7. Anexos

7.1. Introducción a Apache Tomcat

Tomcat, también llamado Jakarta Tomcat o Apache Tomcat, funciona como un contenedor servlet desarrollado bajo el proyecto Jakarta en la Apache Software Foundation (ASF). Tomcat implementa las especificaciones de los servlets y de JavaServer Pages (JSP) de Sun Microsystems. Tomcat puede funcionar como servidor web por sí mismo.

Como contenedor de Servlet, Tomcat es un componente clave de un conjunto de estándares que recibe el nombre genérico de plataforma J2EE (Java 2 Enterprise Edition). El estándar J2EE define un grupo de API (Interfaz de Programación de Aplicaciones) basadas en Java dirigido a la creación de aplicaciones Web para empresas. J2EE se basa en J2SE (Java 2 Standard Edition), que incluye los binarios de Java (como por ejemplo JVM y el compilador de código de bytes), así como bibliotectas de código básicas de Java. J2EE depende de J2SE para funcionar.

El término API se utiliza por programadores de software para describir los servicios que ofrece un proveedor a las aplicaciones (como por ejemplo, un sistema operativo). En el mundo de Java, este término se utiliza para describir muchos de los servicios que la Máquina virtual de Java (JVM) y sus bibliotecas de código ofrecen a los programas de Java (dicho de otra forma, es el conjunto de funciones y métodos que ofrece cierta biblioteca).

En definitiva, Tomcat es un contenedor servlet compatible con J2EE y es la implementación de referencia oficial de las API Java Servlet y JavaServer Pages.

Profundizaremos en los conceptos "servlets", "contenedor de servlets" , "JSP" y "aplicación web".

7.2. Arquitectura de Tomcat

La arquitectura interna de Tomcat indica claramente la forma en la que debe administrarse. Cada sección de la misma está asociada a una función del servidor.

Analizaremos la arquitectura de Tomcat, incluyendo las siguientes funciones [3]:

- Conectores.
- Motores.
- Reinos.
- Válvulas.

- Registradores.
- Host.
- Contextos.

Tomcat 5 está formado por una jerarquía de componentes anidada. Algunos de estos componentes se denominan de nivel superior ya que se encuentran en la cúspide de la jerarquía con una rígida relación entre sí. Los contenedores son componentes capaces de almacenar otros componentes diferentes. Los componentes almacenados en contenedores pero que no pueden almacenar otros componentes se denominan componentes anidados. En la figura siguiente se ilustra la estructura de una configuración típica de Tomcat 5.

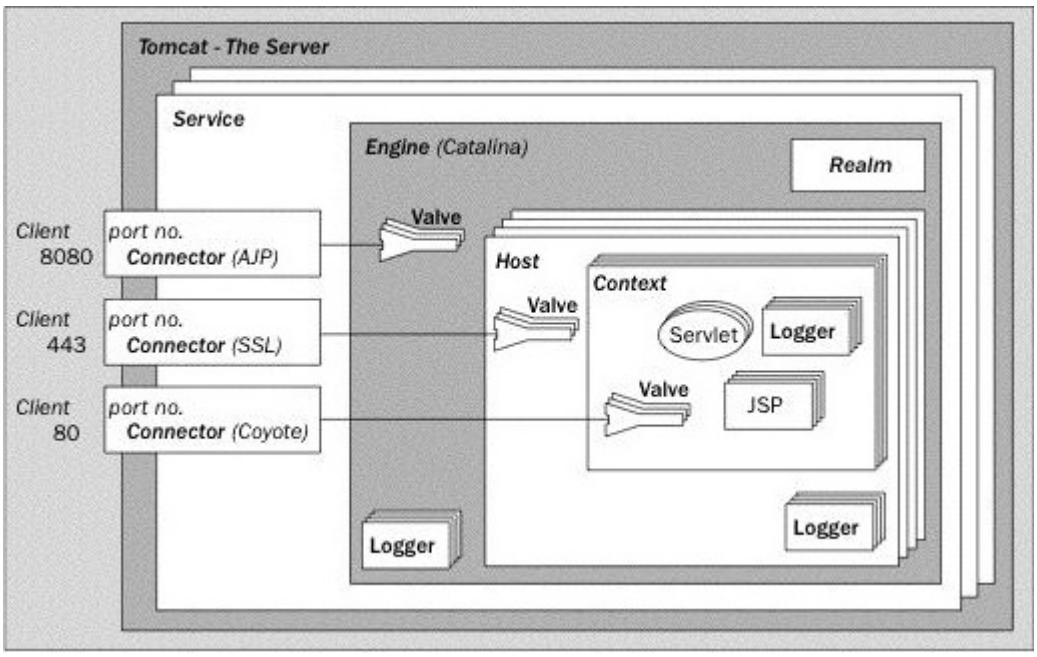

Ilustración 52: Diagrama de bloques de la arquitectura de Apache Tomcat, extraída de [3]

Este diagrama representa la topología más completa de un servidor. Sin embargo, algunos de estos objetos se pueden eliminar sin que ello afecte al rendimiento del servidor. En concreto, el Motor y el Host son innecesarios si un servidor Web externo se encarga de las tareas de resolución de solicitudes a aplicaciones Web.

En este caso, los componentes que se pueden almacenar varias veces son Registradores, Válvula, Host y Contexto. Los Conectores se representan independientemente para ilustrar un aspecto que analizaremos en un apartado posterior.

- El servidor.

El servidor es el propio Tomcat, una instancia de un servidor de aplicaciones Web, y es un componente de nivel superior. Cuenta con un puerto que se utiliza para

cerrar el servidor. Además, se puede establecer en modo de depuración, lo que inicia una versión de la Máquina virtual de Java (JVM) que permite la depuración.
Sólo se puede crear una instancia del servidor dentro de una Máquina virtual de Java.
En un mismo equipo se pueden configurar diferentes servidores en diferentes puertos para aplicaciones que deben reiniciarse de forma independiente. Es decir, si un servidor en el que se ejecute una JVM falla, el resto de aplicaciones estarán protegidas en otra instancia del servidor. Esto se suele realizar en entornos de alojamiento en los que cada cliente tiene una instancia de JVM distinta, para que una aplicación incorrectamente configurada o escrita no cause problemas.

- El servicio.

