib recibe una lista o mapa de elementos, tal y como se muestra a continuación:

```
1. // Genera una lista utilizando un rango
2. <g:select name="user.age" from="{18..65}" value="{age}"
3.     noSelection="['':'-Choose your age-']"/>
4.
5. // Genera una lista con un valor nulo predeterminado para "No selección"
6. <g:select id="type" name='type.id' value="{person?.type?.id}"
7.     noSelection="{['null':'Select One...']}"
8.     from='{PersonType.list()}'
9.     optionKey="id" optionValue="name"/>
10.
11. // Genera selects de tipo múltiple
12. <g:select name="cars" from="{Car.list()}" value="{person?.cars*.id}"
13.     optionKey="id"
14.     multiple="true" />
```

Un ejemplo de uso de *g:select* está en el archivo GSP *views/profesor/create.gsp*:

```
1. <g:select name="profesor.id"
2.     from="{mx.edu.uaeh.promep.Profesor.list()}"
3.     optionKey="id" value="{profesorInstance?.profesor?.id}" />
```

La tecnología de *scaffolding* utiliza *g:select* para las asociaciones *one-to-many* entre las clases de dominio.

Un uso especial de listas desplegables se da con el taglib *g:datePicker*. Grails utiliza este taglib cuando una clase de dominio posee un atributo de tipo *java.util.Date*. Si la clase *Profesor* tuviera un atributo llamado *fechaNacimiento*, se obtendría código similar al siguiente:

```
1. <g:datePicker name="fechaNacimiento" precision="day"  
2.     value="{profesorInstance?.fechaNacimiento}" />
```

Por cada atributo de la fecha (año, mes, día, hora y minuto), se genera una lista desplegable. Un atributo interesante de *g:datePicker* es *precision*. Este atributo permite especificar la precisión máxima que el usuario puede manipular. En el ejemplo mostrado, se generan 3 listas desplegables para año, mes y día, respectivamente. El número máximo de listas desplegables es 5.

7.7.3.4 Elementos de selección

El taglib *g:checkbox* permite renderizar un elemento HTML de tipo *checkbox*. Normalmente es utilizado para representar valores de tipo booleano:

```
<g:checkbox name="myCheckbox" value="{true}" />
```

Otros elementos HTML para selección bastante comunes son los *radioButton*. Grails posee 2 taglibs para el manejo de ellos. El primero es *g:radio*, que genera el código necesario para un solo elemento:

```
1. <g:radio name="myGroup" value="1"/>  
2. <g:radio name="myGroup" value="2" checked="true"/>
```

```
3.
4. // El código HTML equivalente es:
5. <input type="radio" name="myGroup" value="1" />
6. <input type="radio" name="myGroup" checked="checked" value="2" />
```

El segundo taglib es *g:radioGroup*, que permite manejar una lista de *radioButton*. La ventaja de este taglib es que únicamente permite seleccionar un elemento a la vez:

```
1. <g:radioGroup name="myGroup" values="[1,2,3]" value="1" >
2.     <p><g:message code="${it.label}" />: ${it.radio}</p>
3. </g:radioGroup>
4. // El código HTML equivalente es:
5. <p>Radio 1: <input type="radio" name="myGroup" value="1"
6.     checked="checked" /></p>
7. <p>Radio 2: <input type="radio" name="myGroup" value="2" /></p>
8. <p>Radio 3: <input type="radio" name="myGroup" value="3" /></p>
```

Dentro del cuerpo del taglib *g:radioGroup* se pueden utilizar 2 variables: *it.label* e *it.radio*, las cuales pueden utilizarse para la etiqueta del elemento actual y el elemento en sí, respectivamente.

7.7.3.5 Carga de archivos

Si dentro de una clase de dominio se especifica un atributo de tipo *byte[]*, Grails lo renderiza en las vistas como un archivo a cargar. El taglib utilizado es *g:uploadForm*:

```
1. <g:uploadForm action="guardaArchivo">
2.     <input type="file" name="myFile" />
3. </g:uploadForm>
```

En la base de datos, los atributos de tipo *byte[]* se mapea como campos de tipo *BLOB* (o su equivalente, dependiendo del gestor utilizado). Si se desea guardar el archivo en disco en lugar de utilizar

la base de datos, se debe utilizar una acción similar a la que se muestra en la siguiente pieza de código:

```
1. def guardaArchivo = {
2.     // Se utiliza un objeto MultipartFile de Spring
3.     def mf = request.getFile( 'myFile' )
4.     // El archivo a guardar no debe sobrepasar los 200 KB
5.     if ( !mf?.empty && mf.size < 1024*200 ) {
6.         mf.transferTo( new File( "/images/myFile.txt" ) )
7.     }
8. }
```

7.7.3.6 Vínculos y recursos estáticos

Grails permite la creación de vínculos en base a sus controladores, acciones e ids. El taglib *g:link* genera elementos HTML de tipo *anchor*, que son la base de los vínculos en la Web:

```
1. <g:link action="show" id="1">Book 1</g:link>
2. <g:link action="show" id="${currentBook.id}">${currentBook.name}</g:link>
3. <g:link controller="book">Book Home</g:link>
```

Existe otro taglib, *g:createLink* que permite generar únicamente la URL sin necesidad de utilizar el elemento *anchor*. Este taglib es extremadamente útil en la generación de vínculos en código JavaScript:

```
1. <g:createLink action="show" id="1" /> == /shop/book/show/1
2. <g:createLink controller="book" /> == /shop/book
3. <g:createLink controller="book" action="list" /> == /shop/book/list
```

Ya que se menciona JavaScript, Grails permite la renderización correcta de bloques de código de este lenguaje mediante el taglib *g:javascript*:

```
1. // Llama al archivo '/app/js/myscript.js'
2. <g:javascript src="myscript.js" />
3. // Carga todos los archivos js necesarios para utilizar Scriptaculous
4. <g:javascript library="scriptaculous" />
5. <g:javascript>alert('hello')</g:javascript>
```

La carga de recursos estáticos (archivos JS, CSS, imágenes, etc) se facilita mediante el taglib *g:resource*:

```
1. //Genera "/shop/css/main.css"
2. <g:resource dir="css" file="main.css" />
3.
4. // Genera "http://portal.mygreatsite.com/css/main.css"
5. <g:resource dir="css" file="main.css" absolute="true"/>
6.
7. // Genera "http://admin.mygreatsite.com/css/main.css"
8. <g:resource dir="css" file="main.css"
9.     base="http://admin.mygreatsite.com"/>
```

En las primeras versiones de Grails se utilizaba *g:createLinkTo* para la carga de recursos estáticos, pero este taglib ha sido declarado como obsoleto y se recomienda utilizar *g:resource* en su lugar.

7.7.4 Tablas

Las vistas generadas por Grails, especialmente los archivos *list.gsp*, utilizan tablas para desplegar los registros almacenados en la base de datos. Estas tablas tienen la propiedad de paginación, lo cual facilita la navegación de registros. El taglib *g:paginate* es utilizado para realizar la paginación de tablas. La siguiente pieza de código pertenece al archivo *views/profesor/list.gsp*

```
<g:paginate total="{profesorInstanceTotal}" />
```

el taglib puede ser editado con varios atributos:

```
1. <g:paginate controller="profesor" action="list"
2.     total="{Profesor.count()}" />
```

Acompañando a la paginación, las columnas de las tablas de Grails pueden ordenarse de forma ascendente o descendente. El taglib utilizado para implementar la ordenación es *g:sortableColumn*:

```
1. <g:sortableColumn property="id"
2.     title="{message(code: 'profesor.id.label', default: 'Id')}" />
```

7.7.5 Manejo de errores en clases de dominio

Tal y como se mencionó en la sección 4.3, las clases de dominio pueden tener validación de datos. Si al momento de crear un registro se tienen errores, al usuario se le muestra una serie de mensajes. En el archivo *views/profesor/create.gsp* se observa el siguiente código:

```
1. <g:hasErrors bean="{profesorInstance}">
2.     <div class="errors">
3.         <g:renderErrors bean="{profesorInstance}" as="list" />
4.     </div>
5. </g:hasErrors>
```

El taglib *g:hasErrors* verifica si en el bean proporcionado existen errores. De encontrarse, el taglib *g:renderErrors* proporciona una lista de los errores encontrados en la instancia *profesorInstance*.

Otro taglib que permite la manipulación directa de los errores es *g:eachError*:

```
1. <g:eachError bean="{profesorInstance}">
2.     <li>{it}</li>
3. </g:eachError>
```

Este taglib hace una iteración a través de todos los errores encontrados, con la diferencia que el objeto *it* dentro de la iteración

contiene el error, lo que facilita su manipulación.

7.7.6 Formato de datos

Algunos de los atributos utilizados en las clase de dominio necesitan presentarse al usuario en una forma más “amigable”. Un ejemplo muy común es el uso de valores booleanos. Es evidente que al usuario no se le puede mostrar el valor de un atributo como “true” o “false”. Grails proporciona en taglib *g:formatBoolean* para estos casos:

```
<g:formatBoolean boolean="${myBoolean}" true="True!" false="False!" />
```

Los atributos *true* y *false* se utilizan para especificar el texto a mostrar en cada caso, respectivamente.

Otro dato muy común es *java.util.Date*, cuya representación por defecto no es muy amigable. El taglib *g:formatDate* permite dar un formato adecuado a las fechas:

```
<g:formatDate format="yyyy-MM-dd" date="${date}" />
```

En el ejemplo, el 25 de noviembre de 1980 se despliega como *1980-11-25*. El atributo *format* esta basado en la documentación de la clase *java.text.SimpleDateFormat* [JAV01].

Existen aplicaciones que requieren presentar cifras y cantidades monetarias en un formato específico. El taglib *g:formatNumber* facilita esta tarea:

```
1. <g:formatNumber number="${myNumber}" format="\$###,##0" />
2. <g:formatNumber number="${myCurrencyAmount}" type="currency"
3.     currencyCode="USD" />
```

7.7.7 Uso de variables

A continuación se explican algunos taglibs que son útiles para el manejo de variables enviadas por el controlador hacia la vista.

El taglib *g:set* crea una variable dentro del GSP de una forma elegante. Dicha variable puede ser utilizada únicamente dentro del archivo GSP donde se está declarando:

```
1. <g:set var="counter" value="{1}" />
2. <g:each in="{list}">
3.     ${counter}.&nbsp; ${it} -> ${counter % 2 == 0 ? 'even' : 'odd'}
4.     <g:set var="counter" value="{counter + 1}" /><br/>
5. </g:each>
```

El taglib *g:fieldValue* permite obtener el valor del atributo de una variable o instancia de clase de dominio. Su principal ventaja radica en que el valor del atributo es codificado para desplegarse de forma apropiada en la página Web:

```
1. <p>${book.title}</p>
2. <p><g:fieldValue bean="{book}" field="title" /></p>
```

Supónganse que el atributo `title` tiene un valor “Título de la obra”. El primer ejemplo renderiza “Título de la obra” en negrita, y en algunos navegadores el acento suele aparecer de forma incorrecta. El segundo ejemplo, en cambio, utilizará secuencias de escape de HTML para mostrar el contenido correctamente.

El taglib *g:findAll* utiliza el método *findAll* del JDK de Groovy para iterar a través de ciertos elementos que cumplan con una condición:

```
1. Stephen King's Books:
2. <g:findAll in="{books}" expr="it.author == 'Stephen King'">
3.     <p>Title: ${it.title}</p>
```

```
4. </g:findAll>
```

El ejemplo anterior itera a través de todos los atributos *author* de la variable *books* cuyo contenido sea igual a *Stephen King*.

