Procedimiento para la migración de aplicaciones transaccionales de escritorio a aplicaciones web MVC

Carlos Angarita

Universidad de Pamplona

Facultad de Ingenierías y Arquitectura

Pamplona Colombia

2017

Procedimiento para la migración de aplicaciones transaccionales de escritorio a aplicaciones web MVC

Carlos Angarita

Trabajo presentado como requisito parcial para optar al título de:

Ingeniero de Sistemas

Director (a):

M. Sc. Luis Alberto Esteban Villamizar

Universidad de Pamplona

Facultad de Ingenierías y Arquitectura

Pamplona Colombia

2017

Resumen

Este proyecto trata sobre un procedimiento para la migración de una aplicación transaccional de escritorio escrita en PL/SQL para Oracle Forms, hacia una aplicación web que emplea la arquitectura MVC como estrategia de diseño fundamentado en la experiencia como pasante en la empresa de energía de Arauca ENELAR E.S.P. El procedimiento describe la arquitectura del sistema, las herramientas tecnológicas utilizadas y las tareas seguidas al realizar la migración.

Palabras clave: PL/SQL, Java, Migración, Desarrollo Web, Servicios Web, MVC.

Abstract

This Project is about a procedure to migrate from a desktop transactional application written in PL/SQL for Oracle Forms, to a Web Application implementing a MVC Architecture as design strategy based on the experience as an intern at the Arauca energy company ENELAR E.S.P. The procedure describes the system architecture, the technological tools used and the tasks followed during the migration.

Keywords: PL/SQL, Java, Migration, Web Development, Web Services, MVC.

Contenido

Pág.

Tabla de contenido

Pág.

Tabla de Ilustraciones

1. Introducción

La computación es un campo que ha venido evolucionando constantemente en las últimas décadas. Desde la ENIAC[1] hasta los dispositivos móviles de hoy tan solo han pasado 70 años en los cuales lenguajes de programación y paradigmas han nacido, se han desarrollado y han desaparecido. Lo mismo ha sucedido con el software, eventualmente cumple su ciclo de vida por varios motivos; puede que la necesidad que suplía ya no sea importante para la organización en la que estaba o que se haya encontrado un nuevo que realiza las mismas funciones, de una forma más eficiente. Sin embargo, generalmente el culmina su ciclo de vida debido a que la tecnología en la que se soportaba es considera obsoleta, en tal caso las organizaciones pueden optar por iniciar un proceso de migración a una tecnología contemporánea.

En resumen una migración es el proceso por el cual se parte de la lógica de negocio del software anterior (denominado software de legado) y se traslada hacia uno nuevo implementando tecnología contemporánea. La migración presenta ventajas frente al diseño de un software nuevo debido a que la lógica y los casos de uso ya se encuentran suficientemente probados, haciendo que el proceso de diseño e implementación sea rápido y fiable.

En el segundo capítulo se trata el estudio bibliográfico de la arquitectura MVC. Este capítulo se enfoca en el manejo de la arquitectura desde el punto de vista del *Frontend* y del *Backend*. Se introducen los principios teóricos detrás del MVC y se muestra su ejecución desde diferentes *frameworks*; igualmente se discuten los estándares para la comunicación de datos desde el cliente hacia el servidor y viceversa.

En el tercer capítulo se describe el procedimiento para la migración empleado en la Empresa de Energía de Arauca, ENELAR E.S.P. En él se describe la arquitectura del sistema y se detallan los pasos seguidos desde que inicia el requerimiento de migración

[1] Acrónimo de ***E****lectronic* ***N****umerical* ***I****ntegrator* ***A****nd* ***C****omputer* (Computador e Integrador Numérico Electrónico). Una de las primeras computadoras creadas.

hasta que se coloca en producción lo migrado. También se muestra el uso del sistema de control de versiones y la política de documentación del código fuente, al igual que los criterios para la modificación de la lógica del negocio. Finalmente se ejemplifica el procedimiento planteado en el segundo capítulo a través de dos funcionalidades migradas a web y se muestra en ejecución la forma de implementación de ENELAR E.S.P. a partir de las experiencias de los desarrolladores que han hecho parte de la migración del Sistema de Información Comercial.

1.1 Objetivos

OBJETIVO GENERAL

- Proponer un procedimiento para la migración de aplicaciones transaccionales a entornos web MVC.

OBJETIVOS ESPECÍFICOS

- Realizar un estudio bibliográfico sobre arquitecturas MVC[2] para el desarrollo de aplicaciones web.
- Definir un procedimiento para la migración de aplicaciones transaccionales de escritorio a web con arquitectura MVC.
- Validar el procedimiento desarrollando dos funcionalidades web basadas en la arquitectura MVC.

1.2 Planteamiento del problema

Las TIC avanzan a pasos agigantados, día a día emergen nuevas metodologías, métricas, técnicas y tecnologías que ofrecen posibilidades casi infinitas, pero que requiere de una vigilancia y actualización constante. En la empresa de energía de Arauca

[2] Acrónimo de ***M****odelo* ***V****ista* ***C****ontrolador.* Patrón de arquitectura para separar los datos y la lógica del negocio de la interfaz gráfica.

ENELAR E.S.P. se desarrolló hace más de 15 años un sistema de información para el uso interno con una tecnología de la época. Dicha tecnología se encuentra desactualizada comparada con los estándares actuales de la industria de desarrollo de software y requiere de una migración hacia una tecnología contemporánea.

En ENELAR se tomó la decisión de realizar el proceso de migración del Sistema de Información Comercial SIC. Dicho sistema fue diseñado en Oracle Forms, basado en una arquitectura Cliente/Servidor en la que a cada usuario del sistema se le instalaba una instancia del cliente de Forms y se accedía al sistema a través de una unidad de red compartida que contiene las formas.

Desde el punto de vista de la interfaz se dividieron las formas en varias categorías, de acuerdo a las necesidades organizacionales, por ejemplo Facturación, Atención al Cliente, Cartera, etc. Cada categoría se subdivide en cuatro tipos de acciones: Administración, Actualización, Consultas y Reportes y cada Forma solo hacía parte de una de ellas. Desde el punto de vista de arquitectura, el sistema era un conjunto de Módulos o Formas débilmente acoplado en la que cada unidad se encarga de realizar una única función del sistema, es decir, existía una Forma para ingresar usuarios, una forma para imprimir la factura, una forma para crear un crédito, etc.

Aunque la arquitectura favorecía el diseño, programación, prueba y mantenimiento del módulo, tenía la desventaja de que propagar cambios en el sistema se era una tarea compleja, y la interacción entre módulos podía llegar a ser retadora, de igual manera no era escalable, pues al acceder a los ejecutables desde una unidad compartida se generaba una fuerte carga a la red.

1.3 Justificación

Migrar un sistema significa convertirlo de un lenguaje de programación a otro, posiblemente cambiando su paradigma; por ejemplo migrar un programa de C a JAVA implica cambiar de un paradigma estructurado y procedural a un paradigma orientado a objetos. Documentar el procedimiento de migración de un sistema a otro, en el que se defina la ruta de trabajo a seguir y se determinen las ventajas y desventajas que ofrecen

tecnologías contemporáneas permitirá tener un punto de referencia para agilizar futuros trabajos de migración.

Vale la pena mencionar que la migración se hizo en un entorno web debido a las múltiples ventajas que esto trae como la eliminación de la necesidad de una red interna para acceder al sistema de información, la mejora en la experiencia de usuario que hace que el acceso a la información sea más eficiente e intuitiva y por último evita el trabajo de tener que instalar software en cada una de las terminales.

1.4 Metodología del trabajo

Este proyecto se realizó a partir de la experiencia como pasante en el área de sistemas de ENELAR E.S.P. En dicha pasantía se continuó con el trabajo de migración del Sistema de Información Comercial que había comenzado el área, por lo que el enfoque se dirigió hacia la arquitectura y forma de trabajo encontradas. El sistema originalmente era una serie de formularos agrupados por un menú de acuerdo a las reglas del negocio que al migrarlo se convirtió en una serie de proyectos igualmente agrupados por un menú. Cada proyecto consiste en una clase denominada “Modelo” que contienen las funcionalidades del proyecto, una clase “Controlador” que determina que funcionalidad del proyecto ejecutar y una página en JSP que contiene las etiquetas de HTML y el script que determina el comportamiento de la página ante las entradas del usuario y la forma de mostrar la información recibida del servidor.

2. Marco Teórico

El presente capítulo trata de las bases teóricas en las que se basa el proyecto. Acá se verá la relación entre aplicaciones de escritorio y aplicaciones web, el patrón de diseño MVC y su implementación en entornos de frontend y de backend.

2.1 Aplicaciones de escritorio y Aplicaciones Web

Es posible clasificar una aplicación en dos categorías principales de acuerdo al ambiente en el que se despliegan, estas son: aplicaciones de escritorio y aplicaciones web. De manera sencilla, una aplicación de escritorio es aquella que se ejecuta de manera local en un dispositivo de cómputo, mientras que una aplicación web se va mostrando a un dispositivo local desde un servidor remoto a través de Internet.