Un servicio agrupa a un contenedor (normalmente de tipo Motor) con los conectores de dicho contenedor y su componente de nivel superior.
Un motor es un componente de procesamiento de solicitudes que representa al motor de Servlet Catalina. Examina los encabezados HTTP para determinar el host o contexto virtual al que deben pasarse las solicitudes.
Cada servicio representa una agrupación de conectores (componentes que gestionan la conexión entre cliente y servidor) y un contenedor, que acepta solicitudes de los conectores y las procesa para presentarlas al correspondiente host. A cada servicio le corresponde un nombre para que los administradores puedan identificar los mensajes enviados desde cada servicio.
Es decir, el contenedor contiene las aplicaciones Web. Se encarga de aceptar solicitudes, dirigirlas a la aplicación Web concreta y al recurso específico, y de devolver el resultado del procesamiento de la solicitud. Los conectores se sitúan entre el cliente que realiza la solicitud y el contenedor.

- Los conectores.

Los conectores conectan las aplicaciones a los clientes. Constituyen el punto de recepción de las solicitudes por los clientes y cuentan con un puerto en el servidor. El puerto predeterminado para aplicaciones HTTP no seguras es el 8080 para evitar interferir con otros servidores Web que se ejecuten en el puerto estándar (80), aunque se puede cambiar de puerto siempre que esté libre.
En nuestro caso, haremos uso del puerto 7070.
Se pueden configurar varios conectores en un mismo motor o en un componente de nivel de motor, aunque deben tener números de puerto exclusivos.
El conector predeterminado es Coyote, que implementa HTTP 1.1. Otros conectores son Apache JServ Protocol (AJP).
Existen dos conectores estándar incluidos con Tomcat 5:

1. HTTP/1.1: Conecta servicios Web o de navegador al motor Catalina por medio de HTTP 1.1 si el cliente lo admite y, en caso de que sea necesario, puede utilizar HTTP 1.0. Este conector también se puede configurar para admitir conexiones HTTP/SSL seguras.
2. JK2: Se utiliza para conectar servidores Web externos y Tomcat 5 por medio del protocolo AJK 1.3. Utiliza el servidor Web externo para contenidos Web estático, mientras que Tomcat 5 se encarga del procesamiento JSP y Servlet.

- El motor.

El siguiente componente de la arquitectura es el contenedor de nivel superior, un objeto contenedor que no se puede incluir en otro contenedor. Esto significa que no dispone de un contenedor principal. En este nivel los objetos empiezan a agregar componentes secundarios.

Desde un punto de vista estricto, no es necesario que el contenedor sea un motor, simplemente tiene que implementar la interfaz de contenedor. Esta interfaz se encarga de que el objeto que implementa conozca su posición en la jerarquía (que conozca su principal y sus secundarios), que proporcione acceso para el inicio de sesión, que proporcione un reino para la autenticación de usuarios y la autorización basada en funciones, y que tenga acceso a una serie de recursos, incluyendo el administrador de sesiones.

A este nivel el contenedor debe ser un motor, por lo que analizaremos esta función. Un motor es un componente de procesamiento de solicitudes que representa el motor de Servlet Catalina. Examina los encabezados HTTP para determinar el host virtual o el contexto al que deben pasarse las solicitudes.

- El reino.

El reino de un motor gestiona la autenticación y autorización de usuarios. Durante la configuración de una aplicación, el administrador establece las funciones permitidas para cada recurso o grupo de recursos, y el reino se utiliza para hacer cumplir esta política.

Los reinos pueden autenticar archivos de texto, tablas de una base de datos, servidores LDAP y la identidad de red de Windows del usuario.

De forma predeterminada, un usuario debe autenticarse en todas las aplicaciones Web del servidor.

Un reino puede acceder a orígenes de datos externos a Tomcat 5 en los que se almacenen relaciones usuario/contraseña/función. Existen diferentes implementaciones de reinos, que sólo difieren en el origen del que recuperan la información. A continuación se detalla los tipos de reinos estándar de Tomcat 5:

1. Memoria: Utiliza una tabla basada en memoria que contiene las asignaciones entre usuarios, contraseñas y funciones. Habitualmente, se lee en memoria desde un archivo XML durante el inicio del servidor y permanece estática durante el ciclo vital del servidor.
2. UserDatabase: Implementa un reino de memoria actualizable y persistente.
3. JDBC: Utiliza una base de datos relacional para obtener información de autenticación.
4. JNDI: Utiliza JNDI (interfaz de directorios y nombres de Java) para acceder a los datos del reino. Estos datos se suelen utilizar en un directorio basado en LDAP, aunque también se puede emplear cualquier sistema de autenticación compatible con el protocolo LADP (por ejemplo OpenLAPD, Microsoft o Novell cuentan con controladores compatibles con LAPD).
5. JAAS: Funciona con el Servicio de autenticación y autorización de Java (JAAS) para obtener información de autenticación y autorización para el reino.

- Las válvulas.

Las válvulas son componente que permiten a Tomcat interceptar una solicitud y preprocesarla. Los host, contextos y motores pueden contener válvulas.

Se suelen utilizar para habilitar un solo inicio de sesión para todos los host de un servidor, así como para registrar patrones de solicitud, direcciones IP cliente y patrones de uso de tráfico (tráfico punta, uso del ancho de banda, media de solicitudes por minuto, los recursos más solicitados, etc.). Es lo que se conoce como volcado de solicitudes y una válvula de volcado de solicitudes registra la información de encabezados (el URI de la solicitud, los lenguajes admitidos, la dirección IP de origen, el nombre del host solicitado, et.) y todas las cookies enviadas con la solicitud. El volcado de respuestas registra los encabezados de respuesta y las cookies (si las hay) de un archivo.

Por regla general, las válvulas son componentes reutilizables y, por tanto, se pueden añadir y eliminar de la ruta de solicitud en función de las necesidades. Su inclusión es transparente para las aplicaciones Web, aunque el tiempo de respuesta aumenta si se añade una válvula. Una aplicación que quiera interceptar solicitudes para su procesamiento previo y respuestas para su procesamiento posterior debe utilizar filtros que forman parte de las especificaciones de Servlet. Una válvula puede interceptar una solicitud entre un motor y un contexto o host, entre un host y un contexto, y entre un contexto un recurso de la aplicación Web.

- Los registradores.

Los registradores informan del estado interno de un componente. Se pueden configurar para componentes de nivel superior y para los contenedores situados por

debajo. El comportamiento es heredado, por lo que un registrador configurado en el nivel del motor se asigna a todos los objetos secundarios a menos que se reemplace por éstos. La configuración de registradores a este nivel puede ser una forma muy útil de decidir el comportamiento de registro predeterminado de un servidor.
De esta forma se crea un destino para todos los eventos de registro para los componentes que no estén específicamente configurados para generar sus propios registros.