El taglib *g:collect* funciona de la misma forma que *g:findAll*, con la diferencia de que no existe expresión booleana a evaluar:

```
1. Books titles:  
2. <g:collect in="{books}" expr="it.title">  
3.     <p>Title: ${it}</p>  
4. </g:collect>
```

El taglib *g:grep* utiliza el método *grep* del JDK de Groovy y funciona de la misma forma que *g:findAll*, con la diferencia de que se usa un filtro en lugar de una expresión y dicho filtro puede ser una clase, una expresión regular, un valor booleano, etc.

```
1. Stephen King's non-fiction Books:  
2. <g:grep in="{books}" filter="NonFictionBooks.class">  
3.     <p>Title: ${it.title}</p>  
4. </g:grep>  
5. <g:grep in="{books.title}" filter="~/.*Groovy.*/">  
6.     <p>Title: ${it}</p>  
7. </g:grep>
```

El taglib *g:join* utiliza el método *join* del JDK de Groovy para concatenar la representación *toString* de cada elemento de una colección, utilizando un delimitador entre cada una:

```
1. <g:join in="['Grails', 'Groovy', 'Gradle']" delimiter="_"/>  
2. // El resultado es Grails_Groovy_Gradle
```

7.8 Incorporando Web 2.0 mediante Ajax

La documentación de Grails define su soporte para Ajax de la

siguiente manera:

“Ajax es el acrónimo de Asynchronous Javascript and XML y es el motor detrás de las Rich Internet Applications (RIA’s). Este tipo de aplicaciones se adaptan mejor a los frameworks de desarrollo ágil de aplicaciones escritas en lenguajes como Ruby y Groovy. Grails proporciona soporte para la creación de aplicaciones Ajax a través de su colección de taglibs para esta tecnología”

Por defecto, Grails trabaja con una biblioteca escrita en Javascript llamada *Prototype*, y a través de sus plugins se puede tener soporte para otro tipo de herramientas, como *Dojo*, *Yahoo UI*, *Google Web Toolkit* y *jQuery*.

La característica principal de Ajax es que la actualización de elementos HTML se realiza de forma asíncrona, es decir, la tarea de actualizar no involucra toda la página, sino únicamente ciertos elementos de la misma. Esto proporciona un mayor dinamismo e interacción a las páginas Web.

El primer paso para utilizar los taglibs de Ajax en Grails es importar la biblioteca Javascript necesaria dentro del elemento `<head>` del archivo GSP:

```
1. // Importación de la biblioteca Prototype
2. <g:javascript library="prototype" />
3. // Importación de la biblioteca Scriptaculous para animaciones
4. <g:javascript library="scriptaculous" />
```

Como se menciona al principio de esta sección, el soporte para Ajax

en Grails es a través de taglibs. A continuación se describen detalladamente cada uno de ellos.

7.8.1 *g:remoteLink*

Este taglib crea un link que al ser utilizado realiza una llamada remota al servidor:

```
1. <g:remoteLink action="show" id="1" update="divShow">
2.     Show Book 1
3. </g:remoteLink>
```

Esto genera un link que al ser utilizado invoca a la acción *show* del controlador correspondiente. La respuesta de dicha acción se renderiza como contenido del elemento *divShow*. En Grails, las acciones que son llamadas por los taglibs de Ajax utilizan el método *render* para devolver contenido HTML:

```
1. def show = {
2.     render "<p>${Book.get( params['id'] )}</p>"
3. }
```

No es una buena práctica combinar el código HTML en el controlador, ya que se viola el patrón de arquitectura MVC utilizado en Grails. Para solucionar esto, Grails proporciona plantillas (templates) que permiten separar y reutilizar fragmentos de archivos GSP. Para el ejemplo mostrado, la plantilla estaría en el archivo ubicado en *grails-app/views/book/_bookTemplate.gsp*, y el contenido del archivo sería el siguiente:

```
<p>${bookInstance}</p>
```

Y el código de la acción dentro del controlador se cambia de la

siguiente forma:

```
1. def show = {
2.     render ( template:"bookTemplate",
3.             model:[ bookInstance:Book.get( params['id'] ] )
4. }
```

De esta forma, la respuesta recibida del servidor es un párrafo HTML con la representación *toString* de la instancia solicitada. Este contenido se renderiza en el elemento *divShow* de la página web.

7.8.2 *g:remoteField*

La validación de datos en tiempo real es una necesidad que tienen muchas aplicaciones web. Ejemplo de ello son los servidores de correo. Cuando un usuario se registra, normalmente se le pide un nombre de usuario. La aplicación verifica que el nombre de usuario introducido aún no exista y notifica el resultado de su búsqueda sin necesidad de recargar toda la página. El taglib *g:remoteField* genera un campo de texto que envía una llamada remota al servidor cada vez que su contenido se actualiza. En el ejemplo práctico se hace una validación del ISBN de un libro:

```
1. <g:remoteField name="isbn" maxlength="13"
2.     value="{libroInstance?.isbn}" action="validateIsbn"
3.     update="isbnValidationMessage" paramName="isbn"/>
4. <div id="isbnValidationMessage"></div>
```

La acción dentro del controlador se define de la siguiente manera:

```
1. def validateIsbn = {
2.     if (params.isbn.length() != 13 ) {
3.         render "El ISBN introducido debe tener 13 caracteres"
4.     }
```

```
5.     else {
6.         def libroInstance = Libro.findByIsbn( params.isbn )
7.         if ( libroInstance ) {
8.             render "El ISBN introducido ya existe"
9.         }
10.        else {
11.            render "El ISBN introducido es v\u00E9lido"
12.        }
13.    }
14. }
```

Aquí, el método *render* no hace una llamada a una plantilla, sino que envía una cadena de caracteres de forma directa. Dicho método tiene una versatilidad excelente, por lo que se recomienda consultar la documentación oficial de Grails para conocer su funcionalidad a fondo.

7.8.3 *g:remoteFunction*

El taglib *g:remoteFunction* genera el código JavaScript necesario para realizar una llamada remota al servidor. Este taglib puede ser utilizado de varias formas:

```
1. // Llamada a g:remoteFunction en el método onchange del elemento HTML
2. <g:select from="[1,2,3,4,5]" onchange="$ {
3.     remoteFunction(action:'bookByName', update:[success:'great',
4.         failure:'ohno'],
5.     options=[asynchronous:false]})" />
6. // Llamada a g:remoteFunction junto con el uso de una
7. // biblioteca JavaScript
8. $('mydiv').onclick = <g:remoteFunction action="show" id="1" />
```

Nótese el uso del taglib como un método. Todos los taglibs pueden utilizarse ya sea como métodos o como taglibs. La forma de utilizarlos

depende de la respuesta requerida en el servidor y del orden de renderización de los elementos.

7.8.4 *g:formRemote*

El taglib *g:formRemote* permite el uso de formularios con Ajax. Los datos contenidos dentro del formulario son serializados y enviados de forma remota al servidor. Una característica de *g:formRemote* es que si el navegador no soporta JavaScript, el formulario se trata de forma normal, es decir, sin llamadas remotas.

```
1. <g:formRemote name="myForm" on404="alert('not found!')"  
2.     update="updateMe" url="[action:'show']">  
3.     Login: <input name="login" type="text"></input>  
4. </g:formRemote>  
5. <div id="updateMe">this div is updated by the form</div>
```

En este ejemplo existe un atributo interesante: *on404*. Los taglibs de Grails para Ajax soportan llamadas a funciones JavaScript en caso de que la llamada remota al servidor envíe un código de error. El dinamismo de Grails permite escribir cualquier código de error deseado (*on404*, *on405*, *on500*, etc).

7.8.5 *g:submitToRemote*

El taglib *g:submitToRemote* también permite el uso de formularios con Ajax. La principal diferencia con el taglib *g:formRemote* es que se genera un botón que serializa todos los atributos contenidos en el formulario donde se encuentra el taglib y hace una llamada remota al servidor.

```
1. <g:form action="show">
2.     Login: <input name="login" type="text"></input>
3.     <g:submitToRemote update="updateMe" />
4. </g:form>
5. <div id="updateMe">this div is updated by the form</div>
```

Este taglib contiene prácticamente las mismas funciones que los taglibs mencionados anteriormente.

7.9 Resumen

Este capítulo ha manejado la capa de presentación de las aplicaciones web realizadas con Grails. Se ha detallado el funcionamiento de los controladores generados mediante la técnica de *scaffolding*, se generaron las plantillas utilizadas por Grails para la creación de diversos artefactos y se realizó un análisis exhaustivo de los taglibs de Grails utilizados en los archivos GSP.

El conocimiento adquirido hasta el momento proporciona los cimientos básicos para la realización ágil de aplicaciones web con Grails. En la tercera parte se trata el desarrollo de API's tipo REST y el uso de plugins.

Parte 4 Tópicos avanzados de Grails

Los tópicos tratados en la Parte 3 sirven como fundamento para que el lector pueda realizar aplicaciones web completamente funcionales. En esta sección se presentan 2 de los temas más avanzados y utilizados en la actualidad: web services y reutilización de software.

En el *Capítulo 8: Desarrollo de API's tipo REST con Grails*, se explica la forma en que Grails implementa las 4 operaciones del protocolo HTTP utilizadas en REST (GET, POST, PUT y DELETE) mediante controladores, así como la forma en que los datos son enviados y recibidos mediante uno de los formatos ampliamente utilizados para la comunicación entre 2 aplicaciones distintas: XML (Extensible Markup Language).

Para finalizar el estudio de Grails, en el *Capítulo 9: Plugins*, se destaca la aceleración del desarrollo de software mediante la reutilización de piezas ya existentes y se motiva al lector en el uso de la biblioteca de complementos de Grails. Se muestra la funcionalidad de algunos de estos complementos como es la exportación de datos hacia diversos formatos, ingeniería inversa de bases de datos y la creación de tablas dinámicas mediante Ajax con *JQGrid*.

Capítulo 8 Desarrollo de API's tipo REST con Grails

El desarrollo de aplicaciones Web 2.0 ha generado una necesidad debido al crecimiento exponencial de la conexión en red de las personas: la comunicación remota de diversas aplicaciones cuyos lenguajes de programación e inclusive los dispositivos que las ejecutan no son necesariamente los mismos. Es común consultar el correo electrónico tanto en una computadora de escritorio como en un dispositivo móvil (teléfono celular, smartphome, blackberry, etc.); ha crecido en gran medida el número de aplicaciones tipo cliente que se conectan a Twitter y Facebook sin interferir en la infraestructura interna de estas redes sociales. El desarrollo de API's tipo REST está tomando mucha fuerza y se está convirtiendo en una de las tecnologías mas utilizadas para lograr la comunicación remota entre clientes y servidores.

8.1 ¿Que es un API?

Un API es una interfaz de programación de aplicaciones (*Application Programming Interface*, por sus siglas en inglés). Básicamente es una colección de funciones, clases y diversos componentes de un lenguaje, framework o aplicación que facilita a los desarrolladores la generación de software.

Una de las primeras API's fue la Biblioteca Estándar de Plantillas del lenguaje C++. Asimismo, Java posee un conjunto de clases y elementos probados que pueden usarse para evitar la repetición de código. Todas las bibliotecas de clases desarrolladas por terceros

mediante Java son consideradas API's. En los últimos años, el concepto de API se ha extendido para incluir los servicios tipo REST que las aplicaciones web ofrecen.

8.2 Arquitectura REST

REST es el acrónimo de *Representational State Transfer* (Transferencia de Estado Representacional). De acuerdo a [WIK01], REST afirma que la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales clave:

- **Un protocolo cliente/servidor sin estado.** Cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URL's, no son permitidas por REST)
- **Un conjunto de operaciones bien definidas que se aplican a todos los recursos de información.** HTTP en sí define un conjunto pequeño de operaciones, las más importantes son *POST*, *GET*, *PUT* y *DELETE*. Con frecuencia estas operaciones se equiparan a las operaciones *CRUD* (sección 3.8) que se requieren para la persistencia de datos, aunque *POST* no encaja exactamente en este esquema.
- **Una sintaxis universal para identificar los recursos.** En un

sistema REST, cada recurso es direccionable únicamente a través de su URI.

- **El uso de hipermedios.** La representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional. Esta característica permite la comunicación entre diversas aplicaciones que no necesariamente están escritas en el mismo lenguaje de programación.