Las aplicaciones de escritorio se caracterizan por no requerir una conexión constante a Internet para funcionar. La distribución y actualización suele ser compleja puesto que hay que se tiene que realizar en todos los dispositivos y el desarrollo de estas aplicaciones depende de la plataforma y hardware en el que se vayan a desplegar pero esto le da mayor acceso a los recursos del dispositivo en el que se despliegue.

Las aplicaciones web usualmente tienen una dependencia de la conexión a Internet por lo que no es necesario realizar instalaciones y/o actualizaciones en cada máquina cliente pues estas se realizan del lado del servidor. Pueden ejecutarse en diferentes sistemas operativos o plataformas por lo que el desarrollo y distribución son más sencillos, sin embargo no hay un acceso completo a los recursos del dispositivo local.[1]

2.2 MVC

El patrón de diseño MVC (Modelo-Vista-Controlador) fue concebido por Trygve Reenskaug como una forma de desacoplar los distintos elementos que componen un típico desarrollo de software de manera que el código pueda ser fácilmente mantenible y reusable. La triada de clases (Modelo/Vista/Controlador) fue descrita por Krasner y Pope en 1980 consiste en tres tipos de objetos. El Modelo es el objeto de la aplicación, la Vista es la pantalla de presentación y el Controlador define la forma en la que la interfaz de usuario reacciona a las entradas.

MVC desacopla las vistas y los modelos estableciendo un protocolo de suscripción/notificación entre ellas. Una Vista debe asegurar que su apariencia refleje el estado del modelo. Cada vez que el modelo de datos cambia, este notifica a las vista que dependen de él. En respuesta, cada vista obtiene la oportunidad de actualizarse a sí misma. Este acercamiento permite unir múltiples vista a un modelo para proveer diferentes presentaciones. También se pueden crear nuevas vistas para un modelo sin tener que reescribirlo. [2]

2.2.1 Un Vistazo a la Historia

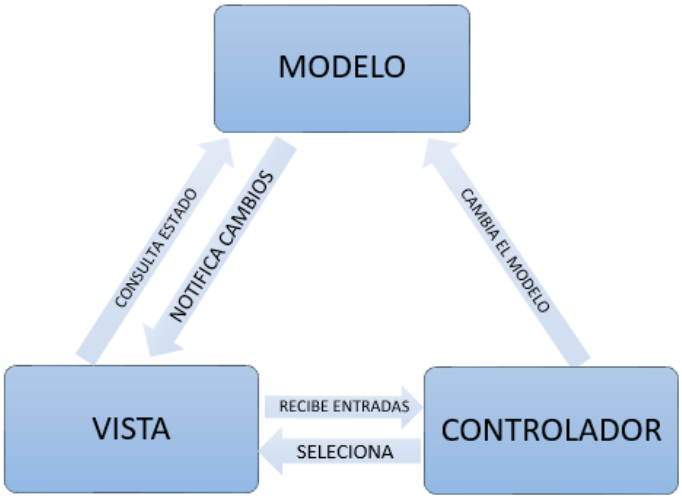

Ilustración 1 Diagrama Relaciones MVC

Originalmente el MVC fue desarrollado como un patrón de diseño para Smalltalk 80 un lenguaje reflexivo de programación, orientado a objetos y con tipado dinámico. Por sus características, Smalltalk puede ser considerado también como un entorno de objetos, donde incluso el propio sistema es un objeto. Metafóricamente, se

puede considerar que un Smalltalk es un mundo virtual donde viven objetos que se comunican entre sí, mediante el envío de mensajes.[3]

Como uno de los primeros intentos serios de trabajar en interfaces de usuario, MVC inicia con la idea de dividir los Objetos de Dominio de los Objetos de Presentación. Los Objetos de Dominio modelan nuestra percepción del mundo real, es decir son objetos que contienes los datos y la lógica del negocio; estos objetos debe ser auto contenidos para que pueda soportar múltiples objetos de presentación. Por otro lado los Objetos de Presentación manejan los elementos visuales en la pantalla, se encargan de observar un modelo, se notifica de cuando este cambia y se actualiza de acuerdo a los cambios encontrados.

En MVC los Objetos de Domino se denominan Modelo. Los objetos de Modelo no conocen nada sobre la Interfaz de Usuario, manejan los datos y la lógica de una aplicación. En el caso de que los datos del modelo cambien, este notifica a la vista o al controlador dependiendo de si se necesita una lógica diferente para actualizar a la vista.

Los Objetos de Presentación se dividen en dos partes, la Vista y el Controlador. La Vista es el objeto que muestra en pantalla los componentes genéricos de interfaz de usuario en los cuales el usuario puede visualizar los datos y producir entradas en la aplicación. Las entradas luego son manejadas por el controlador quien se encarga de decir que hacer con estas, por ello en el MVC clásico debe existir un par vista-controlador por cada elemento en la pantalla.[4][2]

2.3 Arquitectura *Frontend* y Backend

En diseño de software el *frontend* es la parte del software que interactúa con el o los usuarios y el backend es la parte que procesa la entrada desde el *frontend*. La separación del sistema en *frontend*s y *backends* es un tipo de abstracción que ayuda a mantener las diferentes partes del sistema separadas. La idea general es que el *frontend* sea el responsable de recolectar los datos de entrada del usuario, que pueden ser de muchas y variadas formas, y los transforma ajustándolos a las especificaciones que demanda el backend para poder procesarlos, devolviendo generalmente una respuesta

que el *frontend* recibe y expone al usuario de una forma entendible para este. La conexión del *frontend* y el backend es un tipo de interfaz.

En diseño web (o desarrollo web) hace referencia a la visualización del usuario navegante por un lado (*frontend*), y del administrador del sitio con sus respectivos sistemas por el otro (backend).[5]

2.3.1 Arquitectura *Frontend*

La arquitectura *Frontend* es una colección de herramientas y procesos que apunta a mejorar la calidad del código de *frontend* mientras crea un flujo de trabajo más eficiente y sostenible.

Ilustración 2 Tecnologías Frontend

Un arquitecto se define como alguien que diseña, planea y supervisa la construcción de edificios. Esto es exactamente lo que hace un arquitecto *frontend*, excepto que el producto final es un sitio web. Y así como un arquitecto gasta más tiempo en redactar esquemas que en verter hormigón, el arquitecto del *frontend* se preocupa más de construir herramientas y procesos que escribir código de producción.

Entrando en esta definición es posible observar el papel de un arquitecto *frontend*.

- **Diseño** La apariencia general de la página web sigue siendo en las manos de los diseñadores cualificados, pero el arquitecto *frontend* piensa el enfoque *frontend* y la filosofía del sistema de diseño. Al diseñar un sistema en el que todos los desarrolladores de *frontend* van a trabajar, el arquitecto establece una visión clara como el producto final, el código, deberá verse.

  Una vez que un arquitecto *frontend* establece la visión, el proyecto tiene un estándar contra el cual probar el código. Sin un diseño para el producto terminado, ¿cómo se podría determinar si el código realmente cumplió con ese estándar? Un sistema cuidadosamente diseñado tendrá controles y equilibrios

asegurando que todo el código contribuido a ese sistema agrega valor perceptible, en lugar de simplemente agregar líneas de relleno.

- **Planificación** Con un diseño claro en mente, la etapa de planificación implica trazar el flujo de trabajo de desarrollo. ¿Qué pasos tomará un desarrollador para escribir una línea de código y ver ese código hasta producción? En el caso más simple, este plan implica subir un archivo por FTP a un servidor, editar el archivo y pulsar Guardar. Para la mayoría de los proyectos, incluirá una combinación de control de versiones, rutinas de tareas, procesadores CSS, herramientas de documentación, conjuntos de pruebas y servidores de automatización.
- **Vigilancia** La arquitectura de *Frontend* nunca es una propuesta de "establecer y olvidar". Ningún diseño o plan nunca es perfecto o completo. Las necesidades de los clientes (así como las necesidades de los desarrolladores) cambiarán y evolucionarán con el tiempo, y un proceso que funcionaba bien en una fase del proyecto podría necesitar ser revisado posteriormente para mejorar la eficiencia o reducir los errores. Las herramientas de construcción modernas hacen que sea muy fácil cambiar los flujos de trabajo y distribuir esos cambios a cada miembro del equipo.[6]

2.3.2 Arquitectura *Backend*

"Arquitectura" es un término que muchas personas tratan de definir, con poco acuerdo. Hay dos elementos comunes: Uno es el desglose de nivel más alto de un sistema en sus partes; El otro, decisiones que son difíciles de cambiar. También se está dando cuenta cada vez más que no hay una sola manera de indicar la arquitectura de un sistema; Más bien, hay múltiples arquitecturas en un sistema y la visión de lo que es arquitectónicamente significativo es aquella que puede cambiar a lo largo de la vida de un sistema.[7]

El backend es el lado utilizado por el proveedor de servicios. Incluye varios servidores, computadoras, sistemas de almacenamiento de datos y máquinas virtuales que en conjunto constituyen la nube de servicios de computación. Este sistema puede incluir diferentes tipos de programas informáticos. Cada aplicación en este sistema es

administrada por su propio servidor dedicado. El backend tiene algunas responsabilidades para cumplir con el cliente

- Proporcionar mecanismos de seguridad, control de tráfico y protocolos
- Emplear protocolos que conecten las computadoras en red para la comunicación

- **Las Tres Capas Principales**

La arquitectura del backend consta de tres capas principales: presentación, dominio y fuente de datos.