- El host.

Un host imita la conocida funcionalidad de host virtual de Apache. En Apache, esto permite utilizar varios servidores en un mismo equipo, y distinguirlos por su dirección IP o por su nombre de host. En Tomcat, los host virtuales se distinguen por un nombre de host completo. De esta forma, dos sitios Web, pueden alojarse en un mismo servidor, y las solicitudes de cada uno se dirigen a diferentes grupos de aplicaciones Web.
La configuración de un host incluye la definición de su nombre. La mayoría de los clientes puede enviar la dirección IP del servidor y el nombre de host que utilicen para resolver la dirección IP. El nombre de host se incluye en un cabezado HTTP que un motor inspecciona para determinar qué host debe pasarse la solicitud.
Para acceder a nuestro servidor,
http://mariajoproject.dyndns.org:7070/
, el host será mariaproject.dyndns.org.

- El contexto.

Por último, nos encontramos a la aplicación Web, también conocida como contexto. La configuración de una aplicación Web incluye informar al motor o a los host de la ubicación de la carpeta raíz de la misma. También se puede habilitar la carga dinámica para que todas las clases que se hayan modificado se vuelvan a cargar en la memoria. Sin embargo, esta operación consume muchos recursos y nos es recomendable para casos de implementación.
El contexto también puede incluir páginas de error específicas que permitan al administrador del sistema configurar los mensajes de error para adecuarlos al aspecto operativo y visual de la aplicación, así como diversas funciones (un motor de búsqueda, enlaces de utilidad o un componente de creación de informes que notifique al administrador la presencia de errores en la aplicación).
Por último, mencionar que un contexto también se puede configurar con parámetros de inicialización para la aplicación que representa y para controlar el acceso (restricciones de autenticación y autorización).

En resumen, una aplicación Web se representa por medio del componente contexto. Puede incluir registradores que registren mensajes y válvulas que intercepten y procesen solicitudes y respuestas. Las válvulas realizan la intercepción antes y después de que se procese la solicitud y se genere la respuesta. Un contexto reside dentro de un host que representa un host virtual (un alias asignado a la IP actual) y varios contextos pueden compartir el mismo host. El componente contexto puede definir válvulas y registradores. El host se encuentra en el motor, que resuelve solicitudes a los host virtuales. También puede definir válvulas y registradores. Por último, el motor se encuentra en un servicio, que agrupa el motor con los conectores que lo conectan a los clientes. La totalidad del árbol de objetos reside en el componente servidor, es decir, Tomcat.

7.3. Descriptor de implementación de Project.net (archivo web.xml)

Un descriptor de implementación es un archivo XML que contiene información de configuración que la aplicación Web utiliza para ejecutarse en el motor Servlet. Es similar al explicado ya en el apartado anterior. En el anexo se entra en más detalle en este fichero de configuración [15].

```
<!DOCTYPE web-app (View Source for full doctype...)>

<web-app>
   <display-name>Project.net</display-name>

   <session-config>
         <session-timeout>240</session-timeout>
   </session-config>
  <context-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/springapp-servlet.xml</param-value>
   </context-param>

    <context-param>
        <param-name>tapestry.app-package</param-name>
         <param-value>net.project.view</param-value>
   </context-param>
   <filter>
         <f ilter-name>tapestryFilter</filter-name>
         <description>Custom filter for Tapestry Framework</description>
         <filter-class>net.project.hibernate.util.CustomTapestryFilter</filter-
class>
   </filter>
```

```
 <filter>
          filter-name>securityFilter</filter-name>
        <display-name>Security Filter</display-name>
        <description>This filter checks security</description>
       <filter-class>net.project.security.SecurityFilter</filter-class>
 </filter>

 <filter>
       <filter-name>sessionAccessFilter</filter-name>
        <display-name>Session Access Filter</display-name>
        <description>This filter places the session in a ThreadLocal to
provide "back door" access to the HTTPSession.</description>
       <filter-class>net.project.security.SessionAccessFilter</filter-class>
 </filter>

 <filter>
        filter-name>CSSDbValues</filter-name>
       <filter-class>net.project.css.CSSDbValues</filter-class>
       <init-param>
           <param-name>prm.custom.topmenu.height</param-name>
           <param-value>62</param-value>
       </init-param>
 </filter>

 <filter>
        <filter-name>CharacterEncodingFilter</filter-name>
        <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
 </filter>
 <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
 </filter-mapping>
 <filter-mapping>
        <filter-name>sessionAccessFilter</filter-name>
        <url-pattern>*.jsp</url-pattern>
</filter-mapping>
 <filter-mapping>
         filter-name>sessionAccessFilter</filter-name>
        <url-pattern>*.htm</url-pattern>
</filter-mapping>
```

```
<filter-mapping>
      <filter-name>sessionAccessFilter</filter-name>
        url-pattern>/servlet/*</url-pattern>
</filter-mapping>
 filter-mapping>
       filter-name>sessionAccessFilter</filter-name>
      <url-pattern>/resource/*</url-pattern>
</filter-mapping>
<filter-mapping>
      <filter-name>sessionAccessFilter</filter-name>
      <url-pattern>/blog/*</url-pattern>
</filter-mapping>
 filter-mapping>
     <filter-name>sessionAccessFilter</filter-name>
    <url-pattern>/wiki/*</url-pattern>
</filter-mapping>
<filter-mapping>
     <filter-name>sessionAccessFilter</filter-name>
     <url-pattern>/pwiki/*</url-pattern>
</filter-mapping>
<filter-mapping>
     <filter-name>sessionAccessFilter</filter-name>
     <url-pattern>/assignments/*</url-pattern>
</filter-mapping>
<f ilter-mapping>
     <filter-name>sessionAccessFilter</filter-name>
     <url-pattern>/personal/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>sessionAccessFilter</filter-name>
    <url-pattern>/project/*</url-pattern>
</filter-mapping>
  filter-mapping>
     <filter-name>sessionAccessFilter</filter-name>
     <url-pattern>/sessionHook/Extend.*</url-pattern>
</filter-mapping>
<filter-mapping>
      <filter-name>sessionAccessFilter</filter-name>
      <url-pattern>/ajax/schedule/WorkplanRetrieve</url-pattern>
</filter-mapping>
<filter-mapping>
      <filter-name>sessionAccessFilter</filter-name>
      <url-pattern>/ajax/schedule/WorkplanAction</url-pattern>
</filter-mapping>
<filter-mapping>
      <filter-name>securityFilter</filter-name>
```

```
        <url-pattern>*.jsp</url-pattern>
</filter-mapping>
 <filter-mapping>
        <filter-name>securityFilter</filter-name>
        <url-pattern>/servlet/ScheduleController/*</url-pattern>
</filter-mapping>
 <filter-mapping>
        <filter-name>securityFilter</filter-name>
        <url-pattern>/resource/*</url-pattern>
</filter-mapping>
<filter-mapping>
        <filter-name>securityFilter</filter-name>
        <url-pattern>/blog/*</url-pattern>
</filter-mapping>
 <filter-mapping>
       <filter-name>securityFilter</filter-name>
       <url-pattern>/wiki/*</url-pattern>
</filter-mapping>
<filter-mapping>
       <filter-name>securityFilter</filter-name>
       <url-pattern>/pwiki/*</url-pattern>
</filter-mapping>
<filter-mapping>
       <filter-name>securityFilter</filter-name>
       <url-pattern>/assignments/*</url-pattern>
</filter-mapping>
 <filter-mapping>
        <filter-name>securityFilter</filter-name>
          url-pattern>/personal/*</url-pattern>
</filter-mapping>
 <filter-mapping>
        <filter-name>securityFilter</filter-name>
        <url-pattern>/project/*</url-pattern>
</filter-mapping>
<filter-mapping>
        <filter-name>CSSDbValues</filter-name>
        <url-pattern>/styles/noframes.css</url-pattern>
</filter-mapping>
 <filter-mapping>
        <filter-name>CSSDbValues</filter-name>
        <url-pattern>/styles/noframesie.css</url-pattern>
</filter-mapping>
  filter-mapping>
         <filter-name>CSSDbValues</filter-name>
         <url-pattern>/styles/iepngfix.htc</url-pattern>
</filter-mapping>
```

```
<filter-mapping>
        <filter-name>tapestryFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
   <listener-class>net.project.hibernate.util.SpringContextListener
    </listener-class>
</listener>
 <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
 <servlet>
    <servlet-name>QuartzInitializer</servlet-name>
     <display-name>Quartz Initializer Servlet</display-name>
     <servlet-class>org.quartz.ee.servlet.QuartzInitializerServlet
     </ servlet-class>
      <load-on-startup>1</load-on-startup>
</servlet>
 <servlet>
      <servlet-name>dwr-invoker</servlet-name>
      <display-name>DWR Servlet</display-name>
      <servlet-class>uk.ltd.getahead.dwr.DWRServlet
      </servlet-class>
      <init-param>
         <param-name>debug</param-name>
         <param-value>false</param-value>
     </init-param>
</servlet>
<servlet>
        <servlet-name>Download</servlet-name>
        <servlet-class>net.project.base.servlet.DownloadServlet
</servlet-class>
</servlet>
<servlet>
         <servlet-name>ViewDocument</servlet-name>
         <servlet-class>net.project.document.servlet.ViewDocumentServlet
 < /servlet-class>
</servlet>

 ……

 <  servlet-mapping>
       <servlet-name>dwr-invoker</servlet-name>
      <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