8.3 Implementación de un API tipo REST

El desarrollo de un API tipo REST en Grails es sencillo. Para comenzar, se crea un controlador llamado *LibroRest*:

```
> grails create-controller mx.edu.uaeh.promep.LibroRest
```

Como puede notarse, se va a generar un API REST para los libros de los profesores contenidos en el sistema. Una vez generado el controlador, se escribe el siguiente código dentro del mismo:

```
1. package mx.edu.uaeh.promep
2.
3. import grails.converters.XML
4.
5. class LibroRestController {
6.
7.     static allowedMethods = [list: "GET", show: "GET", create: "POST"]
8.
9.     def index = {
```



```
10.     redirect(action: "list", params: params)
11.   }
12.
13.   def list = {
14.     params.max = Math.min(params.max ? params.int('max') : 10, 100)
15.     render Libro.list(params) as XML
16.   }
17.
18.   def show = {
19.     def libroInstance = Libro.findByIsbn(params.id as String)
20.     if (!libroInstance) {
21.       def errorMessage = {
22.         error("Libro not found with isbn \"${params.id}\"")
23.       }
24.       render(contentType: "text/xml; charset=utf-8", errorMessage)
25.     }
26.     else {
27.       render libroInstance as XML
28.     }
29.   }
30.
31.   def create = {
32.
33.     def xml = request.XML
34.     def libroInstance = new Libro()
35.     libroInstance.isbn = xml.isbn.text()
36.     libroInstance.titulo = xml.titulo.text()
37.     libroInstance.autor = xml.autor.text()
38.     libroInstance.edicion = xml.edicion.text() as Short
39.     libroInstance.categoria = xml.categoria.text()
40.     libroInstance.existenciaTotal =
41.       xml.existenciaTotal.text() as Integer
42.     libroInstance.cantidadPrestada =
43.       xml.cantidadPrestada.text() as Integer
```

```
44.
45.     def message
46.     if (libroInstance.save(flush:true)) {
47.         message = {
48.             status("Libro successfully created")
49.         }
50.     }
51.     else {
52.         message = {
53.             status("Fail to create libro")
54.         }
55.     }
56.     render(contentType: "text/xml; charset=utf-8", message)
57. }
58.
59. }
```

Puede notarse la existencia de una línea de código para la utilización de la clase *grails.converters.XML* (línea 3). Esta clase es necesaria para enviar una respuesta tipo XML al cliente que haga peticiones al API. La línea de código:

```
static allowedMethods = [list: "GET", show: "GET", create: "POST"]
```

Indica el tipo de método REST a utilizar en cada una de las acciones definidas en el controlador. Para mostrar una lista o un elemento individual, se utiliza el método GET. Para generar nuevos registros, se utiliza el método POST. Si se utiliza otro método, el servidor arrojará un error con código 405.

Dentro del controlador existen 4 acciones: *index*, *list*, *show* y *create*. La primera redirige a *list* (sección 7.2.1), la cual contiene la siguiente línea de código:

render Libro.list(params) as XML

Se hace uso del método *render* para enviar una lista en formato XML de libros de acuerdo a los parámetros introducidos (sección 6.3.2). La Figura 12 muestra la respuesta enviada al navegador al entrar a la dirección <http://localhost:8080/BDPromep/libroRest/>.

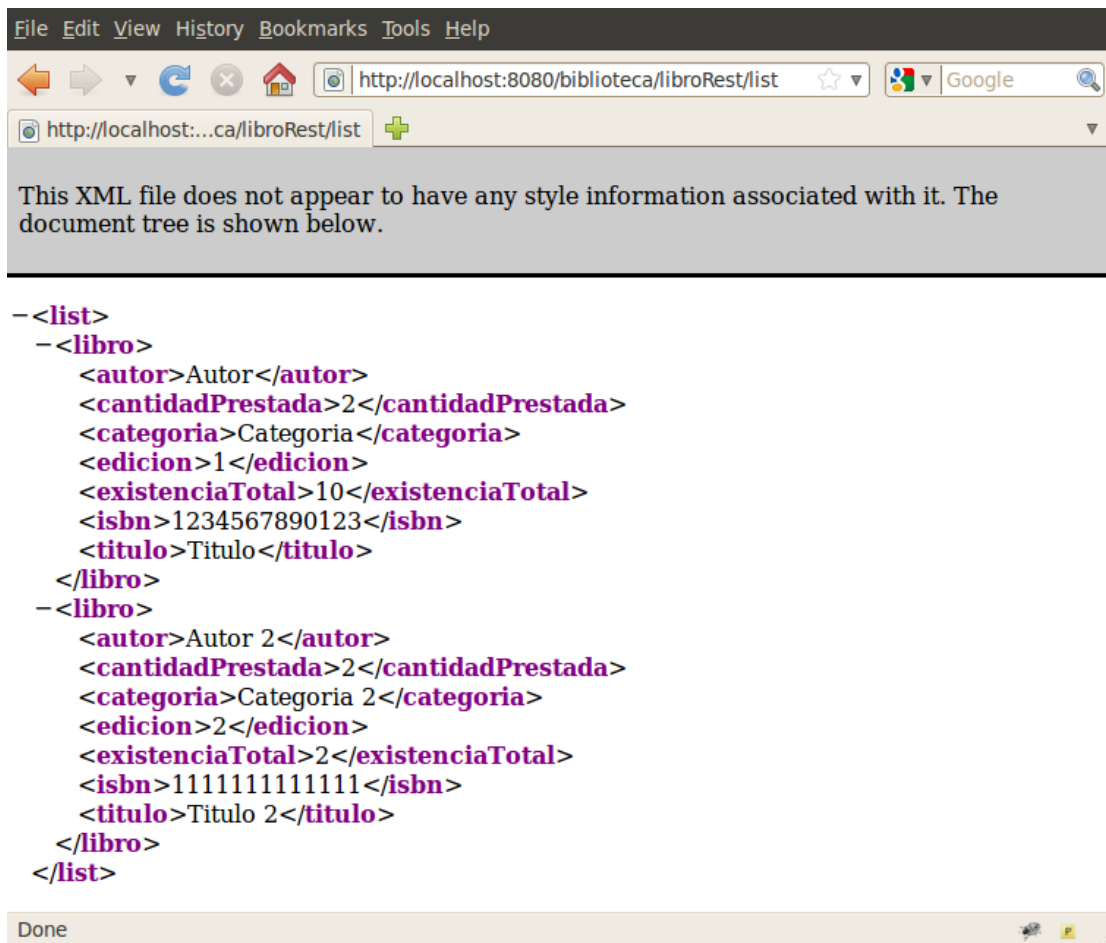


Figura 12: Respuesta del servidor al invocar la acción *list* del API REST.

Con una sola línea de código, Grails ha convertido una lista de objetos en un documento XML que puede ser utilizado en cualquier

aplicación para desplegar información acerca de los libros existentes.

El método *show* funciona de la misma forma. La Figura 13 muestra la respuesta del servidor al buscar el libro con ISBN igual a 1234567890123.



Figura 13: Respuesta del servidor con el método *show*.

¿Que ocurre si se introduce un ISBN que no existe? Las líneas 20 a 24 realizan una validación del ISBN introducido. En caso de no encontrarse un libro con el ISBN enviado, se envía un mensaje de error notificando al cliente que el registro no existe. La Figura 14 muestra la respuesta del servidor al buscar el ISBN “isbninvalido”.

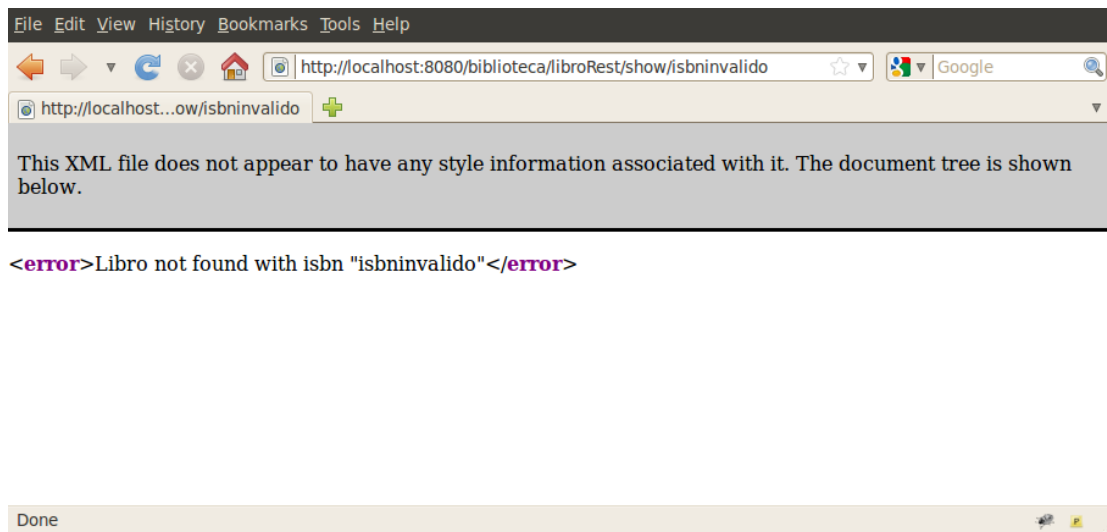


Figura 14: Respuesta del servidor con el método `show` con un isbn inválido.

La respuesta con el mensaje de error es creada mediante un closure llamado `errorMessage`. Este closure es pasado como un parámetro al método `render`, así como el tipo de respuesta enviada:

```
def errorMessage = {  
    error("Libro not found with isbn \"${params.id}\"")  
}  
render(contentType: "text/xml; charset=utf-8", errorMessage)
```

Las acciones `list` y `show` pueden utilizarse con un navegador web porque están especificadas con el método GET. La acción `create` necesita un tratamiento especial, ya que se necesita enviar un archivo XML al servidor. Las siguientes secciones hacen uso de una herramienta especial que permite hacer peticiones REST.

8.4 Experimentando con el API

El objetivo de la creación de un API REST es permitir a diversas

aplicaciones el acceso remoto a diversos recursos de la aplicación web. Una forma de probar las acciones tipo GET del API es a través de un navegador web. Las acciones de otro tipo necesitan ser utilizadas de otra forma.

Firefox permite la creación de peticiones HTTP mediante su plugin *Poster*. Mediante esta herramienta se puede probar la acción *create* del API recién creada. La Figura 15 muestra la interfaz gráfica de *Poster* para hacer la petición POST.

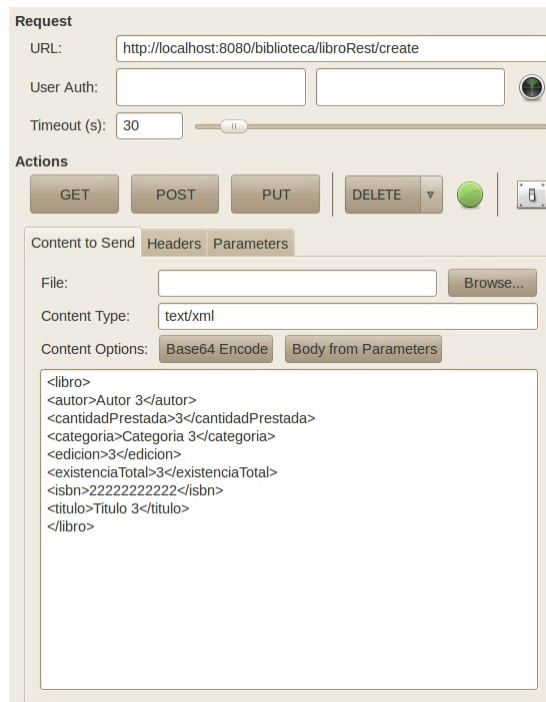


Figura 15: Plugin Poster para la creación de peticiones HTTP.

Al presionar el botón POST, se obtiene la siguiente respuesta del servidor.

```
<status>Libro successfully created</status>
```

8.5 Seguridad en API's tipo REST

Una de los aspectos más importantes en las peticiones HTTP de un API REST es la seguridad. Cuando comenzaron a utilizarse las API's tipo REST, no existían mecanismos de seguridad que garantizaran el acceso restringido a recursos del servidor. La autenticación estándar en HTTP no permitía la encriptación de los datos, por lo que la vulnerabilidad del servidor era evidente.

Hace unos años comenzó el desarrollo de un protocolo denominado *OAuth*. Este protocolo permite restringir el acceso a las API's REST de los servidores mediante el uso de credenciales enviadas como parámetros ocultos. Al momento de escribir este documento, el desarrollo de OAuth en su versión 2.0 corre a cargo de Facebook.