- Presentación: La lógica de la presentación trata de cómo manejar la interacción entre el usuario y el software. Esto puede ser tan simple como un sistema de menús de línea de comandos o basado en texto, pero en estos días es más probable que sea una interfaz gráfica de usuario o una interfaz de usuario de navegador basada en HTML. Las responsabilidades principales de la capa de presentación son mostrar información al usuario e interpretar comandos del usuario en acciones sobre el dominio y el origen de datos.
- Fuente de Datos: La lógica de la fuente de datos consiste en comunicarse con otros sistemas que realizan tareas en nombre de la aplicación. Estos pueden ser monitores de transacciones, otras aplicaciones, sistemas de mensajería, y así sucesivamente. Para la mayoría de las aplicaciones empresariales, la mayor parte de la lógica de la fuente de datos es una base de datos que es principalmente responsable de almacenar datos persistentes.
- Dominio de Datos: La pieza restante es la lógica de dominio, también conocida como lógica de negocio. Este es el trabajo que esta aplicación necesita hacer para el dominio con el que está trabajando. Implica cálculos basados en entradas y datos almacenados, validación de cualquier información que viene de la presentación y determinación exacta de qué lógica de origen de datos enviar, dependiendo de los comandos recibidos de la presentación.

A veces, las capas están dispuestas de manera que la capa de dominio oculte completamente el origen de datos de la presentación. La presentación puede interpretar

un comando del usuario, utilizar el origen de datos para extraer los datos relevantes de la base de datos y, a continuación, permitir que la lógica del dominio manipular los datos antes de presentarlos.

Una sola aplicación a menudo puede tener múltiples paquetes de cada una de estas tres áreas temáticas. Una aplicación diseñada para ser manipulada no sólo por los usuarios finales a través de una interfaz de cliente enriquecido sino también a través de una línea de comandos tendría dos presentaciones: una para la interfaz de cliente enriquecido y otra para la línea de comandos.

Pueden estar presentes múltiples componentes de origen de datos para diferentes bases de datos, pero serían particularmente para la comunicación con paquetes existentes. Incluso el dominio puede ser dividido en distintas áreas relativamente separadas entre sí. Ciertos paquetes de origen de datos sólo pueden ser utilizados por ciertos paquetes de dominio.

El dominio y el origen de datos nunca deben depender de la presentación. Es decir, no debería haber ninguna llamada de subrutina desde el dominio o el código fuente de datos al código de presentación. Esta regla facilita la sustitución de diferentes presentaciones sobre el mismo fundamento y facilita la modificación de la presentación sin mayores ramificaciones. La relación entre el dominio y el origen de datos es más compleja y depende de los patrones arquitectónicos utilizados para el origen de datos.[7]

2.4 MVC aplicado en *Frontend*

El patrón de diseño MVC aplicado al frontend usualmente se realiza en Javascript y varía en su implementación dependiendo del framework en el que se utilice, principalmente el concepto del Controlador. A continuación se introduce el MVC en Javascript, el framework AngularJS y se detallan las herramientas incorporadas en este para la implementación de una aplicación MVC.

2.4.1 MVC en *Javascript*

JavaScript es un lenguaje ligero e interpretado, orientado a objetos con funciones de primera clase, más conocido como el lenguaje de script para páginas web, pero también usado en muchos entornos sin navegador, tales como *node.js* o *Apache CouchDB*.[8] JavaScript tiene ahora una serie de frameworks que cuentan con soporte para MVC (o variaciones de la misma, referida como la familia MV*), permitiendo a los desarrolladores agregar fácilmente estructura a sus aplicaciones sin gran esfuerzo. Estos frameworks incluyen a aquellos como *Backbone*, *Ember.js y AngularJS*.

Se sabe que MVC se compone de tres componentes principales Modelos, Vistas y Controladores. Los modelos gestionan los datos de una aplicación. No se ocupan de las capas de interfaz de usuario ni de presentación, sino que representan formas únicas de datos que una aplicación puede requerir. Cuando un modelo cambia (por ejemplo, cuando se actualiza), normalmente notificará a sus observadores que se ha producido un cambio para que puedan reaccionar en consecuencia.

Las capacidades incorporadas de los modelos varían entre frameworks, sin embargo, es bastante común que soporten la validación de atributos, donde los atributos representan las propiedades del modelo, como un identificador de modelo. Cuando se usan modelos en aplicaciones del mundo real, generalmente también se desea la persistencia de este. La persistencia nos permite editar y actualizar modelos con el conocimiento de que su estado más reciente se guardará en memoria, en el almacén de datos local o sincronizado con una base de datos.

Los textos más antiguos de MVC también pueden contener referencia a una noción de modelos que administran el estado de la aplicación. En las aplicaciones JavaScript, el estado tiene una connotación diferente, que suele referirse al estado actual, es decir, vista o sub-vista (con datos específicos). El Estado es un tema que se discute regularmente cuando se examinan aplicaciones de una sola página, donde el concepto de estado debe ser simulado.

Las vistas son una representación visual de modelos que presentan una vista filtrada de su estado actual. Mientras que las vistas de Smalltalk son sobre pintar y mantener un

mapa de bits, las vistas de JavaScript son acerca de la construcción y el mantenimiento de un elemento DOM.

Una vista normalmente observa un modelo y se notifica cuando el modelo cambia, permitiendo que la vista se actualice en consecuencia. La literatura de patrones de diseño se refiere comúnmente a las vistas como "tonto" dado que su conocimiento de los modelos y controladores en una aplicación es limitado.

Los usuarios pueden interactuar con vistas y esto incluye la capacidad de leer y editar (es decir, obtener o establecer los valores de los atributos) en los modelos. Como la vista es la capa de presentación, generalmente se presenta la posibilidad de editar y actualizar de una manera fácil de usar.

El beneficio de esta arquitectura es que cada componente desempeña su propio papel independiente al hacer que la aplicación funcione según sea necesario.

Durante mucho tiempo se ha considerado una mala práctica de rendimiento el crear manualmente grandes bloques de HTML en memoria a través de concatenación de caracteres. Los desarrolladores que lo hacen han caído presa de iterar con bajo rendimiento a través de sus datos, envolviéndolo en divs anidados y utilizando técnicas anticuadas como *"document.write"* para inyectar la "plantilla" en el DOM. Como esto suele significar mantener el marcado script alineado con el marcado estándar, puede llegar a ser difícil de leer y, lo que es más importante, mantener, especialmente cuando se construyen aplicaciones de tamaño no trivial.

Los controladores son un intermediario entre modelos y vistas que son clásicamente responsables de actualizar el modelo cuando el usuario manipula la vista.

En términos de donde la mayoría de los framework de JavaScript MVC desvirtúan lo que se considera convencionalmente "MVC", es con los controladores. Las razones de esto varían, pero una de ellas, es que los autores del framework inicialmente miran a la interpretación del lado del servidor de MVC, se dan cuenta de que no se traduce 1:1 en el lado del cliente y re-interpretan el C en MVC para significar algo que sienten tiene más sentido. El problema con esto sin embargo es que es subjetivo, aumenta la complejidad en ambos entender el patrón clásico de MVC y por supuesto el papel de reguladores en

frameworks modernos[2]. Por ejemplo en AngularJS el concepto de Controlador es el de un Objeto que contiene propiedades y funciones. El Objeto que representa al controlador es denominado *$scope* y toda propiedad del Modelo que sea visible en la vista hace parte de este.

La separación de preocupaciones en MVC facilita la modularización de la funcionalidad de una aplicación y permite:

• Fácil mantenimiento general. Cuando las actualizaciones deben hacerse a la aplicación es muy claro si los cambios son centrados en los datos, significando cambios en los modelos y tal vez controladores, o simplemente visual, lo que significa cambios en las vistas.

• Desacoplar modelos y vistas significa que es mucho más sencillo escribir pruebas unitarias para la lógica de negocio.

• Dependiendo del tamaño de la aplicación y de la separación de funciones, esta modularidad permite a los desarrolladores responsables de la lógica básica y los desarrolladores que trabajan en las interfaces de usuario trabajar simultáneamente

2.4.2 *AngularJS*

Es un framework de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.

Angular lee el HTML que contiene atributos y etiquetas personalizadas, obedece a las directivas de los atributos personalizados, y une las piezas de entrada o salida de la página a un modelo representado por las variables de JavaScript. Los valores de las variables de JavaScript se pueden configurar manualmente, o recuperados de recursos JSON estáticos o dinámicos.