```
 ……
<welcome-file-list>
       <welcome-file>Login.jsp</welcome-file>
</welcome-file-list>

<error-page>
       <exception-type>net.project.base.PnetException</exception-type>
      <location>/errors.jsp</location>
</error-page>
 <error-page>
        <exception-type>net.project.base.PnetRuntimeException</exception-
 type>
        <location>/errors.jsp</location>
</error-page>

<taglib>
        <taglib-uri>channel</taglib-uri>
        <taglib-location>/WEB-INF/taglibs/channelTags.tld</taglib-location>
</taglib>
<taglib>
       <taglib-uri>display</taglib-uri>
       <taglib-location>/WEB-INF/taglibs/displayTags.tld</taglib-location>
</taglib>
<taglib>
       <taglib-uri>history</taglib-uri>
      <taglib-location>/WEB-INF/taglibs/historyTags.tld</taglib-location>
</taglib>
<taglib>
     <taglib-uri>links</taglib-uri>
     <taglib-location>/WEB-INF/taglibs/linksTags.tld</taglib-location>
</taglib>

…….
 <resource-ref>
     <description>Default data source</description>
     <res-ref-name>jdbc/PnetDB</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
     <res-auth>CONTAINER</res-auth>
</resource-ref>
 <resource-ref>
     <description>Default mail session</description>
     <res-ref-name>mail/PnetSession</res-ref-name>
     <res-type>javax.mail.Session</res-type>
     <res-auth>CONTAINER</res-auth>
</resource-ref>
</web-app>
```

Lo analizaremos elemento a elemento:

- display-name: Especifica un nombre breve de la aplicación Web.
- sesión-config: Define la configuración de sesiones. El elemento <sesión-config> se puede utilizar para establecer el valor de tiempo de espera de una sesión, valor que se calcula mediante patrones de utilización cliente, junto con requisitos de seguridad. Por ejemplo, si a un usuario se le pide que introduzca mucha información, el valor de tiempo de espera de la sesión puede establecerse en un número mayor para evitar que se pierda información. Si el valor es cero o inferior, la sesión nunca caduca y la aplicación debe eliminarla explícitamente.
- context-param: Contiene pares de nombre y valor con los parámetros de inicialización del contexto de Servlet de una aplicación Web.
- filter: Declara un filtro en la aplicación Web. El filtro se asigna a un servlet o a un patrón de URL en el elemento filter-mapping, por medio del valor filter-name como clave de referencia.
- filter-mapping: El filtro se asigna a un Servlet o a un patrón de URL en el elemento filter-mapping, por medio del valor filter-name como referencia.
- Listener: Configura las propiedades del bean escuchador de la aplicación.
- Servlet: El elemento servlet se utiliza para configurar un Servlet o un archivo JSP.
- Servlet-mapping: Elemento que define la asignación entre un Servlet y un patrón de URL. El nombre de Servlet debe coincidir con el nombre definido en el elemento servlet.
- Welcome-file-list: Elemento que contiene una lista ordenada de archivos de bienvenida (por ejemplo, index.html) y que se especifica a través del elemento secundario welcome-file. Este archivo se muestra cuando alguien se desplaza hasta el URL de la aplicación.
- error-page: Se define una excepción de Java o una página de error basada en código HTTP.
- taglib: Declara una biblioteca de etiquetas. Las bibliotecas de etiquetas, son componentes reutilizables de Java que se pueden invocar mediante etiquetas de marcado de página. La definición de biblioteca de etiquetas la especifica el programador de la aplicación y los diseñadores HTML. Sin embargo, la configuración de estos componentes se realiza en un archivo diferente (con la extensión .tld), ya que esta entrada permite asignar un

alias, en forma de URI, a la ubicación de este documento de configuración A la ubicación exacta del archivo, que se proporciona como referencia del directorio raíz de la aplicación Web, se puede hacer referencia por su alias. Esta creación de alias posibilita la independencia de la ubicación (es decir, los archivos de configuración de la biblioteca de etiquetas se pueden cambiar sin tener que modificar las páginas JSP que hacen referencia a los mismos, siempre que las entradas del registro apunten a dichos archivos).

- resource-ref: Declara y administra las conexiones a los recursos, como por ejemplo una referencia a la conexión del recurso de agrupación de objetos (similar a la agrupación de conexiones de bases de datos), para aumentar la eficacia del proceso. También se encarga de referenciar a objetos administrados, permitiendo a la aplicación acceder a la administración del recurso en tiempo de ejecución. Al estudio de este tipo de conexiones dedicaremos el apartado siguiente de la memoria.

7.4. Configuración Conectores HTTP

El nuevo conector HTTP/1.1 Coyote es el conector predeterminado configurado para Tomcat 4.1x y 5.x. Sustituye a los antiguos conectores HTTP/1.0 y HTTP/1.1 de Tomcat 3.x y 4.0. El conector Coyote ofrece compatibilidad inversa y se puede instalar en Tomcat 3.x y 4.0.

Los conectores HTTP son clases de Java que implementan el protocolo HTTP. La clase Connector de Tomcat (por ejemplo, la clase org.apache.coyote.tomcat5.CoyoteConnector de Tomcat 5) se invoca cuando hay una solicitud HTTP en el puerto del conector. El puerto en el que se escucha el conector se especifica en el archivo de configuración /conf/server.xml, en nuestro caso es el 7070. La clase del conector incluye código para analizar la solicitud HTTP y realizar la correspondiente acción: servir contenido estático o pasar la solicitud por el motor de Servlet de Tomcat.