Con el paso del tiempo, se espera que el uso de OAuth 2.0 se difunda como un estándar en la seguridad de API's tipo REST.

Para un tratamiento detallado de OAuth, se sugiere visitar [OAU01].

8.6 Ejemplo de la vida real: el API de Twitter

Las redes sociales como Twitter y Facebook se ha caracterizado por permitir a aplicaciones ajenas a su infraestructura el acceso a sus recursos. En esta sección se muestra un ejemplo del uso del API de Twitter.

Twitter ofrece la documentación de su API en [TWI01] (Figura 16). Este documento online muestra una lista de las URI's disponibles

para su uso, así como los parámetros que pueden recibir.

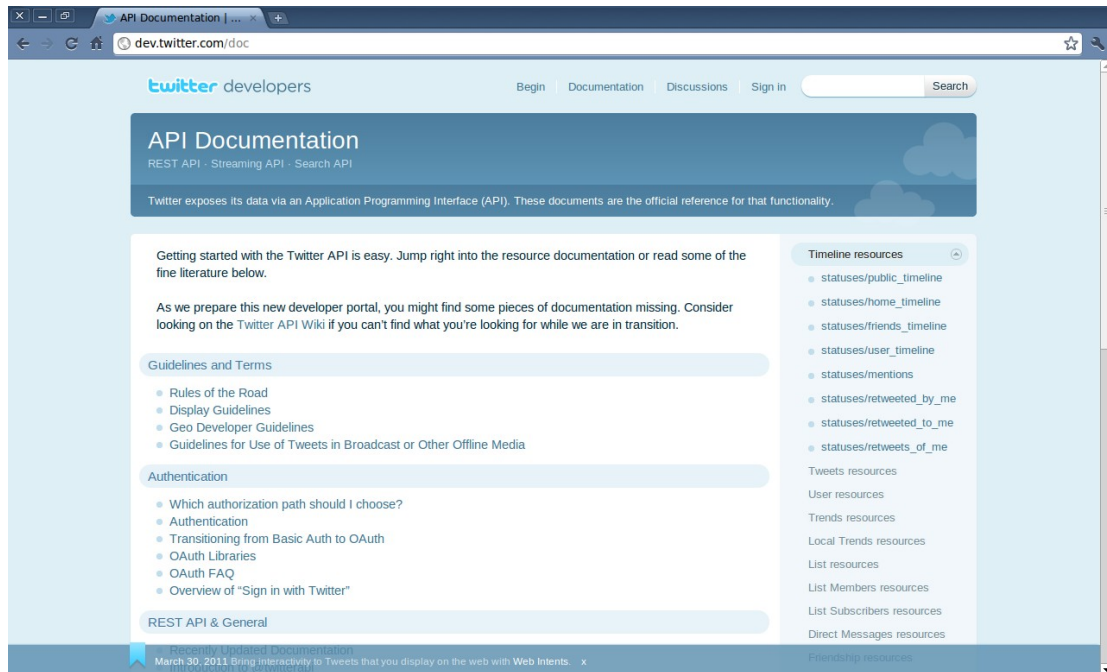


Figura 16: Página principal del API de Twitter.

Twitter hace uso de OAuth 1.0, por lo que el uso de ciertas URI's requieren autenticación. Para demostrar el uso de la misma, se utiliza una URL que devuelve el timeline en tiempo real. Para ello, se escribe la siguiente dirección web en el navegador: http://api.twitter.com/1/statuses/public_timeline.xml. El API nos envía una respuesta similar a la mostrada en la Figura 17.


```

File Edit View Help
<?xml version="1.0" encoding="UTF-8"?>
<statuses type="array">
<status>
<created_at>Tue Apr 26 17:33:05 +0000 2011</created_at>
<id>6293222361088000</id>
<text>o @tooka foi repicar o cabelo,6#233; bom ele s6#243; REPICAR u.u se n6#227;o vai apanha sexta kk</text>
<source>web</source>
<truncated>false</truncated>
<favorited>false</favorited>
<in_reply_to_status_id</in_reply_to_status_id>
<in_reply_to_user_id</in_reply_to_user_id>
<in_reply_to_screen_name</in_reply_to_screen_name>
<retweet_count>0</retweet_count>
<retweeted>false</retweeted>
<user>
<id>176867652</id>
<name>raa cazelli</name>
<screen_name>raaczelli</screen_name>
<location>http://raaczelli.tumblr.com/</location>
<description></description>
<profile_image_url>http://a1.twimg.com/profile_images/1231035604/nshkhfksu_normal.jpg</profile_image_url>
<url>http://www.orkut.com.br/Main#Profile?uid=18035099424774113260</url>
<protected>false</protected>
<followers_count>257</followers_count>
<profile_background_color>0f0507</profile_background_color>
<profile_text_color>e1294</profile_text_color>
<profile_link_color>5ffa05</profile_link_color>
<profile_sidebar_fill_color>ed1f6</profile_sidebar_fill_color>
<profile_sidebar_border_color>22f222</profile_sidebar_border_color>
<friends_count>476</friends_count>
<created_at>Tue Aug 10 17:11:07 +0000 2010</created_at>
<favorites_count>0</favorites_count>
<utc_offset>-1400</utc_offset>
<time_zone>Santiago</time_zone>
<profile_background_image_url>http://a2.twimg.com/profile_background_images/209646702/0gAAAG3MLf32G9J2R9yR2vPqdl9VIndja0b8ChBzKqD0LB0ST90x3TcH39uX1G06uSvHAnLnehR6v1SFKY_i2PI:
<profile_background_tile>true</profile_background_tile>
<profile_use_background_image>true</profile_use_background_image>
<notifications</notifications>
<geo_enabled>false</geo_enabled>
<verified>false</verified>
<following</following>
<status_count>8346</status_count>
<lang>es</lang>
<contributors_enabled>false</contributors_enabled>
<follow_request_sent></follow_request_sent>

```

Figura 17: Respuesta del API de Twitter al solicitar el timeline público.

Como ya se había mencionado anteriormente, la respuesta en XML permite a cualquier aplicación hacer uso de la información recibida.

8.7 Resumen

Grails contiene un soporte robusto y estable que permite el desarrollo de API's tipo REST de forma rápida y sencilla. El contenido mostrado en este capítulo es solo una pequeña parte del soporte REST de Grails, por lo que se sugiere consultar la documentación oficial para un tratamiento más profundo.

Para concluir el estudio de Grails, el siguiente capítulo hace uso de una de las capacidades más poderosas de este framework: el uso de plugins.

Capítulo 9 Plugins

Uno de los aspectos más importantes de la programación orientada a objetos es la reutilización de código. En muchas disciplinas se hace uso de la reutilización de componentes para evitar la repetición de código. Un ejemplo de ello es la Electrónica, en la cual se dispone de un catálogo de elementos electrónicos que el diseñador de circuitos puede utilizar para construir nuevos aparatos, acelerando su desarrollo y basando su trabajo en componentes ya existentes.

El mundo del desarrollo de software no es la excepción. En el Capítulo 8 se mencionó que existen API's desarrolladas por terceros que pueden ser utilizadas por otros programadores. En el caso de Grails, existe un amplio repositorio de plugins con diversas funcionalidades que aceleran el proceso de desarrollo. En este capítulo se hace uso de estas herramientas.

9.1 Introducción a los plugins de Grails

El repositorio de plugins de Grails cuenta con una amplia variedad de funcionalidades. Cuando se requiere alguna en específico, lo primero que se debe hacer es consultar dicho repositorio para ver si ya existe algún plugin que cubra los requerimientos. La mayoría de las veces alguien más ya implementó la solución y la comparte con la comunidad de Grails mediante un plugin.

Para consultar la lista de plugins disponibles, se ejecuta el siguiente comando:

```
> grails list-plugins
```

Grails comienza la descarga de la información de los plugins:

```
Downloading: http://.../plugins-list.xml
```

Cuando finaliza la descarga, se muestra una lista de los plugins disponibles, de los plugins disponibles en GrailsCore y de los plugins instalados actualmente en el proyecto.

Para instalar un plugin, se escribe lo siguiente:

```
> grails install-plugin [nombrePlugin]
```

Para instalar la versión específica de un plugin, se agrega la versión al comando anterior:

```
> grails install-plugin [nombrePlugin] [version]
```

Para desinstalar un plugin, se ejecuta lo siguiente:

```
> grails uninstall-plugin [nombrePlugin]
```

Las siguientes secciones muestran la instalación y funcionalidad de algunos plugins.

9.2 Creación dinámica de menús

La navegación estándar de una aplicación Grails a través de los diversos catálogos que contiene no es muy amigable. Los vínculos principales se encuentran en la página principal de la aplicación. Para acceder a un catálogo, se tiene que escribir la URL de forma manual o se tiene que regresar a la página principal para obtener el vínculo correspondiente.

Grails cuenta con un plugin denominado *navigation* que facilita el acceso a los catálogos por medio de un menú principal. Para instalarlo, se usa el siguiente comando:

```
> grails install-plugin navigation
```

El siguiente paso es agregar la propiedad estática `navigation = true` en los controladores:

```
1. class ProfesorController {
2.     ...
3.     static navigation = true
4.     ...
5. }
```

Finalmente, se edita el archivo `grails-app/views/layouts/main.gsp` para agregar las siguientes piezas de código:

```
1. <html>
2.     <head><nav:resources/></head>
3.     <body>
4.         <div id="menu">
5.             <nav:render/>
6.         </div>
7.     <g:layoutBody/>
8. </body>
9. </html>
```

Al ejecutar la aplicación, se observa un menú similar al de la Figura 18.



Figura 18: Menú generado por el plugin "navigation".

Al hacer clic en alguna de las opciones, el menú conduce al catálogo correspondiente.

El menú puede ser editado de acuerdo a las necesidades del usuario. Ejemplo de ello es en el controlador *ProfesorController*:

```
1. static navigation = [  
2.     group: 'tabs',  
3.     order: 1,  
4.     title: 'Lista de Profesores',  
5.     action: 'list'  
6. ]
```

Se hace el mismo cambio para *LibroController* y *PrestamoController*, cambiando el atributo *order* a 2 y 3, respectivamente. Asimismo, se modifica el archivo *grails-app/views/layouts/main.gsp* de la siguiente forma:

```
1. <html>  
2.     <head><nav:resources/></head>  
3.     <body>  
4.         <div id="menu">  
5.             <nav:render group="tabs"/>  
6.         </div>  
7.         <g:layoutBody/>  
8.     </body>  
</html>
```

Hechos estos cambios, el menú luce como en la Figura 19.



Figura 19: Aspecto del menú al hacer modificaciones.

El plugin ofrece otras funcionalidades, como el uso de submenús y estilos CSS. Se recomienda consultar [GRA04] para un tratamiento

detallado del mismo.

9.3 Exportación de datos a diversos formatos

Muchas aplicaciones web utilizan tablas para mostrar los registros de sus bases de datos. En ocasiones, estas tablas requieren ser trasladadas a diversos formatos, especialmente PDF, hojas de cálculo y texto plano. Existen API's como *iText* [ITE01] y *Apache POI* [APA01] que proporcionan las herramientas necesarias para la generación mediante código de dichos documentos. Grails, por su parte, ofrece el plugin *export*, el cual genera diversos tipos de documentos con solo un clic.