AngularJS está construido en torno a la creencia de que la programación declarativa es la que debe utilizarse para generar interfaces de usuario y enlazar componentes de software, mientras que la programación imperativa es excelente para expresar la lógica

de negocio. Este framework adapta y amplía el HTML tradicional para servir mejor contenido dinámico a través de un data binding bidireccional que permite la sincronización automática de modelos y vistas. Como resultado, AngularJS pone menos énfasis en la manipulación del DOM y mejora la testeabilidad y el rendimiento.

Los objetivos de diseño:

- Disociar la manipulación del DOM de la lógica de la aplicación. Esto mejora la capacidad de prueba del código.
- Considerar a las pruebas de la aplicación como iguales en importancia a la escritura de la aplicación. La dificultad de las pruebas se ve reducida drásticamente por la forma en que el código está estructurado.
- Disociar el lado del cliente de una aplicación del lado del servidor. Esto permite que el trabajo de desarrollo avance en paralelo, y permite la reutilización de ambos lados.
- Guiar a los desarrolladores a través de todo el proceso del desarrollo de una aplicación: desde el diseño de la interfaz de usuario, a través de la escritura de la lógica del negocio, hasta las pruebas.[9]

Ilustración 3 Logo AngularJS

Angular ofrece una serie de funcionalidades para diseñar una aplicación MVC, por ejemplo las directivas que permiten insertar y extender el HTML con nuevas funcionalidades, haciendo que la lectura del código fuente sea más clara para los desarrolladores; los servicios que son objetos que permiten compartir código en una aplicación y se utilizan para acceder a recursos remotos; por último los controladores que son constructores que permiten la instanciación de un *$scope*, su estado inicial y comportamiento.

- **Controladores en AngularJS**

En AngularJS, un controlador se define mediante una función de constructor de JavaScript que se utiliza para aumentar el scope de AngularJS.

Cuando un Controlador está conectado al DOM a través de la directiva ng-controller, AngularJS instanciará un nuevo objeto Controller, usando la función constructora del Controlador especificada. Se creará un nuevo scope hijo y se pondrá a disposición como parámetro inyectable a la función constructora del controlador como *$scope*.

> Si el controlador se ha conectado utilizando el "controller as" como sintaxis, la instancia del controlador se asignará a una propiedad en el nuevo scope.

Los controladores se usan para:

- Configurar el estado inicial del objeto *$scope.*
- Agregar comportamiento al objeto *$scope.*

Los controladores no se usan para:

- Manipular DOM - Los controladores deben contener sólo lógica de negocio. Poner cualquier lógica de presentación en los controladores afecta significativamente su testeabilidad. AngularJS tiene enlace de datos para la mayoría de los casos y directivas para encapsular la manipulación DOM manual.
- Entrada de formato: utilice los controles de formulario AngularJS.
- Filtro de salida - En su lugar, utilice filtros AngularJS.
- Compartir código o estado en todos los controladores - Utilice los servicios de AngularJS en su lugar.
- Administrar el ciclo de vida de otros componentes (por ejemplo, para crear instancias de servicio).

- **Configuración del estado inicial de un objeto *$scope***

Normalmente, cuando se crea una aplicación, es necesario configurar el estado inicial del *$scope.* Se configura el estado inicial de un scope adjuntando propiedades al objeto *$scope*. Las propiedades contienen el modelo de vista (el modelo que será presentado por la vista). Todas las propiedades *$scope* estarán disponibles para la plantilla en el punto del DOM donde el controlador está registrado.

El siguiente ejemplo demuestra la creación de un GreetingController, que adjunta una propiedad denominada *greeting* que contiene la cadena 'Hola!' en el *$scope*:

```
1| var myApp = angular.module('myApp',[]);
2|
3| myApp.controller('GreetingController', ['$scope',
   function($scope) {
4|   $scope.greeting = 'Hola!';
5| }]);
```

Se crea un Módulo denominado myApp para la aplicación. Se añade la función constructor del controlador al módulo usando el método *.controller().*

Luego se añade el controlador al DOM usando la directiva *ng-controller.* La propiedad *greeting* puede ser enlazada a la plantilla:

```
1| <div ng-controller="GreetingController">
2|   {{ greeting }}
3| </div>
```

- **Adición de comportamiento a un objeto scope**

Con el fin de reaccionar a los eventos o ejecutar el cálculo en la vista se debe proporcionar el comportamiento al scope. Se añade el comportamiento al scope adjuntando métodos al objeto *$scope*. Estos métodos están disponibles para ser llamados desde la plantilla/vista.

El ejemplo siguiente utiliza un Controlador para agregar un método, que duplica un número en el scope:

```
1| var myApp = angular.module('myApp',[]);
2|
3| myApp.controller('DoubleController', ['$scope', function($scope) {
4|   $scope.double = function(value) { return value * 2; };
5| }]);
```

Una vez que el Controlador ha sido conectado al DOM, el método *double* puede ser invocado en una expresión AngularJS en la plantilla:

```
6| <div ng-controller="DoubleController">
7|   Dos veces <input ng-model="num"> es igual a {{ double(num) }}
8| </div>
```

Los objetos (o primitivas) asignados al scope se convierten en propiedades del modelo. Todos los métodos asignados al scope están disponibles en la plantilla/vista, y se pueden invocar mediante expresiones AngularJS y las directivas de manejo de eventos como ngClick.[10]

- **Servicios en Angular**

AngularJS apoya los conceptos de "separación de preocupaciones" utilizando la arquitectura de servicios. Los servicios son funciones JavaScript y son responsables de realizar tareas específicas. Esto los convierte en una entidad individual que es mantenible y comprobable. Los controladores y filtros pueden llamarlos a medida que se necesiten. Los servicios se inyectan normalmente usando el mecanismo de la inyección de la dependencia de AngularJS.

Cada servicio es responsable de una tarea específica, por ejemplo, *$https*: se utiliza para realizar una llamada Ajax para obtener los datos del servidor. *$route* se utiliza para definir la información de enrutamiento y así sucesivamente. Los servicios incorporados siempre tienen el prefijo $.

Hay dos maneras de crear un servicio: Factory y Service.

- **Utilizando el método *factory***

Utilizando el método *factory*, primero se define un *factory* y luego se le asigna un método.

```
1| var mainApp = angular.module("mainApp", []);
2| mainApp.factory('MathService', function() {
3|    var factory = {};
4|
5|    factory.multiply = function(a, b) {
6|       return a * b
7|    }
8|
9|    return factory;
10|});
```

- Utilizando el método *service*

Mediante el método *service*, se define un servicio y luego se le asigna un método inyectado un servicio ya disponible.[11]

```
1| mainApp.service('CalcService', function(MathService){
2|    this.square = function(a) {
3|       return MathService.multiply(a,a);
4|    }
5| });
```

Ambos métodos son válidos para generar un servicio, la diferencia radica en que una llamada a un método creado con el *service* retornará una nueva instancia del objeto que contiene sus propiedades, mientras el llamado al método creado con *Factory* retornará el objeto como tal.

- **Directivas en AngularJS**

En un nivel alto, las directivas son marcadores en un elemento DOM (como un atributo, nombre de elemento, comentario o clase CSS) que indican al compilador HTML de AngularJS (*$compile*) que adjunte un comportamiento a ese elemento DOM (por ejemplo, a través de escuchadores de eventos), o incluso que transforme el elemento DOM y sus hijos.

AngularJS viene con un conjunto de estas directivas integradas, como ngBind, ngModel y ngClass. Al igual que crear controladores y servicios, puede crear sus propias directivas para que AngularJS pueda utilizar. Cuando AngularJS inicia su aplicación, el compilador HTML recorre las directivas que encuentra en el DOM con los elementos DOM.

- **Tipos de Directivas**

$compile puede coincidir con directivas basadas en nombres de elemento (E), atributos (A), nombres de clase (C) y comentarios (M).

Una directiva puede especificar cuál de los 4 tipos coincidentes que admite en la propiedad restrict del objeto de definición de directiva. El valor predeterminado es EA.

- **Creación de Directivas**

Al igual que los controladores, las directivas se registran en los módulos. Para registrar una directiva, se utiliza la API *module.directive*. *Module.directive* toma el nombre de directiva normalizada (es decir en estilo camelCase) seguido por una función *factory*. Esta función de *factory* debe devolver un objeto con las diferentes opciones para decirle a *$compile* cómo debería comportarse la directiva cuando coincida.

La función de *factory* se invoca sólo una vez cuando el compilador coincide con la directiva por primera vez. Puede realizar cualquier trabajo de inicialización aquí. La función se invoca con *$injector.invoke* que lo hace inyectable como un controlador.

A continuación el Javascript que construye la directiva:

```
1| angular.module('docsSimpleDirective', [])
2| .controller('Controller', ['$scope', function($scope) {
3|   $scope.customer = {
4|     name: 'Naomi',
5|     address: '1600 Amphitheatre'
6|   };
7| }])
8| .directive('myCustomer', function() {
9|   return {
10|    restrict: 'A',
11|    template: 'Name: {{customer.name}} Address: {{customer.address}}'
12|  };
13|});
```

Y el HTML de la directiva es el siguiente:

```
1| <div ng-controller="Controller">
2|   <div my-customer></div>
3| </div>
```

2.5 MVC aplicado en *Backend*

El patrón MVC aplicado en el contexto de backend divide las tres capas de la arquitectura de forma que el Modelo representa el dominio fuente de datos, la Vista representa la capa de Presentación que se diseña utilizando una arquitectura frontend y el Controlador que determina la comunicación entre el Modelo y la Vista, dicha comunicación pude ser estandarizada utilizando el estilo de arquitectura REST[3].