```
<!--
   Define a non-SSL HTTP/1.1 Connector on port 7070
-->
<Connector port="7070" maxHttpHeaderSize="8192" maxThreads="150"
minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
redirectPort="8443" acceptCount="100" connectionTimeout="20000"
disableUploadTimeout="true" />
```

El único atributo obligatorio para la configuración del conector es port, se pueden configurar muchos otros, que se incluyen a continuación:

- acceptCount: Longitud máxima de cola para las solicitudes de conexión entrantes cuando se utilizan todos los procesos de procesamiento de solicitudes posibles. Se rechazarán todas las solicitudes recibidas cuando se llena la cola. Este valor se pasa como parámetro de registro cuando se crea un socket de servidor Tomcat. La longitud de cola predeterminada es 10 y el valor máximo depende del sitemas operativo.
- address: Este atributo especifica la dirección IP a la que se vincula el servidor Tomcat. Si no se especifica el atributo address, Tomcat se vincula a todas las direcciones (si el host tiene varias direcciones IP).
- allowTrace: Habilita el método TRACE HTTP si se establece en true. El valor predeterminado es false.
- bufferSize: Especifica el tamaño (en bytes) del búfer de flujo de entrada creado por este conector. El valor predeterminado es de 2048 bytes.
- compressibleMimeTypes: Lista separada por comas de tipos MIME en los que se puede utilizar compresión HTTP.
- Compression: El conector puede utilizar compresión GZIP HTTP/1.1 para obtener un mejor ancho de banda del servidor. Se puede habilitar por medio del atributo compression. Los valores válidos son off (deshabilita la compresión), on (la habilita), forcé (fuerza la compresión en todos los casos) o un valor numérico que especifique la cantidad mínima de datos necesaria antes de comprimir el resultado. El valor predeterminado del atributo es off.
- connectionLinger: Establece el número de milisegundos que espera este conector (tras aceptar una conexión) para presentar la línea URI de la solicitud. El valor predeterminado es 60000 milisegundos.
- Debug: Este atributo establece el nivel de detalle de los mensajes de registro. Los valores más altos devuelven un mayor nivel de detalle. (El valor máximo de este atributo no está documentado, aunque si se utiliza 4 ó 5 se imprime la mayoría de los mensajes de registro). El valor predeterminado de este atributo es cero, lo que desactiva la depuración. Toda la información de registro y excepciones se redirige automáticamente al componente Registrador. Los componentes Registrador se pueden asociar al motor relacionado, a un host virtual o incluso a un contexto de aplicación determinado.
- disableUploadTimeout: Permite establecer un tiempo de espera diferente para cargas de datos durante la ejecución de un Servlet. El valor predeterminado es false.

- enableLookups: Si se establece en true, todas las llamadas a request.getRemoteHost() (una llamada de API Servlet de J2SE) realizan una búsqueda DNS para devolver el nombre de host del cliente remoto. Cuando se establece en false, se omite la búsqueda DNS y sólo se devuelve la dirección IP. El valor predeterminado es false. Se puede desactivar este atributo por motivos de rendimiento para evitar la sobrecarga de la búsqueda DNS.
- maxKeepAliveRequest: Atributo que controla el comportamiento de las solicitudes HTTP que permiten conexiones persistentes (es decir, múltiples solicitudes enviadas por la misma conexión HTTP). Especifica el número máximo de solicitudes que se pueden dirigir hasta que el servidor cierra la conexión. El valor predeterminado es 100. Si se establece en 1, se deshabilita este comportamiento.
- maxPostSize: Especifica el tamaño máximo en bytes del POST que puede procesar el contenedor. De forma predeterminada es 2MB. Si se establece en 0 o en un valor negativo, se desactiva.
- maxSpareThreads: Atributo que controla el número máximo de procesos sin utilizar permitidos antes de que Tomcat empiece a detener los procesos sin utilizar. El valor predeterminado es 50.
- minSpareThreads: Especifica el número mínimo de procesos que se inician al inicializar el conector. El valor predeterminado es 4.
- maxThreads: Especifica el número máximo de procesos que se crean para que este conector procese solicitudes. A su vez. Especifica el número máximo de solicitudes concurrentes que puede procesar el conector. El valor predeterminado es de 200 procesos.
- noCompressionUserAgentes: Lista separada por comas que coincide con el valor UserAgente HTTP de los navegadores Web que tienen compatibilidad incompleta con compresión HTTP/1.1. Se pueden utilizar expresiones regulares.
- Port: Especifica el número de puerto TCP en el que este conector creará un socket de servidor y esperará las conexiones entrantes. Sólo se puede vincular una aplicación de servidor a una determinada combinación de número de puerto y dirección IP.
- protocol: Especifica el protocolo HTTP que utilizar y debe establecerse en HTTP/1.1.
- proxyName: Este atributo (junto con el atributo proxyPort) se utiliza al ejecutar Tomcat tras un servidor proxy.
- proxyPort: Como hemos mencionado anteriormente, el atributo proxyPort se utiliza en configuraciones de proxy. Especifica el número de puerto que se devuelve para las llamadas request.getServerName().