Para instalarlo, se usa el siguiente comando:

```
> grails install-plugin export
```

El siguiente paso es modificar el archivo *grails-app/conf/Config.groovy* para agregar los MimeTypes soportados por el plugin:

```
1. grails.mime.types = [ html: ['text/html', 'application/xhtml+xml'],
2.     xml: ['text/xml', 'application/xml'],
3.     text: 'text-plain',
4.     js: 'text/javascript',
5.     rss: 'application/rss+xml',
6.     atom: 'application/atom+xml',
7.     css: 'text/css',
8.     csv: 'text/csv',
9.     pdf: 'application/pdf',
10.    rtf: 'application/rtf',
11.    excel: 'application/vnd.ms-excel',
12.    ods: 'application/vnd.oasis.opendocument.spreadsheet',
```

```

13.     all: '*/*',
14.     json: ['application/json','text/json'],
15.     form: 'application/x-www-form-urlencoded',
16.     multipartForm: 'multipart/form-data'
17. ]

```

Posteriormente se agrega el siguiente taglib en el header del GSP deseado:

```
<export:resource />
```

Asimismo, se agrega la siguiente línea de código justo debajo de la paginación de las tablas en los archivos *grails-app/views/./list.gsp*:

```
<export:formats formats="['csv', 'excel', 'ods', 'pdf', 'rtf', 'xml']" />
```

Finalmente, se agrega el siguiente código al controlador en el closure *list*:

```

1. package mx.edu.uaeh.promep
2.
3. import org.codehaus.groovy.grails.commons.ConfigurationHolder
4.
5. class ProfesorController {
6.
7.     def exportService
8.     ...
9.     def list = {
10.         params.max = Math.min(params.max ? params.int('max') : 10, 100)
11.         if(params?.format && params.format != "html"){
12.             response.contentType =
13.                 ConfigurationHolder.config.grails.mime.types[
14.                     params.format ]
15.             response.setHeader("Content-disposition",
16.                 "attachment; filename=personas.${params.extension}")
17.             exportService.export(params.format, response.outputStream,
18.                 Profesor.list(params), [:], [:])

```

```

19.         }
20.
21.         [profesorInstanceList: Profesor.list(params),
22.          ProfesorInstanceTotal: Profesor.count()]
23.     }
24. ...

```

Al ejecutar la aplicacion, se observa una barra en la parte inferior de la tabla similar a la de la Figura 20.



Figura 20: Barra de exportación de datos.

Al hacer clic en alguna de las opciones mostradas, el navegador entrega un archivo con el formato correspondiente que contiene los registros mostrados en la tabla superior. Por defecto, se renderizan todos los atributos en orden alfabético.

Convocatoria	Participante	Aprobado
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	ACOSTA HERNÁNDEZ JUAN ALBERTO	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	AHUMADA MEDINA ALBINO	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	ANTON DE LA CONCHA JOSE LUIS	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	ARENAS FLORES ALBERTO	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	BALLESTEROS GARCIA VICTOR	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	BEZIES CRUZ PATRICIA	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	BUENO SORIA JUAN MANUEL	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	CARO CANALES IRMA	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	CRUZ ÁVILA DANIEL MARIO	SI
1997 - RECONOCIMIENTO A PERFIL DESEABLE Y APOYO	QUEVAS RAMÍREZ LOURDES TERESA	SI

Figura 21: Reporte generado por el plugin export.

En la aplicación para la dirección de PROMEP-UAEH, se requiere que los reportes generados contengan todos los registros de la base de datos y que puedan ser enviados a un archivo PDF, XLS o CSV. La Figura 21 muestra un reporte obtenido de la aplicación en formato PDF. Se aprecia que el documento contiene 52 páginas, lo cual confirma la entrega de todos los registros existentes.

Las tablas generadas pueden ser estilizadas de acuerdo a las necesidades del usuario. Para un mejor tratamiento del plugin *export*, se sugiere consultar [GRA03].

9.4 Uso de tablas dinámicas con JQGrid

El uso de JavaScript y Ajax es fundamental para Web 2.0. La presentación y dinamismo de una página web hace que su contenido sea interesante y llamativo. Grails facilita la creación de tablas con jQuery mediante el plugin JQGrid. Para instalarlo, se utiliza el siguiente comando:

```
> grails install-plugin jqgrid
```

Como ejemplo de uso, se necesita editar el archivo *grails-app/views/libro/list.gsp* sustituyendo todo el código existente por el siguiente:

```
1. <%@ page import="mx.edu.uaeh.promep.Libro" %>
2. <html>
3.     <head>
4.         <meta name="layout" content="main" />
5.         <jq:resources />
6.         <jqui:resources />
7.         <jqgrid:resources />
```

```
8.     <script type="text/javascript">
9.         $(document).ready(function() {
10.            <jqgrid:grid
11.                id="libro"
12.                url="'${createLink(action: 'listJSON')}'"
13.                colNames=" 'ISBN', 'Titulo', 'Autor', 'Edicion'"
14.                colModel="{name:'isbn', editable: false},
15.                    {name:'titulo', editable: true},
16.                    {name:'autor', editable: true},
17.                    {name:'edicion', editable: true}"
18.                sortname=" 'titulo'"
19.                caption=" 'Lista de Libros'"
20.                height="300"
21.                autowidth="true"
22.                scrollOffset="0"
23.                viewrecords="true"
24.                showPager="true"
25.                datatype=" 'json'">
26.            <jqgrid:filterToolbar id=",libroStandard"
27.                searchOnEnter="false" />
28.            <jqgrid:navigation id="libroStandard" add="true"
29.                edit="true" del="true" search="true"
30.                refresh="true" />
31.            <jqgrid:resize id="libroStandard"
32.                resizeOffset="-2" />
33.        </jqgrid:grid>
34.        });
35.    </script>
36. </head>
37. <body>
38.     <jqgrid:wrapper id="libro" />
39. </body>
40. </html>
```

El código es tomado de la documentación oficial del plugin [GRA05]. También es necesario agregar la acción listJSON en el controlador correspondiente:

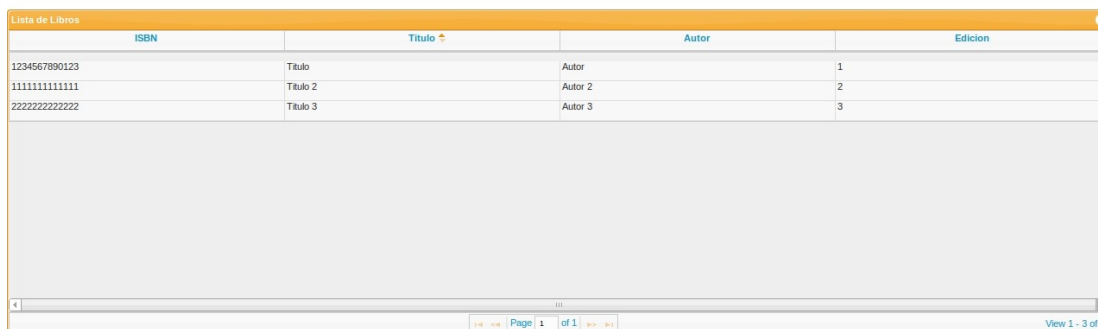
```
1. def listJSON = {
2.     def sortIndex = params.sidx ?: 'titulo'
3.     def sortOrder = params.sord ?: 'asc'
4.     def maxRows = Integer.valueOf(params.rows)
5.     def currentPage = Integer.valueOf(params.page) ?: 1
6.     def rowOffset = currentPage == 1 ? 0 : (currentPage - 1) * maxRows
7.     def libros = Libro.createCriteria().list(max: maxRows,
8.         offset: rowOffset) {
9.         if (params.isbn)
10.            ilike('isbn', "%${params.isbn}%")
11.
12.         if (params.titulo)
13.            ilike('titulo', "%${params.titulo}%")
14.
15.         if (params.autor)
16.            ilike('autor', "%${params.autor}%")
17.
18.         if (params.edicion) {
19.            ilike('edicion', "%${params.edicion}%")
20.        }
21.     order(sortIndex, sortOrder).ignoreCase()
22. }
23.
24. def totalRows = libros.totalCount
25. def numberOfPages = Math.ceil(totalRows / maxRows)
26.
27. def results = libros?.collect {
28.     [
29.         cell: [it.isbn, it.titulo, it.autor, it.edicion],
```

```

30.     ]
31.   }
32.
33.   def jsonData = [rows: results, page: currentPage,
34.                 records: totalRows, total: numberOfPages]
35.   render jsonData as JSON
36. }

```

El código también es tomado de la documentación oficial del plugin. Al ejecutar la aplicación y acceder a la dirección web <http://localhost:8080/BDPromep/libro/list>, se obtiene una respuesta similar a la mostrada en la Figura 22.



ISBN	Título	Autor	Edición
1234567890123	Título	Autor	1
1111111111111	Título 2	Autor 2	2
2222222222222	Título 3	Autor 3	3

Figura 22: Tabla con JQGrid y Ajax.

El plugin proporciona una codificación amigable de JQGrid. Para conocer más a detalle su funcionalidad, se sugiere visitar [JQG01].

9.5 Adición de Seguridad mediante Spring Security

La seguridad es uno de los aspectos más importantes de una aplicación web. La restricción del acceso a diversos recursos de un servidor es una característica indispensable para evitar entradas no autorizadas. Spring cuenta con un módulo de seguridad denominado

Spring Security, el cual cuenta con una adaptación para Grails mediante un plugin denominado *Spring Security Core*.

Para instalarlo, se utiliza el siguiente comando:

```
> grails install-plugin spring-security-core
```

El primer paso es la creación de 3 clases de dominio:

- Clase que representa a los usuarios.
- Clase que representa los roles desempeñados por los usuarios.
- Una clase que relaciona las dos anteriores.

Para ello, el plugin cuenta con un comando:

```
> grails s2-quickstart mx.edu.uaeh.promep User Role
```

Es recomendable explorar el contenido de dichas clases. Una vez creadas, se programa la generación de 2 usuarios al momento del arranque de la aplicación a través del archivo ubicado en *grails-app/conf/Bootstrap.groovy*:

```
1. import mx.edu.uaeh.promep.Role
2. import mx.edu.uaeh.promep.User
3. import mx.edu.uaeh.promep.UserRole
4.
5. class BootStrap {
6.     def springSecurityService
7.     def init = { servletContext ->
8.         if ( User.findByUsername( 'admin' ) &&
9.             User.findByUsername( 'user' ) ) return
10.         def adminRole = new Role(authority: 'ROLE_ADMIN')
11.             .save(flush: true)
12.         def userRole = new Role(authority: 'ROLE_USER')
13.             .save(flush: true)
```

```

14.     String password =
15.         springSecurityService.encodePassword('password')
16.     def adminUser = new User(username: 'admin', enabled: true,
17.         password: password)
18.     adminUser.save(flush: true)
19.     def normalUser = new User(username: 'user', enabled: true,
20.         password: password)
21.     normalUser.save(flush: true)
22.     UserRole.create(adminUser, adminRole, true)
23.     UserRole.create(normalUser, userRole, true)
24.     assert User.count() == 2
25.     assert Role.count() == 2
26.     assert UserRole.count() == 2
27. }
28. }

```

Finalmente, se agregan las restricciones de acceso a las URL's de la aplicación en el archivo *grails-app/config/Config.groovy*:

```

1. import grails.plugins.springsecurity.SecurityConfigType
2. ...
3. grails.plugins.springsecurity.securityConfigType=
4.     SecurityConfigType.InterceptUrlMap
5. grails.plugins.springsecurity.interceptUrlMap = [
6.     '/profesor/**': ['ROLE_ADMIN'],
7.     '/grupo/**':   ['ROLE_ADMIN'],
8.     '/plaza/**':  ['ROLE_ADMIN', 'ROLE_USER'],
9.     '/login/**':  ['IS_AUTHENTICATED_ANONYMOUSLY'],
10.    '/logout/**':  ['IS_AUTHENTICATED_ANONYMOUSLY'],
11.    '/**':         ['ROLE_ADMIN', 'ROLE_USER']
12. ]

```

Al ejecutar la aplicación, se solicita al usuario introducir sus credenciales (Figura 23).



The image shows a login form with a light blue background and dashed borders. At the top, it says "Please Login..". Below this, there are three sections separated by dashed lines: "Login ID" with a text input field, "Password" with a text input field, and "Remember me" with a checkbox. At the bottom, there is a "Login" button.

Figura 23: Solicitud de credenciales con Spring Security.

Se recomienda utilizar los usuarios creados para observar el comportamiento de la aplicación. Para cerrar sesión, se debe acceder a la URL <http://localhost:8080/BDPromep/logout>. Si se inicia sesión como usuario “user” y se intenta acceder a la lista de personas o libros, se obtiene un mensaje de acceso no autorizado.

El acceso a la aplicación Grails de la dirección de PROMEP-UAEH debe restringirse por medio de claves de usuario y el rol que desempeñan dentro de la misma dirección. Esto con el fin de especializar las funciones de cada uno de los usuarios.