2.5.1 El modelo

La capa de Modelo se puede dividir en tres parte, el Datasource que gestiona la conexión a una base de datos, una clase Modelo también denominada DAO en la que se realizan las operaciones de lógica del negocio, y opcionalmente una clase POJO[4] también llamado VO.

[3] Transferencia de Estado Representacional (en inglés Representational State Transfer) o REST es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.
[4] Plain Old Java Object

Ilustración 4 Componentes del Modelo

- **El DataSource**

Los objetos DataSource pueden proporcionar agrupaciones de conexiones y transacciones distribuidas. Esta funcionalidad es esencial para la computación de bases de datos empresariales. El trabajo realizado para implementar las clases que hacen posible estas operaciones, que un administrador del sistema suele hacer con una herramienta (como Apache Tomcat o Oracle WebLogic Server), varía con el tipo de objeto DataSource que se está implementando.

Los objetos instanciados por clases que implementan DataSource representan un DBMS[5] particular o algún otro origen de datos, como un archivo. La interfaz DataSource es implementada por un proveedor de controladores. Se puede implementar de tres maneras diferentes:

- Una implementación básica de DataSource produce objetos de conexión estándar que no se agrupan ni se utilizan en una transacción distribuida.
- Una implementación de DataSource que admite el agrupamiento de conexiones produce objetos de conexión que participan en agrupamiento de conexiones, es decir, las conexiones que se pueden reciclar.
- Una implementación de DataSource que admite transacciones distribuidas produce objetos de conexión que se pueden utilizar en una transacción distribuida, es decir, una transacción que tiene acceso a dos o más servidores de DBMS.

[5] Manejador de Base de Datos (en Ingles Data Base Manager System)

La implementación de un objeto DataSource consta de tres tareas:

- Creación de una instancia de la clase DataSource
- Estableciendo sus propiedades
- Registrarlo con un servicio de nombres que utiliza la API de Java Naming and Directory Interface (JNDI).[12]

- **El DAO**

El objeto de acceso a datos (DAO) es un objeto que proporciona una interfaz abstracta a algún tipo de base de datos u otro mecanismo de persistencia. Al asignar llamadas de aplicación a la capa de persistencia, el DAO proporciona algunas operaciones de datos específicos sin exponer detalles de la base de datos. Este aislamiento apoya el principio de responsabilidad individual. Separa los datos de acceso que la aplicación necesita, en términos de objetos específicos de dominio y tipos de datos (la interfaz pública del DAO), de cómo estas necesidades se pueden satisfacer con un DBMS específico, esquema de base de datos, etc.

Aunque este patrón de diseño es igualmente aplicable a la mayoría de los lenguajes de programación, la mayoría de los tipos de software con necesidades de persistencia y la mayoría de los tipos de bases de datos, tradicionalmente se asocia con aplicaciones Java EE[6] y con bases de datos relacionales a través de la API JDBC debido a su origen en las guías de buenas prácticas de Sun Microsystems "Core J2EE Patterns" para esa plataforma.

La ventaja de usar objetos de acceso a datos es la separación relativamente simple y rigurosa entre dos partes importantes de una aplicación que puede pero no deben saber nada el uno del otro, y que se puede esperar que evolucione con frecuencia e independientemente. El cambio de lógica de negocio puede confiar en la misma interfaz DAO, mientras que los cambios en la lógica de persistencia no afectan a los clientes DAO siempre que la interfaz permanezca implementada correctamente. Todos los detalles del almacenamiento se ocultan del resto de la aplicación. Por lo tanto, los posibles cambios en el mecanismo de persistencia se pueden implementar con sólo

[6] Java Edición Empresarial

modificar una aplicación DAO mientras que el resto de la aplicación no se ve afectada. Los DAO actúan como intermediarios entre la aplicación y la base de datos. Mueven datos de un lado a otro entre los objetos y los registros de la base de datos. Las pruebas de unitarias de código se facilitan mediante la sustitución de la DAO con un doble de prueba, por lo que las pruebas no dependen de la capa de persistencia.

En el contexto no específico del lenguaje de programación Java, Data Access Objects como concepto de diseño puede implementarse de varias maneras. Esto puede ir desde una interfaz bastante sencilla que separa las partes de acceso a datos de la lógica de la aplicación, a marcos y productos comerciales. Las tecnologías como Java Persistence API y Enterprise JavaBeans vienen incorporadas a los servidores de aplicaciones y pueden utilizarse en aplicaciones que utilizan un servidor de aplicaciones Java EE. Los productos comerciales como TopLink están disponibles basados en mapeo Objeto-relacional (ORM). Los productos ORM de código abierto más populares incluyen implementaciones de Doctrine, Hibernate, iBATIS y JPA como Apache OpenJPA.[13]

- **El POJO**

Un POJO (acrónimo de Plain Old Java Object) es una sigla creada por Martin Fowler, Rebecca Parsons y Josh MacKenzie en septiembre de 2000 y utilizada por programadores Java para enfatizar el uso de clases simples y que no dependen de un framework en especial. Este acrónimo surge como una reacción en el mundo Java a los frameworks cada vez más complejos, y que requieren un complicado andamiaje que esconde el problema que realmente se está modelando. En particular surge en oposición al modelo planteado por los estándares EJB anteriores al 3.0, en los que los "Enterprise JavaBeans" debían implementar interfaces especiales.

Un objeto POJO es una instancia de una clase que no extiende ni implementa nada en especial. Por ejemplo, un Servlet tiene que extender de HttpServlet y sobrescribir sus métodos, por lo tanto no es un POJO. En cambio, si se define una clase 'Persona', con sus atributos privados y sus correspondientes getters y setters públicos, una instancia de esta simple clase es un objeto POJO.

Con el auge de JSON se utilizan POJO para serializar los objetos en formato JSON, con bibliotecas como Gson[14].

Usualmente un POJO es una entidad que representa el conjunto de datos que sirven como parámetro o como resultado de los métodos del DAO, estandarizando los tipos de datos y los resultados esperados; también representa una gran ventaja al serializar las respuestas de una petición HTTP al convertir el objeto a formato JSON.

2.5.2 La Vista

La capa de la vista es con la que interactúa el usuario, allí se muestra la información y se permite ingresar datos al sistema; gracias a la arquitectura FrontEnd y BackEnd, la vista es manejada por el primero y lo único que necesita conocer es donde se encuentran los recursos o servicios, dependiendo de cómo se haya planteado el sistema[7].

2.5.3 El Controlador

El controlador interactúa con el modelo y con la vista. Controla el flujo de datos que llega y sale del modelo y actualiza la vista cuando los datos de modifican. En Java, existen múltiples frameworks que ayudan con la implementación del controlador, por ejemplo en Jersey[8] que proporciona soporte para la creación de servicios web.

JAX-RS proporciona algunas anotaciones para ayudar a mapear una clase recurso (un POJO) como un recurso web. Entre estas anotaciones se incluyen:

- `@Path` especifica la ruta de acceso relativa para una clase recurso o método.
- `@GET`, `@PUT`, `@POST`, `@DELETE` y `@HEAD` especifican el tipo de petición HTTP de un recurso.
- `@Produces` especifica los tipos de medios MIME de respuesta.
- `@Consumes` especifica los tipos de medios de petición aceptados.

[7] Bajo la filosofía del REST se observa al backend como un conjunto de recursos sobre los que se puede operar, mientras que bajo el protocolo SOAP (Simple Object Access Protocol) se observa como un conjunto de servicios a consumir.
[8] Jersey es una implementación del JAX-RS: Java API for RESTful Web Services.

Además, proporciona anotaciones adicionales para los parámetros de método para extraer información de la solicitud. Todas las anotaciones `@*Param` toman una clave de alguna forma que se utiliza para buscar el valor requerido.

- `@PathParam` enlaza el parámetro a un segmento de ruta.
- `@QueryParam` enlaza el parámetro al valor de un parámetro de consulta HTTP.
- `@MatrixParam` enlaza el parámetro al valor de un parámetro de matriz de HTTP.
- `@HeaderParam` enlaza el parámetro a un valor de cabecera HTTP.
- `@CookieParam` enlaza el parámetro a un valor de cookie.
- `@FormParam` enlaza el parámetro a un valor de formulario.
- `@DefaultValue` especifica un valor por defecto para los enlaces anteriores cuando la clave no es encontrada.
- `@Context` devuelve todo el contexto del objeto. (Por ejemplo: @Context HttpServletRequest request)

3. Procedimiento para la migración de aplicaciones transaccionales a Web.

La migración de un sistema de computación sucede cuando se lo traslada de una plataforma a otra, lo que puede involucrar cambios de arquitectura y/o de tecnología, y normalmente lleva implícita la necesidad de reescribir los programas en un lenguaje diferente.