- redirectPort: Si el conector sólo admite solicitudes que no sean SSL y se dirige una solicitud de usuario a este conector para un origen SSL, Catalina redirigirá la solicitud al número de puerto redirectPort. La configuración predeterminada de Tomcat especifica 8443 como puerto de redirección, como se indica en el ejemplo de configuración presentado anteriormente. Si se omite, se utiliza 443 de forma predeterminada.
- restrictedUserAgents: Lista separada por comas que coincide con el valor UserAgent HTTP de los navegadores Web que tienen compatibilidad incompleta con HTTP/1.1. Se pueden utilizar expresiones regulares.
- scheme: Atributo que se establece en el nombre del protocolo. El valor especificado se devuelve por la invocación del método request.getScheme(). El valor predeterminado es http y, para conectores SSL, https.
- secure: Atributo que se establece en true para conectores SSL. Este valor se devuelve en las invocaciones del método request.getScheme(). El valor predeterminado es true.
- socketBuffer: Especifica el tamaño, en bytes, del búfer utilizado para almacenar en búfer la salidad del socket. La utilización del búfer de socket mejora el rendimiento. De forma predeterminada, se utiliza un búfer de 9000 bytes y si este parámetro se establece en -1, se desactiva el almacenamiento en búfer.
- tcpNoDelay: Al establecer este atributo en true, activa la opción de socket de red TCP_NO_DELAY, lo que mejora el rendimiento.
- URIEncoding: Especifica la codificación de caracteres utilizada para decodificar byte URI. De forma predeterminada es ISO-8859-1.
- useBodyEncodingForURI: Si se establece en true, este atributo utiliza la codificación de URI especificada en contentType, en lugar del atributo URIEncoding. De forma predeterminada se establece en false.
- xpoweredBy: Si se estable en true, se crea un encabezado X-Powered-By en las respuestas generadas por Servlet devueltas por el conector.

Si quisiéramos configurar Tomcat 5.x para conexiones seguras, tendríamos que añadir los siguientes atributos al conector HTTP:

- algorithm: Especifica el algoritmo de codificación de certificados que utilizar. Es SunX509 de forma predeterminada.
- ciphers: Lista separada por comas de cifras de codificación.
- clientAuth: Si se establece en true (el valor predeterminado es false), la conexión cliente debe presentar un certificado válido. Si se establece en false y

el recurso Web solicitado está protegido por autenticación CLIENT-CERT, ésta tendrá preferencia (es decir, el cliente tendrá que presentar un certificado).

- keystoreFile: Especifica el nombre de ruta del archivo de repositorio de claves. Este archivo contiene las claves públicas y privadas del servidor en forma de certificados. De forma predeterminada, el valor de este atributo es keystore en el directorio principal del usuario, que es específico del sistema operativo.
- keystorePass: Debe establecerse en la contraseña necesaria para acceder a keystoreFile. La predeterminada es changeit.
- keystoreType: Especifica el tipo de archivo repositorio de claves. De forma predeterminada es JKS.
- sslProtocol: Indica qué versión del protocolo SSL utilizar (el valor predeterminado es TLS).

7.5. Cargadores de clase

Tras la especificación Servlet, es obligatorio que Tomcat asigne un cargador de clases exclusivo a cada aplicación Web [3]. Cada vez que se instancia una clase como objeto o se hace referencia a la misma de forma estática, la clase debe cargarse en memoria desde la Máquina virtual de Java (JVM). Por ello, incluso instrucciones tan sencillas como String greeting=”hello” o int maxValu= Intenger.MAX_VALUE utilizan un cargador de clases. Para cargarse, necesitan la clase String y la clase Integer respectivamente.

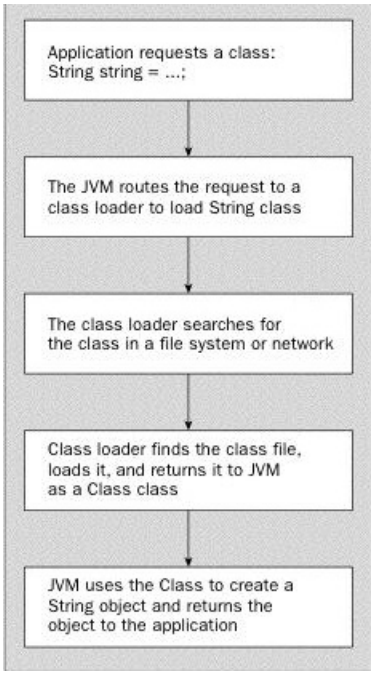

Ilustración 53: Funcionamiento del cargador de clases de Apache Tomcat, extraída de [3]

Desde la aplicación J2SE, la JVM utiliza tres cargadores de clases distintos, que analizamos a continuación:

- Cargador de clases de inicio

Como su nombre indica, este cargador de clases lo utiliza la JVM para cargar las clases de Java necesarias para su funcionamiento. De hecho, se encarga de cargar todas las clases de Java básicas (como java.lang.* y java.io.*).

Como los cargadores de clases se escriben en Java, el cargador de clases de inicio resuelve un problema técnico: como puede la JVM cargar un cargador de clases basado en Java cuando es necesario cargar el propio cargador. Al incluir el cargador de clases de inicio en la JVM se resuelve este problema y diferentes distribuidores de JVM (incluyendo Sun) implementan el cargador de clases de inicio por medio de código nativo.

- Cargador de clases de extensión

En Java 1.2 se presentó el mecanismo de extensiones estándar. Habitualmente, cuando los programadores desean que la JVM buque archivos de clases en determinados puntos, utilizan la variable de entorno CLASSPATH. Sun introdujo el mecanismo de extensiones estándar como método alternativo. Se pueden añadir archivos JAR a un directorio de extensiones estándar y la JVM los buscará de forma automática.

El cargador de clases de extensión se encarga de cargar todas las clases de uno o varios directorios de extensión. Al igual que las rutas del cargador de clases de inicio pueden variar entre diferentes JVM, lo mismo sucede con las rutas del cargador de clases de extensión. En la JVM de Sun, el directorio de extensiones estándar es /jdk/jre/lib/ext.