En esta aplicación, se utiliza el plugin de seguridad *spring-security-core* mencionado en la sección 9.5. La lista de los diversos roles de los usuarios es la siguiente:

- Administrador.
- Director.

- Trabajador.
- Recepcionista.
- Profesor.
- Área.
- DES (Dependencias de Educación Superior)
- CA (Cuerpo Académico)
- Directivo.

La pantalla de inicio de sesión de la Figura 23 se modifica para ser similar a la mostrada en la Figura 24.

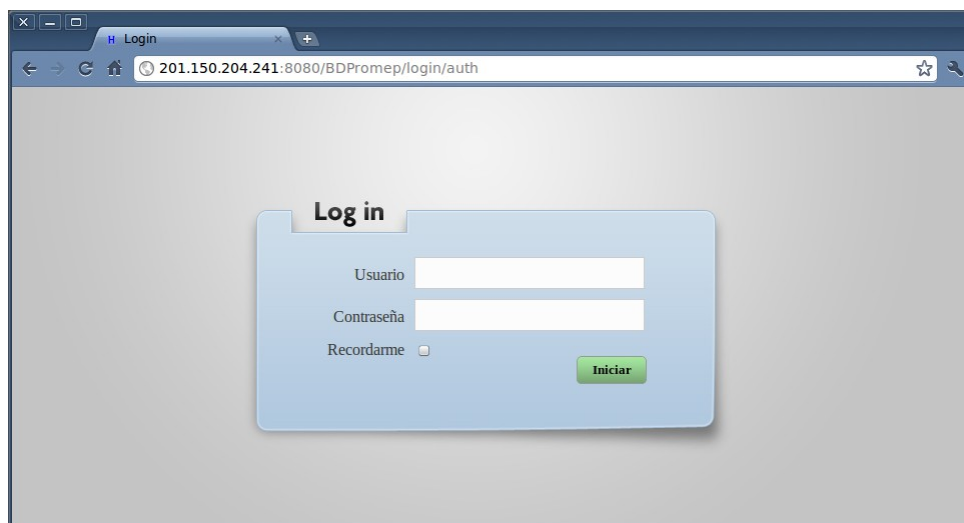


Figura 24: Página de inicio de sesión de la aplicación Grails de PROMEP-UAEH.

Dependiendo de los roles asignados, se despliega la información correspondiente. Por ejemplo, la Figura 25 ilustra el menú de un

usuario tipo DES.



Figura 25: Pantalla principal para un usuario tipo DES.

Como puede observarse, se restringen las funcionalidades del menú a las que el usuario tiene acceso. Esto se logra mediante los taglibs proporcionados por el plugin.

El plugin para Spring Security de Grails tiene mucha funcionalidad que podría abarcarse en un libro completo. Para un tratamiento más profundo acerca del mismo, se recomienda consultar [GRA06].

9.6 Ingeniería inversa de bases de datos

Es común que al hacer una aplicación web ya exista una base de datos. Esto ocasiona que el desarrollador haga un mapeo manual de las clases de dominio y sus relaciones entre si tomando como base las

tablas existentes. Existe un plugin en Grails llamado *db reverse engineering* que automatiza esta tarea.

Tal como se mencionaba en la sección 2.3.1, la dirección de PROMEP-UAEH ya cuenta con una base de datos donde se maneja la información propia del área. Para las necesidades mencionadas en el Capítulo 2, es necesario retomar la base de datos existente para permitir la portabilidad de la información.

La instalación del plugin se realiza de la siguiente forma:

```
> grails install-plugin db-reverse-engineer
```

Posteriormente se agrega la siguiente propiedad al final del archivo *grails-app/conf/Config.groovy*

```
grails.plugin.reveng.packageName = 'mx.edu.uaeh.promep'
```

Por defecto, el plugin lee todas las tablas y genera las clases correspondientes. Como no se necesitan todas las tablas, es necesario indicar las tablas requeridas. Esto se hace agregando la propiedad *grails.plugin.reveng.includeTables* en el archivo de configuración *grails-app/conf/Config.groovy*:

```
grails.plugin.reveng.includeTables = [ 'tabla1', 'tabla2', 'tabla3' ]
```

Finalmente, se ejecuta el siguiente comando:

```
> grails db-reverse-engineer
```

Al analizar la carpeta *grails-app/domain*, se pueden observar las siguientes clases de dominio generadas de forma automática.

Una vez que se realiza la ingeniería inversa con el comando *grails db-reverse-engineer*, se revisan las clases de dominio para verificar si

se deben hacer cambios o modificaciones al código generado por el plugin. En el caso de la aplicación para PROMEP-UAEH, es necesario especificar los nombres de las columnas de los atributos correspondientes objetos en el closure mapping (sección 5.4), para así manejar apropiadamente las llaves foráneas.

Para un tratamiento más profundo acerca del plugin, se sugiere revisar [GRA07].

9.7 Inclusión de otros plugins en el sistema

Para enriquecer la experiencia del usuario en la utilización de la aplicación Grails de PROMEP-UAEH, se incluyeron plugins que van desde el uso de GUI's de JavaScript hasta seguridad en el acceso a la información.

9.7.1 Uso de calendarios con el plugin Calendar

La captura de fechas en diversos formularios tiene ciertos detalles. En ocasiones, no se les dice a los usuarios el formato de fecha que se requiere, lo que puede ocasionar confusión y molestias.

Grails cuenta con un plugin denominado *calendar* que proporciona una interfaz gráfica amigable al usuario para facilitar la captura de fechas. La GUI tiene la forma de un calendario, lo cual facilita al usuario su uso y familiarización.

La Figura 26 muestra el uso de *calendar*.

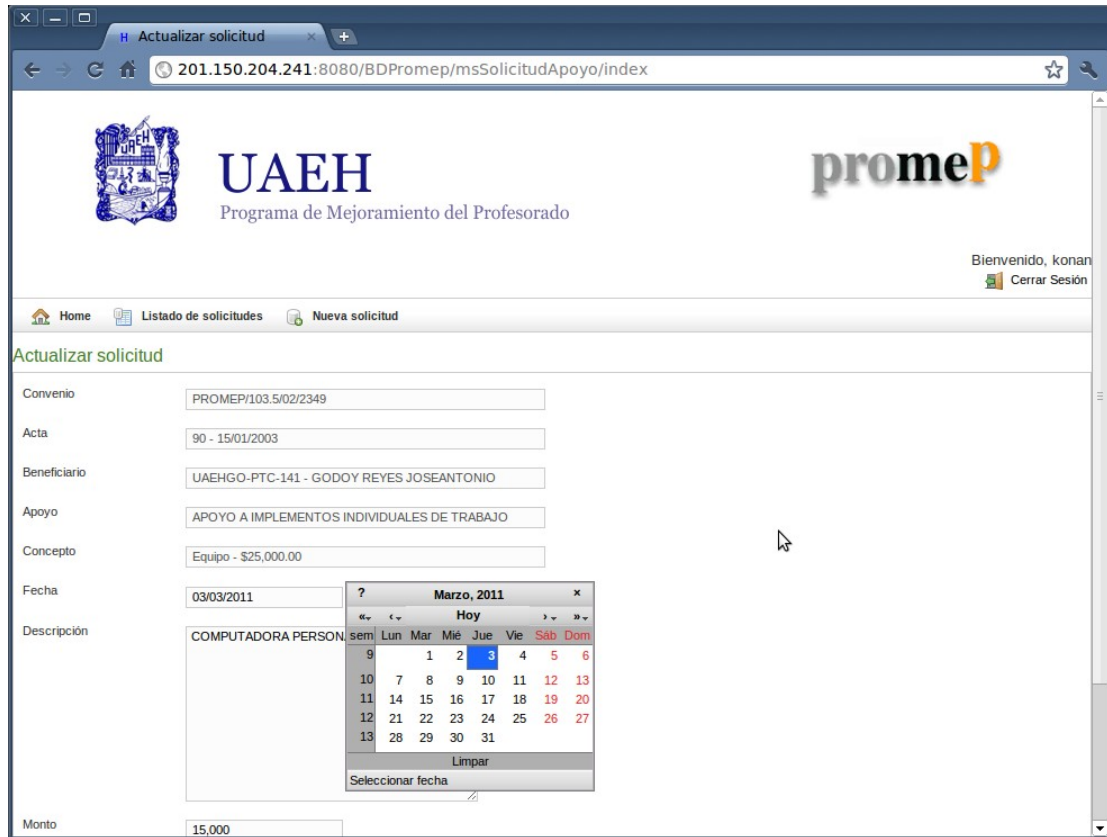


Figura 26: Uso del plugin calendar.

9.7.2 Despliegue de ayuda mediante Help-Balloons

En ocasiones, los usuarios requieren información adicional acerca de la información que se requiere capturar. Grails cuenta con un plugin llamado *help-balloons* que permite la inclusión de “globos informativos” en la aplicación. La Figura 27 ilustra el uso de este plugin.

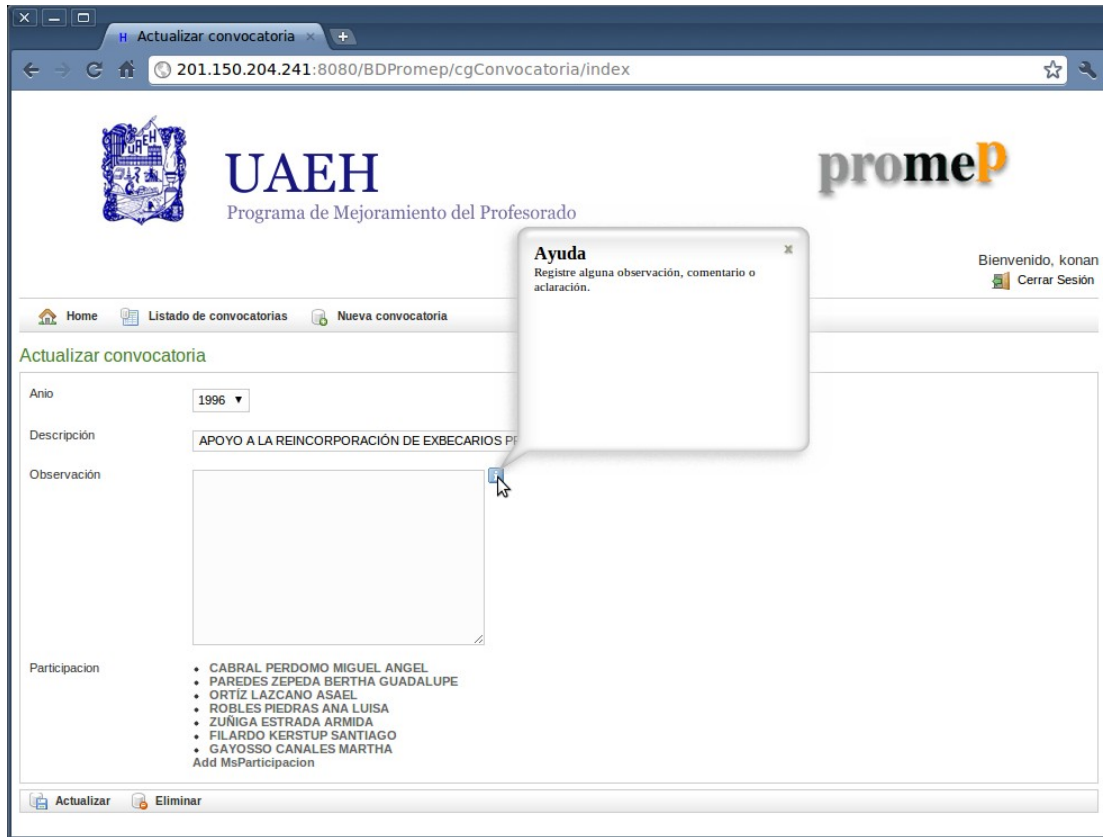


Figura 27: Uso del plugin help-ballons.

9.7.3 Generación dinámica de gráficas con Google Chart y jQuery

Uno de los objetivos de la aplicación Grails para la dirección de PROMEP-UAHEH es la generación de gráficas descriptivas de datos históricos basadas en la información recabada a lo largo del tiempo que ayuden en la toma de decisiones. Para la creación de gráficas, Grails cuenta con un plugin que facilita el uso de la herramienta denominada Google Chart [GOO01], el cual lleva el mismo nombre.