Si solo se considera la conversión de los lenguajes de los programas, en el mercado existen traductores de código que tienen la finalidad de contribuir a facilitar esta tarea. Sin embargo, estos cumplen una función esencialmente sintáctica, normalmente pobre desde el punto de vista semántico, y sus resultados se alejan demasiado del objetivo deseado. La traducción del código sin cambio en el paradigma conduce a programas monolíticos, ineficientes y difícilmente mantenibles. Por el contrario, si se considera el concepto de migración en su interpretación más amplia, el problema adquiere la dimensión de un proyecto de ingeniería de software y debe ser tratado en consecuencia, para lo cual se presentan diferentes alternativas.[15]

3.1 Descripción del Sistema de Legado

En ENELAR se tomó la decisión de realizar el proceso de migración del Sistema de Información Comercial SIC. Dicho sistema fue diseñado en Oracle Forms, basado en una arquitectura Cliente/Servidor en la que a cada usuario del sistema se le instalaba una instancia del cliente de Forms y se accedia al sistema a través de una unidad de red compartida que contiene las formas. Desde el punto de vista de arquitectura, el sistema era un conjunto de Módulos o Formas débilmente acoplado en la que cada unidad se encarga de realizar una única función del sistema, es decir, existia una Forma para ingresar usuarios, una forma para imprimir la factura, una forma para crear un crédito, etc.

Aunque la arquitectura favorecía el diseño, programación, prueba y mantenimiento del módulo, tenía la desventaja de que propagar cambios en el sistema se era una tarea compleja, y la interacción entre módulos podía llegar a ser retadora, de igual manera no era escalable, pues al acceder a los ejecutables desde una unidad de red se generaba una fuerte carga a la red. Desde el punto de vista de la interfaz se dividieron las formas en varias categorías, de acuerdo a las necesidades organizacionales, por ejemplo Facturación, Atención al Cliente, Cartera, etc. Cada categoría se subdivide en cuatro tipos de acciones: Administración, Actualización, Consultas y Reportes y cada Forma solo hacía parte de una de ellas.

3.2 Descripción del Sistema de Actual

El nuevo Sistema de Información Comercial se ha diseñado bajo la filosofía de mantener la lógica del negocio como una serie de servicios web accedidos por un cliente que conozca la dirección en la cual se encuentran publicados. Actualmente el sistema se compone de una serie de proyectos construidos a partir del código fuente de la Forma. Al igual que en el sistema de legado, cada proyecto hace parte de una categoría y se ubica en uno de los cuatro tipos de acciones (Administración, Actualización, Consultas o Reportes). Dicho proyecto solo puede ser visto y accedido por un usuario si tiene los permisos adecuados para su uso. Gracias a la separación de preocupaciones del patrón de diseño MVC, la lógica del negocio puede ser utilizada por una o varias vistas y los cambios se pueden propagar fácilmente por todo el sistema en comparación con la versión de legado.

Cada proyecto se diseña para respetar tres principios básicos de Ingeniería de Software, **la modularidad, cohesión y acoplamiento.**

- **Modularidad**: la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos que se puedan compilar por separado, pero que tienen conexiones con otros módulos. Según Bertrand Meyer: "*El acto de particionar un programa en componentes individuales para reducir su complejidad en algún grado… A pesar de particionar un programa es útil por esta razón, una justificación más poderosa para particionar un programa es que crea una serie de límites bien definidos y documentados en el programa. Estos límites, o interfaces, son muy valiosos en la comprensión del programa*".[16]
- **Cohesión:** La cohesión tiene que ver con que cada módulo del sistema se refiera a un único proceso o entidad. A mayor cohesión, mejor: el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener.

  Las clases tendrán alta cohesión cuando se refieran a una única entidad. Se puede garantizar una fuerte cohesión disminuyendo al mínimo las responsabilidades de una clase: si una clase tiene muchas responsabilidades probablemente haya que dividirla en dos o más. Y el test a aplicar sería ver si se

puede describir a la clase con una oración simple que tenga un único sustantivo en el sujeto. Si la clase estuviera representando alguna operación (por la aplicación de algún patrón de diseño, por ejemplo), también habría que tratar de "sustantivarla" y aplicarle la prueba para ver si es cohesiva. Una clase con alta cohesión suele cumplir el principio de única responsabilidad.[17]

- **Acoplamiento:** El acoplamiento mide el grado de relacionamiento de un módulo con los demás. A menor acoplamiento, mejor: el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener. Una clase, tendrá bajo acoplamiento cuando tenga la menor dependencia posible de otras clases. Esta dependencia significa que – si bien puede haber muchas clases que dependen de una – debería haber pocas dependencias hacia otras clases desde una sola. Las dependencias que importan son, de mayor a menor: generalización/herencia, composición, asociación y dependencia débil. Para visualizar estas cuestiones, los diagramas de clases son herramientas fundamentales. [17]

Para lograr estos tres principios, se parte de que cada Forma, por su diseño original es modular. Cada módulo típicamente se compone de tres capas basadas en el patrón MVC, una capa con la Vista en la forma de una página web basada en el estilo SPA[9]. Una capa con el Controlado como una clase de Java en la que se definen los servicios web, se validan los datos de entrada y salida y se realiza la redirección al modelo adecuado. Finalmente se cuenta con una capa de Modelo en la que se encuentra la lógica del negocio; acá es donde se aplica principalmente los principios de cohesión y acoplamiento. La capa de modelo típicamente consta de una sola clase de Java que no depende de ninguna otra para realizar sus operaciones, es decir, es poco acoplada. Igualmente la clase de modelo solo contiene métodos que se refieran a la operación que son necesarias para el módulo actual; es típico por ejemplo que cada servicio web ejecute un solo método para obtener el resultado.

[9] Acrónimo de Single Page Application. Es una aplicación web o es un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios como los de una aplicación de escritorio.

En la práctica lo que se encuentra en el SIC Web es una página de login en la que el usuario ingresa sus credenciales. De allí el sistema se redirecciona a una página con el listado de categorías a las que tiene acceso. Al hacer clic en una categoría se abre una nueva ventana con los cuatro tipos de acciones en un menú desplegable en el que se listan los nombres de los proyectos. Al hacer clic en el nombre del proyecto se carga en una página interna una instancia de la vista del proyecto seleccionado, esta a su vez realiza las llamadas asincrónicas necesarias al servidor para cargar el estado inicial. Dichas llamadas normalmente se realizan en el contexto del mismo módulo, asegurando su cohesión. Del mismo modo, si la operación que necesita realizar es común para todo el sistema, se realiza la llamada al servicio adecuado, por ejemplo, si se necesitan listar todas las zonas de operación de la empresa, se llama a un servicio de datos comunes, de esta forma se pueden reutilizar los módulos ya creados y al mismo tiempo maximizar la cohesión del módulo actual.

3.3 Etapas del proceso de Migración

Una vez determinada la arquitectura y filosofía de diseño del sistema se inicia el proceso de migración por módulos. Para que el proceso de cambio no sea traumático para la organización se mantienen simultáneamente la versión anterior y la nueva, mientras que se fomenta la cultura de cambio en los usuarios finales. Eventualmente se procede a remover del entorno de producción del sistema de legado el módulo migrado, dejando solamente la nueva versión.

Para organizar la migración de un módulo, se dividió el proceso en seis etapas secuenciales donde la última etapa, la etapa de soporte, se extiende durante todo el ciclo de vida del sistema.

3.3.1 Selección del módulo a migrar bajo criterios organizacionales.

El primer paso de esta etapa es documentar toda la estructura del sistema. La documentación se puede basar en el menú de navegación y debe contener un número de identificación para el módulo, el nombre, la URL[10] donde está o estará ubicado, el estado y la fecha de inicio y finalización de la migración. Es una buena práctica diseñar una estructura de nomenclatura para identificar fácilmente el módulo, por ejemplo utilizar una abreviatura de la categoría, el número de identificación de la categoría, el nombre abreviado del módulo y el número de identificación del mismo. Toda esta información permite rastrear el progreso de la migración del sistema y la evaluación de indicadores pertinentes al proceso.

Debido a la estructura modular del sistema, se pueden seleccionar las funcionalidades de acuerdo a las necesidades organizacionales presentes en el momento. Se establece una tabla de criterios para evaluar la prioridad y determinar el modulo que se va a migrar.

Tabla 3-1: Criterios de selección para el módulo a migrar.

Criterio de Selección	Prioridad
Módulo con errores o con cambios drásticos.	5
Módulo solicitado por parte de otra dependencia de la organización	4
Módulo frecuentemente utilizado por miembros de la organización	3
Módulo con diseño o funcionalidad similar al último módulo migrado.	2
Módulo seleccionado bajo criterio personal.	1

[10] Acrónimo de *Uniform Resource Locator* o Localizador Uniforme de Recursos.

Otros	1

Como se puede ver en la Tabla 2-1, toda vez que se quiera decidir qué módulo migrar se utiliza la tabla de criterios para seleccionar según el orden de prioridades dado el caso en el que dos módulos se encuentren empatados, se decide bajo criterio personal cual realizar. Luego de tomada la decisión de qué módulo migrar, se procede a la siguiente etapa.