- Cargador de clases de sistema

Este cargador busca sus clases en los directorios y archivos JAR especificados en la variable de entorno CLASSPATH. También se utiliza para cargar la clase de punto de entrada de una aplicación (es decir, la clase con el método main()) y es el cargador de clases predeterminado para cargar las clases que no abarquen los otros dos cargadores.

Como ya hemos mencionado, J2SE tiene tres cargadores de clases diferentes. Si se crea una instancia de jav.lang.String, el cargador de clases de inicio se encarga de cargarla y si se crea una instancia de una clase de usuario, el responsable es el cargador de clases de sistema. Se preguntará cómo sabe la JVM qué cargador de clases utilizar.

Para ello, utiliza el modelo de delegación. En todas las versiones de Java desde la JDK 1.2, siempre que un cargador de clases recibe una solicitud para cargar una clase, primero pide a su principal que procese la solicitud (es decir, delega la

solicitud a su cargador de clase principal). Antes de que el principal del cargador de clases cargue la clase solicitada, la delega a su principal y así sucesivamente hasta llegar al cargador de clases de inicio. Si el principal carga satisfactoriamente la clase, el objeto de clase resultante se devuelve para que se pueda crear una instancia del mismo (o que se haga referencia al mismo de forma estática). Sólo si el principal del cargador de clases (y su principal, etc.) no puede cargar la clase, el cargador de clases original intenta hacerlo.

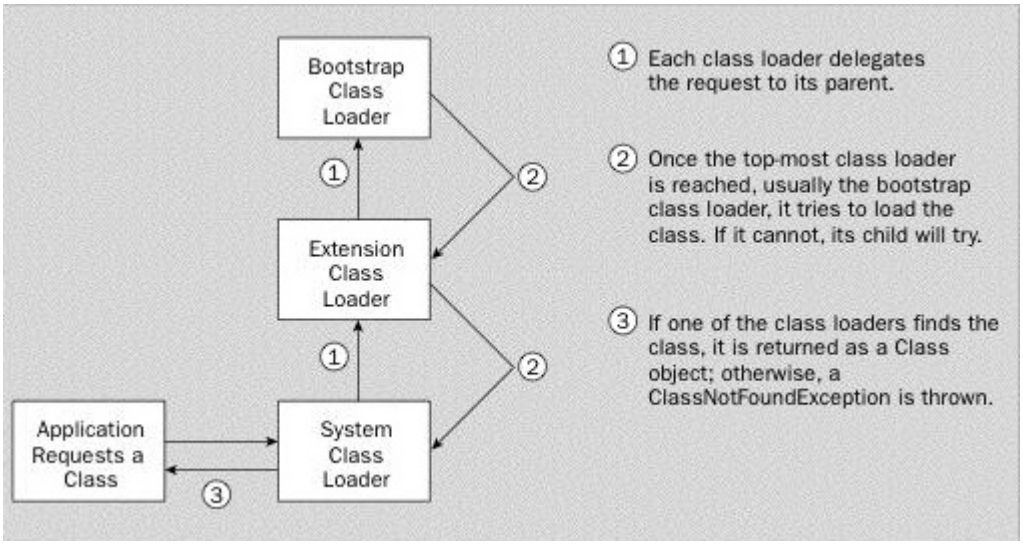

Ilustración 54: Modelo de delegación de clases, extraída de [3]

Por ello, cuando en un programa de Java se hace referencia a una clase, la JVM dirige automáticamente una solicitud al cargador de clases de sistema para que cargue la clase necesaria. Tras ello, el cargador de clases de sistema solicita al cargador de clases de excepción que cargue la clase especificada y éste, a su vez, hace lo mismo con el cargador de clases de inicio. El proceso se detiene en el cargador de clases de inicio, que busca en las bibliotecas de Java (y todo lo demás que tenga que buscar) la clase solicitada.

Si la clase no existe en el dominio del cargador de clases de inicio, el cargador de clases de extensión busca la ubicación d extensiones estándar de la clase. Si no aparece, el cargador de clases de sistema busca las ubicaciones especificadas para la clase por la variable CLASSPATH. Si no se encuentra la clase, se genera una excepción ClassFoundException.

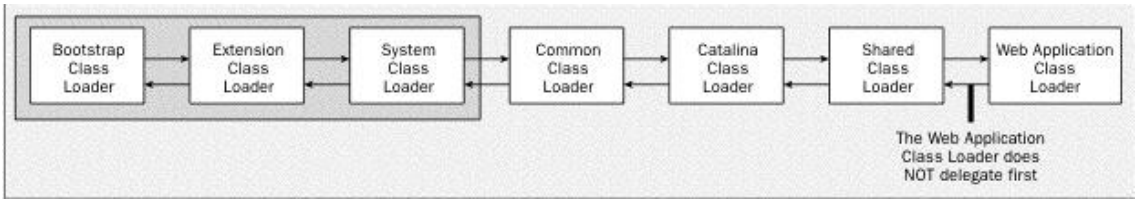

Ilustración 55: Proceso de búsqueda del cargador donde se encuentra una clase determinada, extraída de [3]

Tomcat utiliza el cargador de clases de sistema predeterminado pero de forma diferente al comportamiento predeterminado de la JVM. En el archivo de inicio de Tomcat (startup.bat en Win32 invoca catalina.bat/sh) se borra la variable de entorno CLASSPATH y, en su lugar, Tomcat la apunta a dos de sus archivos: bootstrap.jar y tools.jar.

El cargador de clases de sistema busca en CLASSPATH. Como Tomcat establece la variable CLASSPATH en estos dos archivos, se anula el efecto normal de este cargador de clases. La configuración de la variable CLASSPATH anterior al inicio del servidor se ignora por completo en lo que a Tomcat se refiere. El archivo bootstrap.jar contiene las clases de inicio de Tomcat y tools.jar contiene el compilador javac, que se utiliza para compilar páginas JSP en archivos de clases durante el tiempo de ejecución.

Al iniciarse, Tomcat modifica el mecanismo de sustitución de estándares confirmados para que apunte al directorio /commom/endorsed en lugar de a los que mencionamos anteriormente.

Tomcat incluye una versión del conocido analizador XML Xerces de Apache en este directorio y una versión del API JAXP. Como resultado, este analizador es preferible a cualquier otro incluido con el JRE utilizado para iniciar Tomcat.

El siguiente elemento de la jerarquía es el cargador de clases comunes, responsable de las clases que utiliza Tomcat y disponible de forma pública para todas las aplicaciones Web. Busca estos archivos de clases en dos puntos diferentes /commom/lib y /common/classes. Tomcat incluye una serie de archivos JAR en /common/lib, como por ejemplo una versión de Apache Ant, muchos de los proyectos Jakarta Commons, Jasper (un compilador JSP), así como las clases de API compatibles con Tomcat (Servlet, JSP, JNDI y JMX).

Tomcat puede hacer referencia a todas las clases incluidas en el dominio de este cargador de clases en sus propias aplicaciones Web y también excluirlas de las mismas (de hecho, los programadores no pueden incluir las clases de API Servlet/JSP en sus aplicaciones Web). Sin embargo, no conviene hacer referencia a nada a excepción de las clases de API, por estos motivos:

Depender del cargador de clases comunes para cargar Jakarta Commons, clases Ant, etc., desde este directorio puede anular la portabilidad de la aplicación Web.

No es obligatorio que los contenedores de Servlet proporcionen estas clases. Por ello, cuando se cambia una aplicación Web de Tomcat a uno de estos contenedores de Servlet, pueden surgir problemas.

Las versiones de las bibliotecas que incluye Tomcat pueden ser diferentes a las versiones que espere la aplicación Web. Estos errores resultan díficiles de solucionar.