Esta API de tipo REST (Capítulo 8) recibe una URL y devuelve una imagen como respuesta. La URL puede utilizarse para ser embebida en una página web o en cualquier aplicación que lea imágenes por medio del protocolo HTTP.

La Figura 28 ilustra el uso del plugin *google-chart* en conjunto con *jQuery* para el despliegue de una ventana de diálogo que contiene la gráfica solicitada.

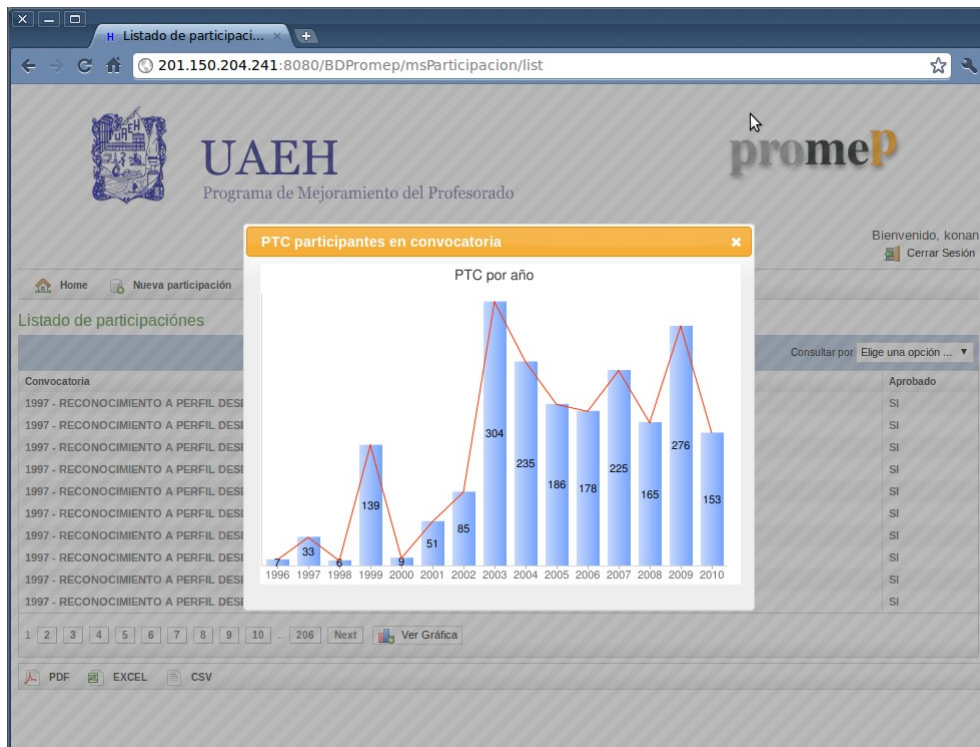


Figura 28: Uso de Google Chart y jQuery para la visualización de gráficas.

Asimismo, la sección de indicadores del tablero de control utiliza el plugin para la generación de diversas gráficas. Esto se muestra en la Figura 29.

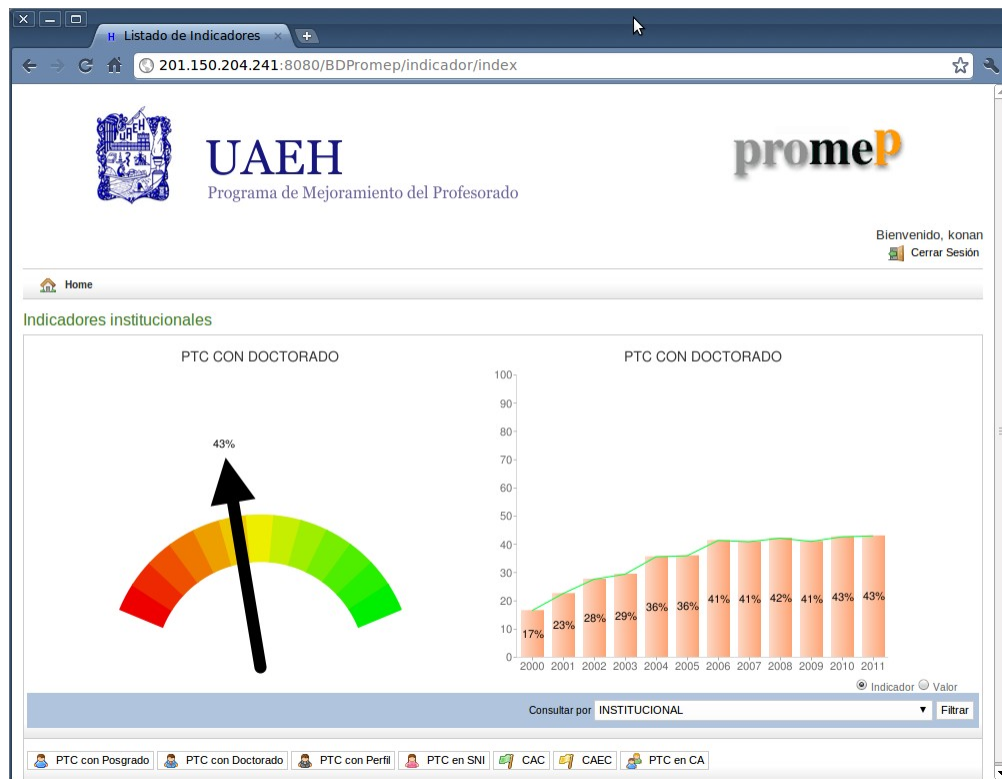


Figura 29: Uso de Google Chart para la generación de gráficas de indicadores.

El tablero de indicadores institucionales se divide en 6 rubros:

- Profesores de Tiempo Completo con Posgrado.
- Profesores de Tiempo Completo con Doctorado.
- Profesores de Tiempo Completo con Perfil.
- Profesores de Tiempo Completo en el Sistema Nacional de Investigadores.
- Cuerpos Académicos Consolidados

- Cuerpos Académicos en Consolidación.
- Profesores de Tiempo Completo en Cuerpos Académicos.

La revisión histórica y visualización gráfica de estos indicadores es de gran ayuda para la toma de decisiones en la dirección de PROMEP-UAEH y demás directivos que requieren de esta información para fortalecer el proceso de gestión y dirección de la UAEH.

9.8 Resumen

El uso del repositorio de plugins del framework acelera aún más este proceso, proporcionando diversas funcionalidades que otros desarrolladores han implementado y compartido con la comunidad de Grails. Con este capítulo se concluye la exploración de las capacidades más sobresalientes de Grails y la exposición de algunas de sus aplicaciones en el sistema de PROMEP-UAEH.

Conclusiones y perspectivas a futuro

A lo largo de este documento se presentó Grails, una herramienta ágil de desarrollo de aplicaciones web que permite entregar resultados en muy poco tiempo. Su facilidad de uso aminora la curva de aprendizaje tanto a los desarrolladores novatos como expertos. A éstos últimos les permite apreciar la necesidad de desarrollar de forma ágil (que no es sinónimo de velocidad) mediante la automatización de tareas repetitivas y de bajo nivel. Asimismo, les permite reutilizar toda la experiencia adquirida en el desarrollo web con otros frameworks, como Spring y Hibernate.

La presentación de un caso práctico implementado en la dirección de PROMEP-UAEH comprueba que el uso de Grails y sus plugins permite desarrollar funcionalidades que de hacerse con otras herramientas llevarían meses e incluso años en realizarse, lo que aumenta el valor del uso del framework para entornos críticos de entrega. En un futuro, se pretende incluir más funcionalidad dinámica y asíncrona mediante Ajax y la migración posterior de la aplicación a la versión 2.0 de Grails, la cual contiene, entre muchas otras características, la inclusión de HTML 5, lo que representa un gran paso en la evolución de los frameworks de desarrollo web. Asimismo, está por liberarse la versión 1.0 de Griffon Framework [GRI01], una herramienta similar a Grails enfocada a la realización de aplicaciones de escritorio desarrollada y mantenida por el ingeniero de software mexicano Andrés Almiray.

El futuro del desarrollo de aplicaciones web, de escritorio y móviles dependerá de la agilidad, creatividad y sobre todo, facilidad de uso de los frameworks desarrollados para este fin. Grails y Griffon han marcado la pauta en una nueva generación de herramientas de desarrollo que prometen un desarrollo eficiente, sencillo y confiable.

Cabe destacar que las versiones preliminares de este documento han sido repartidas y aplicadas en diversos cursos y seminarios, entre ellos el *Congreso Universitario en Tecnologías de Información y Comunicaciones 2011*; asimismo, alumnos de octavo semestre han sido introducidos en el uso de Grails en la materia de Bases de Datos II usando este documento como referencia, teniendo gran aceptación entre los mismos.

Glosario de términos

Ajax: Asynchronous JavaScript and XML, es la tecnología utilizada para hacer peticiones asíncronas a un servidor, lo que permite mayor dinamismo en las páginas web.

API: Application Programming Interface, es un conjunto de herramientas de código probadas y optimizadas que pueden ser reutilizadas para la aceleración de escritura de código.

Bean: En Grails y Spring, es una entidad u objeto que desempeña una tarea determinada e interactúa con otras para hacer funcionar el sistema.

Bytecodes: En Java, los bytecodes son el resultado de la compilación del código fuente y son el lenguaje que la Máquina Virtual de Java puede leer y ejecutar.

Closure: En Groovy, un closure es una pieza de código similar a las clases anónimas de Java pero con capacidades superiores.

Controlador: En el modelo MVC, el controlador es la parte que conecta a la vista (la interfaz gráfica de usuario) con la lógica de negocios del sistema.

DBMS: DataBase Management System, Sistema Gestor de Bases de Datos.

DSL (Domain Specific Language): En Groovy, un DSL es una función que permite extender el lenguaje y facilitar la funcionalidad de diversos plugins.

Framework: En Ingeniería de Software, un framework es un

conjunto de herramientas cuyo objetivo es facilitar y acelerar el desarrollo de código mediante versiones más sencillas de software.

GSP: Groovy Server Pages, la tecnología utilizada por Grails para la generación de código HTML y plantillas web.

HTML: HyperText Markup Language, Lenguaje de Marcado de Hipertexto; es la tecnología utilizada por los navegadores web para la generación y visualización de contenido web.

HTTP: HyperText Transfer Protocol, Protocolo de Transferencia de Hipertexto; es el protocolo utilizado para la transmisión de datos entre computadoras.

JAR: Java Archive, es el formato más utilizado en Java para almacenar API's y código ejecutable.

JavaScript: Lenguaje Script utilizado en HTML para proporcionar contenido dinámico,

JSON: JavaScript Object Notation, es una nomenclatura utilizada para la representación de objetos anidados en Javascript.

Métodos setters y getters: En Java un objeto debe proporcionar acceso a sus propiedades. Esto se logra mediante los métodos get (obtener) y set (establecer).

MVC (Modelo-Vista-Controlador): Arquitectura de diseño utilizada para separar la lógica de negocios de la interfaz gráfica de usuario, comunicándolas a través de un controlador.

Object-Relational Mapping: Mapeo Objeto-Relacional, es el paradigma utilizado para manipular las bases de datos relacionales con el paradigma de la Programación Orientada a Objetos.

Paquete: En Java, un paquete es la ruta en donde se almacenan clases y recursos estáticos.

PDF: Portable Document Format, formato ampliamente utilizado para la generación de documentos de sólo lectura que pueden ser leídos en diversas plataformas

Plugin: En Grails, un plugin es un complemento que agrega funcionalidad a una aplicación web.

REST: Representational State Transfer, Transferencia de Estado Representacional; técnica de arquitectura de software utilizada para la obtención y generación de recursos en un servidor web.

SCRUM: En Ingeniería de Software, SCRUM es una técnica ágil que permite la entrega incremental de software funcional en períodos cortos de tiempo.

Sobrecarga de operadores: En programación, sobrecargar un operador significa proporcionarle mayor funcionalidad. Ejemplo de ello es la sobrecarga del operador "+", el cual sirve tanto para la suma de números como para la concatenación de cadenas

SQL: Structured Query Language, Lenguaje de Consultas Estructurado. El lenguaje estándar para la consulta y manipulación de bases de datos relacionales.