3.3.2 Análisis de las funcionalidades del módulo a migrar.

Ya con el módulo seleccionado se inicia una fase de análisis de la Forma, en esta etapa se observa el comportamiento esperado del módulo en el sistema de legado, se realiza una revisión del código fuente y la GUI[11] con el objetivo de identificar las características o cálculos especiales y las entradas y salidas del módulo.

La etapa se constituye de tres pasos que se describen a continuación.

a) **Revisión de los datos de entrada:** Se ejecuta el módulo del sistema de legado para determinar que variables de entrada necesita y cómo se le presenta al usuario los formularios de entrada. Se realiza una captura de pantalla que funciona como insumo para la siguiente etapa.
b) **Prueba de escritorio del código fuente:** Se realiza una revisión del código fuente para determinar el flujo normal y alternativo del módulo, es en este paso donde posibles mejoras al sistema suelen ser determinadas y de ser posible se aclaran dudas con el o los responsables de mantener el código de legado. Se registra una copia del código fuente como insumo para la siguiente etapa.
c) **Revisión de los datos de salida:** Se observa la salida de la ejecución del módulo con el objetivo de establecer el resultado esperado y utilizarlo como comparativo para la evaluación en las etapas posteriores.

[11] Acrónimo de *Graphic User Interface* o Interfaz Gráfica de Usuario

Una vez se hayan completado los pasos y se hayan registrado los insumos para las etapas posteriores, se procede a la siguiente etapa.

3.3.3 Diseño del módulo bajo el patrón MVC.

Luego de que se haya observado el comportamiento esperado del módulo se procede a diseñar el nuevo módulo bajo el patrón MVC. Se diseña la nueva interfaz gráfica, y se determina que servicios crear o llamar. A continuación se describe cada paso.

a) **Diseño de la interfaz:** Utilizando como insumo el diseño de interfaz anterior se concibe un diseño de interfaz similar pero con los componentes gráficos que se hayan desarrollado para el nuevo sistema. Es recomendable utilizar los mismos componentes a través de todo el sistema de forma que sea más sencillo para el usuario final entender el funcionamiento de la interfaz de cualquier módulo. Existen muchos frameworks que tienen sistemas de plantillas que dan la facilidad de reutilizar componentes y escalarlos[12] por todo el sistema.
b) **Diseño de los servicios:** Debido a la filosofía de diseño, se ve el sistema como una serie de servicio consumidos por el cliente. Cada servicio posee una URL única, consume un tipo de contenido y produce otro tipo de contenido. Una buena práctica es producir contenido bajo el tipo *application/json* puesto que este formato es ampliamente conocido, el analizador sintáctico es más liviano y el procesamiento de los datos se puede hacer en varios lenguajes de programación. Es buena práctica retornar los datos en una estructura que sea entendida por todo el sistema y en la que se puedan retornar metadatos de la operación.
    - **Servicios ya creados:** Si el módulo requiere información u operaciones ya realizadas por algún servicio ya disponible en el sistema, se reutiliza a este de forma tal que no se haya código duplicado.
    - **Servicios a crear:** Basándose en el código fuente y la prueba de escritorio, se agrupan conjuntos de operaciones o de información en un servicio. Cada servicio tiene que tener un único propósito, por ejemplo

[12] Entiéndase escalar como la propiedad de un sistema de crecer o expandirse sin perder calidad o fiabilidad.

crear un usuario, actualizar una factura, etc. Utilizando como insumo el análisis del módulo, se determinan los tipos de contenidos que podrá recibir y retornar el servicio; esta información será implementada en la capa del controlador.

Luego de diseñada la interfaz gráfica y los servicios nuevos a implementar o los servicios ya creados a utilizar se pasa a la siguiente etapa del proceso de migración.

3.3.4 Implementación del módulo

La etapa de implementación del módulo el control de versiones, traducción del código fuente, construcción de la interfaz gráfica, llamado a los servicios web y documentación. En esta etapa se logra construir el módulo bajo la arquitectura, lenguaje de programación y estilo de diseño nuevos; da pie a implementar las mejoras identificadas en el análisis de las funcionalidades o en encontrar nuevas oportunidades de mejora. En resumen, el sistema se divide en dos, una parte de Backend implementada en Java y una parte de Frontend escrita en Javascript. Ambas partes se comunican a través de objetos JSON sobre los que fluye la información. El control de la aplicación se divide entre las dos partes, donde el Backend determina que servicio o página responde a un llamado y el Frontend se encarga de gestionar los eventos del usuario y el orden de renderizado en pantalla.

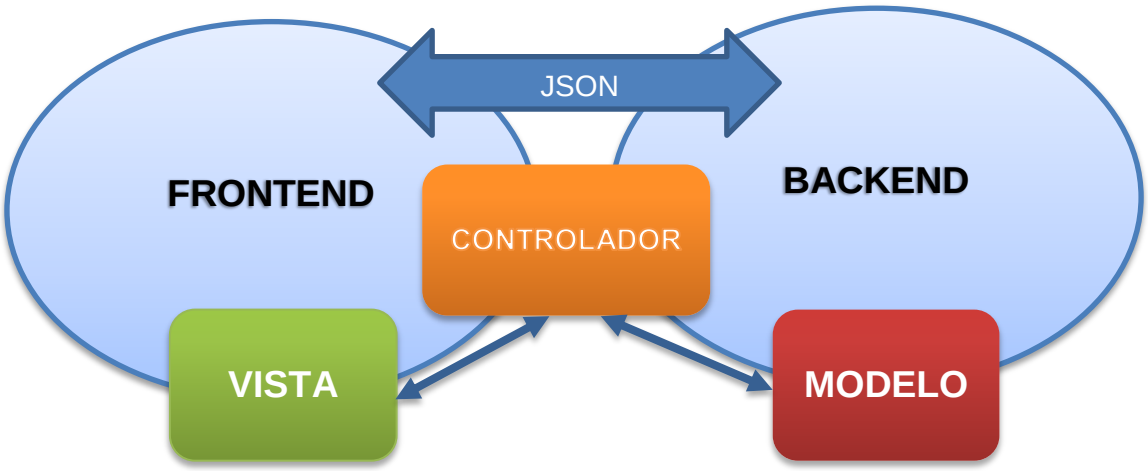

Ilustración 5 Diagrama de la arquitectura

En el Diagrama de la arquitectura se ve la relación de las capas del MVC con las dos partes del sistema. Cada capa tiene una tecnología base, en el caso del Modelo es framework Hibernate, encargado de gestionar la conexión y ejecución de consultas en el almacén de datos. En el caso de la vista se utiliza Angular que se encarga de mostrar y modificar la interfaz gráfica de acuerdo a las entradas del usuario. Por último el controlador se encuentra compartido entre el backend y el frontend, el primero usando la librería Jersey para la administración de los servicios y recursos mientras que el segundo utiliza las características de Angular para gestionar el acceso a los recursos de Backend que necesitan ser consumidos.

- **Implementación del Arquetipo:** Un arquetipo es un modelo o prototipo de una obra material o inmaterial. En desarrollo de software, el arquetipo permite generalizar proyectos que tienen una configuración similar y automatizar la construcción de un módulo nuevo. Con un arquetipo es posible iniciar la implementación con la estructura ya definida y con componentes reutilizables en el desarrollo. Al implementar la migración es buena práctica partir de un arquetipo hecho con un módulo ya suficientemente probado. Por ejemplo en Java, la herramienta de gestión de proyectos Apache Maven tiene una extensión para diseñar un arquetipo o para descargar uno de un repositorio libre.
- **Control de Versiones:** Un sistema de control de versiones es un software que ayuda a la gestión de los cambios que se realizan sobre los elementos de algún producto. Con la ayuda de un sistema de control de versiones, como el Git[13] es posible tener el historial de cambios y realizar trabajo colaborativo con otros desarrolladores sobre el mismo código fuente. Una vez creado e inicializado el repositorio de control de versiones se puede iniciar la codificación del módulo.
- **Traducción de código fuente:** Luego de diseñar los servicios web que componen la funcionalidad del módulo, se codifican en la capa del controlador. En

[13] Git es un sistema de control de versiones diseñado por Linus Torvalds para mantener las versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

la capa del modelo se traduce a partir de las fuentes de legado con la salvedad de mantener el estilo del lenguaje de destino. Por ejemplo en Java, es una práctica recomendada mantener la notación en el estilo *CameleCase*. Dependiendo de la situación queda a criterio del desarrollador si se realizan los cálculos u operaciones en el cliente, servidor o fuente de datos, garantizando que no se altere la lógica del negocio ni la fiabilidad de los datos. Se recomienda utilizar algún sistema de registro de errores para que realizar las tareas de pruebas y soporte sea más sencillo.