El cargador de clases Catalina se utiliza para cargar todas las clases específicas de Tomcat, clases que no son visibles para otras aplicaciones. Se almacenan en /server/lib y /server/classes.

El cargador de clases compartidas es similar al de clases comunes, a excepción de que los programadores pueden añadir sus propias clases y archivos JAR al dominio de este cargador. Busca en los siguientes directorios shared/lib y shared/classes. Siempre que los programadores quieran compartir clases generales entre dos o más aplicaciones Web, deben añadirlas a estas ubicaciones.

Project.net tiene su propio cargador de clases, que busca en las siguientes ubicaciones /webapps/WEB-INF/classes y webapps/WEB-INF/lib. Este cargador de clases tiene dos propiedades que lo distinguen. Por un lado, no utiliza el modelo de delegación que los cargadores de clases deben utilizar. En su lugar, primero trata de cargar las clases, antes de delegar la solicitud a otros cargadores (excepto en determinadas circunstancias). Este comportamiento hace que resulte muy sencillo reemplazar clases de los cargadores de clases compartidas y comunes de forma concreta. Cuando se solicita una clase, este cargador de clases comprueba en su caché de clases si la clase ya se ha cargado. Si no la encuentra, delega la solicitud al cargador de clases de sistema, para evitar que las aplicaciones Web, en este caso Project.net, intente crear instancias de las clases que incorpora el JRE. Si el cargador de clases de sistema no encuentra la clase, el cargador de clases de aplicaciones trata de determinar si ésta pertenece a alguno de los siguientes paquetes:javax.*, org.xml.sax.*, org.w3c.dom.*, org.apache.commons.logging.*, org.apache.xerces.*, org.apache.xalan.*. Si pertenece a uno de estos paquetes, el cargador de clases de aplicaciones Web delega la solicitud a su principal, el cargador de clases compartidas. Si la clase sigue sin aparecer, el cargador de clases de aplicaciones Web comprueba si está en su dominio. Si no la encuentra y no ha delegado la solicitud a su principal (es decir, si la clase pertenece a uno de los paquetes anteriores), lo hará en su momento. Por otra parte, cada aplicación Web tiene su propia instancia de este cargador de clases, lo que significa que no pueden ver los archivos de clases de cada una.

7.6. Arquitectura Modelo Vista Controlador de Ruby

Las piezas de la arquitectura Modelo Vista Controlador en Ruby on Rails son las siguientes:

- Modelo.

En las aplicaciones web orientadas a objetos sobre bases de datos, el Modelo consiste en las clases que representan a las tablas de la base de datos.

En Ruby on Rails, las clases del Modelo son gestionadas por ActiveRecord. Por lo general, lo único que tiene que hacer el programador es heredar de la clase ActiveRecord::Base, y el programa averiguará automáticamente qué tabla usar y qué columnas tiene.

Las definiciones de las clases también detallan las relaciones entre clases con setencias de mapeo objeto relacional. Por ejemplo, si la clase Imagen tiene una definición has_many:comentarios, y existe una instancia de Imagen llamada a, entonces a.comentarios devolverá un array con todos los objetos Comentario cuya columna imagen_id (en la tabla comentarios) sea igual a a.id.

Las rutinas de validación de datos (por ejemplo, validates_uniqueness_of:checksum) y las rutinas relacionadas con la actualización (por ejemplo, after_destroy:borrar_archivo, before_update:actualizar_detalles) también se especifican e implementan en la clase del modelo.

- Vista.

En MVC, *Vista* es la lógica de visualización, o cómo se muestran los datos de las clases del *Controlador*. Con frecuencia en las aplicaciones web la vista consiste en una cantidad mínima de código incluido en HTML. Existen en la actualidad muchas maneras de gestionar las vistas. El método que se emplea en Rails por defecto es usar Ruby Embebido (archivos.rhtml, desde la versión 2.x en adelante de RoR archivos.html.erb), que son básicamente fragmentos de código HTML con algo de código en Ruby, siguiendo una sintaxis similar a JSP. También pueden construirse vistas en HTML y XML con Builder o usando el sistema de plantillas Liquid. Es necesario escribir un pequeño fragmento de código en HTML para cada método del controlador que necesita mostrar información al usuario. El "maquetado" o distribución de los elementos de la página se describe separadamente de la acción del controlador y los fragmentos pueden invocarse unos a otros.

- Controlador.

En MVC, las clases del Controlador responden a la interacción del usuario e invocan a la lógica de la aplicación, que a su vez manipula los datos de las clases del Modelo y muestra los resultados usando las Vistas. En las aplicaciones web basadas en

MVC, los métodos del controlador son invocados por el usuario usando el navegador web.

La implementación del Controlador es manejada por el ActionPack de Rails, que contiene la clase ApplicationController. Una aplicación Rails simplemente hereda de esta clase y define las acciones necesarias como métodos, que pueden ser invocados desde la web, por lo general en la forma http://aplicacion/ejemplo/metodo, que invoca a EjemploController#metodo, y presenta los datos usando el archivo de plantilla /app/views/ejemplo/metodo.html.erb, a no ser que el método redirija a algún otro lugar.