Taglib: En Java y Groovy, un taglib es una etiqueta que proporciona funcionalidades propias ambos lenguajes que pueden ser embebidas en código HTML.

URL: Uniform Resource Locator, conjunto de caracteres que identifican un recurso en la web.

WAR: Web Application Archive, es el formato de Java para el empaquetado de aplicaciones web.

Wrapper: En Java, un wrapper es una clase envolvente de los datos primitivos, de tal forma que pueden ser utilizados como objetos.

XML: Extensible Markup Language, formato ampliamente utilizado para la configuración de aplicaciones y para la transferencia de datos en la web.

Bibliografía

[AAL01]: Agile Alliance (2011). *Agile Alliance*. Obtenido en febrero de 2011 de <http://www.agilealliance.org/>

[APA01]: The Apache Software Foundation (2011). *The Apache POI Project*. Obtenido en febrero de 2011 de <http://poi.apache.org/>

[APT01]: The Apache Software Foundation (2011). *Apache Tomcat*. Obtenido en febrero de 2011 de <http://tomcat.apache.org/>

[COR07]: Cornejo Velázquez, Eduardo (2007). *Sistema de Soporte a las Decisiones Orientado a Web*. Tesis de Maestría, Universidad Autónoma del Estado de Hidalgo.

[DEI07]: Deitel & Deitel (2007). *Java, How to Program*. (Edición 7). Editorial Prentice Hall.

[DIN04]: Martin Fowler (2004). *The Dependency Injection pattern*. Obtenido en febrero de 2011 de <http://martinfowler.com/articles/injection.html>

[FLX01]: Spring Source (2011). *Spring Flex*. Obtenido en febrero de 2011 de <http://www.springsource.org/spring-flex>

[GLA01]: Oracle (2011). *GlassFish - Open Source Application Server*. Obtenido en febrero de 2011 de <http://glassfish.java.net/>

[GOO01]: Google (2011). *Google Chart*. Obtenido en febrero de 2011 de <http://code.google.com/apis/chart/>

[GRA01]: Spring Source (2011). *Grails: The search is over*. Obtenido en febrero de 2011 de <http://grails.org/>

[GRA02]: Spring Source (2011). *Grails Documentation*. Obtenido en febrero de 2011 de <http://grails.org/doc/latest/>

[GRA03]: SpringSource (2011). *Export Plugin*. Obtenido en febrero de 2011 de <http://www.grails.org/plugin/export>

[GRA04]: SpringSource (2011). *Navigation Plugin*. Obtenido en febrero de 2011 de <http://www.grails.org/Navigation+Plugin>

[GRA05]: SpringSource (2011). *JQGrid Plugin*. Obtenido en febrero de 2011 de <http://grails.org/plugin/jqgrid>

[GRA06]: SpringSource (2011). *Spring Security Plugin*. Obtenido en febrero de 2011 de <http://j.mp/bJ254y>

[GRA07]: SpringSource (2011). *Reverse Engineering Plugin*. Obtenido en febrero de 2011 de <http://grails.org/plugin/db-reverse-engineer>

[GRA09]: Glen Smith, Peter Ledbrook (2009). *Grails in Action*. (Edición 1). Editorial Manning.

[GRI01]: Codehaus Foundation (2011). *The Griffon Framework*. Obtenido en febrero de 2011 de <http://griffon.codehaus.org/>

[GRO01]: Groovy (2011). *Groovy - A dynamic language for the Java platform*. Obtenido en febrero de 2011 de <http://groovy.codehaus.org/>

[HDC01]: JBoss Community (2011). *Hibernate Documentation*. Obtenido en febrero de 2011 de <http://www.hibernate.org/docs>

[HIB01]: JBoss Community (2011). *Hibernate - Relational Persistence for Java*. Obtenido en febrero de 2011 de <http://www.hibernate.org/>

[HIB02]: JBoss Community (2011). *Restrictions documentation*. Obtenido en febrero de 2011 de <http://j.mp/uwUQDY>

[HIB03]: JBoss Community (2011). *HQL: The Hibernate Query Language*. Obtenido en febrero de 2011 de <http://j.mp/sOYPjL>

[HSQ01]: HyperSQL (2011). *HSQldb*. Obtenido en febrero de 2011 de <http://hsqldb.org/>

[IAN05]: Ian Somerville (2005). *Ingeniería de Software*. (Edición 7). Editorial Addison Wesley.

[IEE01]: IEEE (2011). *Standard for Binary Floating-Point Arithmetic*. Obtenido en febrero de 2011 de <http://grouper.ieee.org/groups/754/>

[ITE01]: iText (2011). *iText*. Obtenido en febrero de 2011 de <http://www.itextpdf.com/>

[JAV01]: Oracle (2011). *SimpleDateFormat class*. Obtenido en febrero de

2011 de <http://j.mp/upLhzC>

[JAV02]: Oracle (2011). *The Numbers Classes*. Obtenido en febrero de 2011 de <http://j.mp/tQtyRG>

[JAX01]: Oracle (2011). *JAX-WS Reference Implementation*. Obtenido en febrero de 2011 de <http://jax-ws.java.net/>

[JDB01]: Oracle (2011). *JDBC Overview*. Obtenido en febrero de 2011 de <http://j.mp/vSwg1H>

[JEE10]: Eric Jendrock, Ian Evans y otros (2010). *The Java EE 6 Tutorial: Basic Concepts*. (Edición 4). Editorial Java Series.

[JET01]: Codehaus Foundation (2011). *Jetty WebServer*. Obtenido en febrero de 2011 de <http://jetty.codehaus.org/jetty/>

[JGO01]: Wikipedia (2011). *James Gosling*. Obtenido en febrero de 2011 de http://en.wikipedia.org/wiki/James_Gosling

[JMA01]: Oracle (2011). *JavaMail*. Obtenido en febrero de 2011 de <http://j.mp/uDoYIX>

[JMS01]: Oracle (2011). *Java Messaging Service*. Obtenido en febrero de 2011 de <http://j.mp/rzYfBO>

[JQG01]: jQuery (2011). *JQGrid Documentation*. Obtenido en febrero de 2011 de <http://www.trirand.com/blog/>

[JSP01]: Oracle (2011). *JavaServer Pages Technology*. Obtenido en febrero de 2011 de <http://j.mp/uR61dU>

[KAP99]: Robert S. Kaplan, David P. Norton (1999). *El cuadro de mando integral*. (Edición 3). Editorial Gestión 2000.

[KGP05]: Andrew Davison (2005). *Killer Game Programming in Java*. (Edición 1). Editorial O'Reilly. Sección "Java Is Too Slow for Games Programming".

[MSQ01]: Oracle (2011). *MySQL*. Obtenido en febrero de 2011 de <http://www.mysql.com/>

[MSS01]: Microsoft Inc. (2011). *SQL Server*. Obtenido en febrero de 2011 de

<http://www.microsoft.com/sqlserver/>

[MVC01]: Wikipedia (2011). *Model-view-controller*. Obtenido en febrero de 2011 de <http://j.mp/oR1wwT>

[OAU01]: The OAuth Community (2011). *OAuth*. Obtenido en febrero de 2011 de <http://oauth.net/>

[PRE01]: Roger S. Pressman (2005). *Ingeniería del Software: Un Enfoque Práctico*. (Edición 6). Editorial Mc Graw Hill. Página 20.

[PRE02]: Roger S. Pressman (2005). *Ingeniería del Software: Un Enfoque Práctico*. (Edición 6). Editorial Mc Graw Hill. Página 92.

[PRE03]: Roger S. Pressman (2005). *Ingeniería del Software: Un Enfoque Práctico*. (Edición 6). Editorial Mc Graw Hill. Página 52.

[PRE04]: Roger S. Pressman (2005). *Ingeniería del Software: Un Enfoque Práctico*. (Edición 6). Editorial Mc Graw Hill. Página 50.

[PRE05]: Roger S. Pressman (2005). *Ingeniería del Software: Un Enfoque Práctico*. (Edición 6). Editorial Mc Graw Hill. Página 95.

[PRO01]: Secretaría de Educación Pública (2011). *PROMEP*. Obtenido en febrero de 2011 de <http://promep.sep.gob.mx/presentacion.html>

[PYT01]: Spring Source (2011). *Spring Python*. Obtenido en febrero de 2011 de <http://j.mp/tflxTn>

[RMI01]: Oracle (2011). *Remote Method Invocation*. Obtenido en febrero de 2011 de <http://j.mp/uKBAJx>

[SCR01]: SCRUM (2011). *SCRUM: Training, Assessments, Certifications*. Obtenido en febrero de 2011 de <http://www.scrum.org/>

[SDN01]: Spring Source (2011). *Spring .NET*. Obtenido en febrero de 2011 de <http://www.springframework.net/>

[SMO02]: Spring Source (2011). *Spring Mobile*. Obtenido en febrero de 2011 de <http://www.springsource.org/spring-mobile>

[SPR01]: Spring Source (2011). *Spring Framework*. Obtenido en febrero de 2011 de <http://www.springsource.com/>

[SPR02]: Rod Johnson (2002). *Expert One-on-One J2EE Design and Development*. (Edición 1). Editorial Wrox.

[TSS04]: Dion Almaer (2004). *The Java EE 6 Tutorial: Basic Concepts*. Obtenido en febrero de 2011 de <http://bit.ly/hBofG9>

[TWI01]: Twitter (2011). *Twitter API Documentation*. Obtenido en febrero de 2011 de <https://dev.twitter.com/docs>

[W3C01]: W3Schools (2011). *HTML <input> Tag*. Obtenido en febrero de 2011 de http://www.w3schools.com/TAGS/tag_input.asp

[WEB01]: IBM (2011). *WebSphere*. Obtenido en febrero de 2011 de <http://www-01.ibm.com/software/websphere/>

[WIK01]: Wikipedia (2011). *Representational State Transfer*. Obtenido en febrero de 2011 de <http://j.mp/xmFnv>

[WIK02]: Wikipedia (2011). *Tablero de Control*. Obtenido en febrero de 2011 de http://es.wikipedia.org/wiki/Tablero_de_control

[WLO01]: Oracle (2011). *Oracle WebLogic Server*. Obtenido en febrero de 2011 de <http://j.mp/thbVhH>

Anexo A: Instalación de Grails

Este anexo tiene por objetivo explicar el procedimiento paso a paso de la instalación de Grails.

Prerrequisitos

Antes de comenzar a utilizar Grails, se requiere tener instalado el Kit de Desarrollo de Software de Java, mejor conocido como *Java SDK* y establecer una variable de entorno llamada `JAVA_HOME` que apunte a la instalación del mismo. La versión mínima requerida del SDK depende de la versión de Grails a utilizar:

- Java SDK 1.4+ para Grails 1.0.x y 1.1.x.
- Java SDK 1.5+ para Grails 1.2 o superior.

El SDK se puede descargar en <http://j.mp/cyvkd3>.

Lista de pasos a seguir

- Descargar la última versión de Grails.
- Extraer los archivos en una ruta apropiada, típicamente `C:\grails` en Windows o `~/grails` en Unix.
- Crear una variable de entorno llamada `GRAILS_HOME` que apunte a la ruta donde se extrajo el archivo ZIP de Grails.
- Si no se ha creado la variable de entorno `JAVA_HOME`, es momento de hacerlo. Esta variable debe apuntar a la ruta donde se encuentra instalado el SDK de Java.

- Anexar una referencia a la carpeta “bin” ubicada dentro de la instalación de Grails a la variable PATH (%GRAILS_HOME%\bin en Windows o \$GRAILS_HOME/bin en Unix). En entornos Windows, es importante que la variable PATH y la variable GRAILS_HOME estén en el mismo nivel, por ejemplo “Variables de Sistema”.
- Abrir la línea de comandos y escribir el comando “grails” para comprobar que la instalación ha sido exitosa.
- En sistemas Unix, si se obtiene un mensaje de error, ejecutar el comando “chmod +x grails” dentro de la carpeta “bin”.

**Anexo B: Diagrama de Entidades del Sistema
(CSI) de PROMEP-UAEH**