Durante este paso pueden surgir dudas que requieran realizar un nuevo análisis de la funcionalidad y una revisión al código fuente de legado. Finalmente todo servicio debe retornar una respuesta en un formato JSON con información de la operación realizada. Un ejemplo de una respuesta estándar sería el siguiente:

```
{
        "respuesta": "OK",
        "mensaje": "SE HA ENCONTRADO EL SUSCRIPTOR",
        "data": "['NOMBRE':'JHON DOE', 'EDAD': 17]",
        "cantidad": 1,
        "estado": "1"
}
```

- **Construcción de la Interfaz gráfica:** Inicialmente se deben definir componentes de interfaz de usuario que puedan ser reutilizados a través de todo el sistema. Con estos componentes de base se diseña una interfaz gráfica similar a la del sistema de legado de modo que la curva de aprendizaje de los usuarios no sea muy elevada. Acá se pueden definir restricciones a las entradas de datos de parte del usuario.
- **Llamado a los servicios web:** Desde el Frontend se realiza el llamado a los servicios web de acuerdo a las entradas del usuario. La capa de la vista se encuentra constantemente supervisando la presentación de la interfaz de usuario y determinando cómo mostrar los resultados de las respuestas del servidor. Gracias a las respuestas estándar, es posible decir si una operación se completó satisfactoriamente o si sucedió algún evento inesperado e informar de ello al usuario.
- **Documentación:** Para mantener un registro de qué hace cada clase o método, es buena práctica mantener una documentación adecuada. Por ejemplo, en Java existe Javadoc una utilidad de Oracle para la generación de documentación de

APIs en formato HTML a partir de código fuente Java[18]. La documentación permite a los desarrolladores guiarse y dar soporte al sistema mucho más fácilmente.

La implementación es la etapa más extensa de la migración, es acá donde todo el análisis y diseño previo se pone a prueba hasta lograr un resultado adecuado.

3.3.5 Pruebas unitarias, de integración y producción

A pesar de que la lógica del negocio ya está lo suficientemente probada, en la etapa de implementación suelen suceder situaciones inesperadas. Es por ello que realizar pruebas se vuelve una parte importante del proceso de migración. En un primer momento se realizan unas pruebas unitarias por parte del desarrollador, una vez culminadas se realiza una prueba de integración con el resto de sistema y finalmente se libera a producción donde el usuario final lo usará y evaluará.

- **Pruebas Unitarias:** Es una forma de comprobar el correcto funcionamiento de una unidad de código. Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, se verifica que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el estado inicial es válido entonces el estado final es válido [19]. Como se parte de un módulo de legado suficientemente probado, se puede validar que el resultado arrojado por este sea igual al del módulo migrado.
- **Pruebas de Integración:** Las pruebas de integración validan el funcionamiento o integración entre dos o más sistemas. Por ejemplo, el acceso a un servicio web o el almacenamiento en la base de datos. El objetivo de las pruebas de integración es validar el comportamiento de múltiples componentes. Dependiendo de la implementación pueden validar estados intermedios[20]. A pesar de que el sistema es débilmente acoplado, existe como parte de un todo. Cuando se realiza una prueba de integración se valida que el módulo funcione como parte del sistema, que realice llamados a otros servicios e importe librerías correctamente.

- **Producción:** Una vez validado el módulo se libera a producción de donde se recibe retroalimentación por parte de los usuarios finales. Es común en este paso recibir solicitudes de cambio en la capa de la Vista, estos cambios presentan prioridad frente a otros proyectos que se vengan realizando.

Completada esta etapa, se puede considerar que se ha migrado un módulo desde un sistema de legado a un sistema web en arquitectura MVC.

3.3.6 Soporte

Durante el resto del ciclo de vida del módulo se deberá dar soporte al mismo, por ello una buena documentación es esencial. Modificar un módulo en sus primeras horas es sencillo porque el desarrollador tiene claro cómo funciona todo, pero meses o años después si se cambió algo del modelo del negocio, se necesita entender el funcionamiento del módulo para poder modificarlo.

Las actividades de soporte deben iniciar con una solicitud por parte del usuario y finalizar con la conformidad del mismo con respecto a los cambios. Durante el soporte por errores el desarrollador debe identificar la falla del módulo, para ello es recomendado implementar un sistema de logs que registre en un archivo físico los eventos inesperados del sistema. A partir de los logs y de los pasos exactos para reproducir el error, el desarrollador ejecuta una solución y realiza pruebas unitarias y de integración para validar que el error no ocurra. Finalmente se libera a producción el módulo corregido y se espera retroalimentación de parte de los usuarios finales.

Si el soporte ocurre debido a cambios en la lógica o modelo del negocio, se implementan las modificaciones y se realiza nuevamente pruebas unitarias y de integración antes de liberarse a producción.

3.4 Validación

Para comprobar el proceso de implementación a continuación se muestra cómo se ejecutó y cuál fue el resultado final. Se mostrará el módulo que gestiona el listado,

creación y edición de un usuario del sistema a partir de los servicios web e interfaz gráfica que lo componen.

3.4.1 Modelo

Inicialmente se diseñó la capa del Modelo compuesta por una clase Usuario, una clase DAO y una clase Model. Se creó un objeto Usuario que contiene la estructura de lo que es un usuario del sistema y los métodos *getters* y *setters* para cada una de las propiedades.

```
4|
5|  @Entity
6|  @Table(name = "USUARIOS")
7|  public class Usuario implements Serializable {
8|      @Id
9|      private String username;
10|     private String password;
11|     @Column(name = "fecha_creacion")
12|     @Temporal(javax.persistence.TemporalType.TIMESTAMP)
13|     private Date fechaCreacion;
14|     @Column(name = "ultimo_acceso")
15|     @Temporal(javax.persistence.TemporalType.DATE)
16|     private Date ultimoAcceso;
17|     @Column(name = "estado_sesion")
18|     private String estadoSesion;
19|     @Column(name = "rol")
20|     private String rol;
21|     private String nombre;
22|     private String email;
23|
24|     public Usuario() {
25|     }
26|
27|     public Usuario(String username, String password, Date
    fechaCreacion, Date ultimoAcceso, String estadoSesion, String rol,
    String nombre, String email) {
28|         this.username = username;
29|         this.password = password;
30|         this.fechaCreacion = fechaCreacion;
31|         this.ultimoAcceso = ultimoAcceso;
32|         this.estadoSesion = estadoSesion;
33|         this.rol = rol;
34|         this.nombre = nombre;
35|         this.email = email;
36|     }
37|     @PrePersist
38|     protected void onCreate() {
39|         if (this.fechaCreacion == null) {
40|             this.fechaCreacion = new Date();
41|         }
42|     }
43|     @Override
44|     public String toString() {
45|         return "Usuario{" + "username=" + username + ", password=" +
    password + ", fechaCreacion=" + fechaCreacion + ", ultimoAcceso=" +
```

```
    ultimoAcceso + ", estadoSesion=" + estadoSesion + ", rol=" + rol + ",
    nombre=" + nombre + ", email=" + email + '}';
46|     }
47|
48|     public String getNombre() {
49|         return nombre;
50|     }
51|
52|     public void setNombre(String nombre) {
53|         this.nombre = nombre;
54|     }
55|
56|    /*Demás Getters y Setters */
57|
58| }
```

Luego de creado el Usuario, se creó una clase DAO encargada de listar la información y de gestionar la persistencia de los cambios en un objeto Usuario. Esta clase se puede inicializar con los objetos de conexión a la base de datos y tiene métodos para crear, actualizar, eliminar y listar usuarios.

```
1| public class UsuarioDAO {
2|
3|     private static final Logger LOG =
    Logger.getLogger(UsuarioDAO.class.getName());
4|
5|     private Session session;
6|     private Transaction trns;
7|
8|     public UsuarioDAO(Session session, Transaction trns) {
9|         this.session = session;
10|        this.trns = trns;
11|    }
12|
13|    public Session getSession() {
14|        return session;
15|    }
16|
17|    public void setSession(Session session) {
18|        this.session = session;
19|    }
20|
21|    public Transaction getTrns() {
22|        return trns;
23|    }
24|
25|    public void setTrns(Transaction trns) {
26|        this.trns = trns;
27|    }
28|
29|    /**
30|     * CREA UN USUARIO NUEVO (NO HACE COMMIT)
31|     *
32|     * @param user Usuario a crear
33|     */
```

```
34|     public void crearUsuario(Usuario user) {
35|         try {
36|             session.save(user);
37|         } catch (RuntimeException e) {
38|             if (trns != null) {
39|                 trns.rollback();
40|             }
41|             LOG.log(Level.SEVERE, null, e);
42|             throw new HibernateException(e);
43|         }
44|     }
45|
46|     /**
47|      * LISTA TODOS LOS USUARIO
48|      *
49|      * @return Lista con todos los Usuarios
50|      */
51|     public List<Usuario> getUsuarios() {
52|         List<Usuario> users = new ArrayList<>();
53|         try  {
54|             users = session.createQuery("from Usuario").list(); //Usa
   nombre de clase en vez de tabla
55|         } catch (RuntimeException e) {
56|             if (trns != null) {
57|                 trns.rollback();
58|             }
59|             LOG.log(Level.SEVERE, null, e);
60|             throw new HibernateException(e);
61|         }
62|         return users;
63|     }
64|
65|      /*Demás operaciones CRUD */
66| }
```

Finalmente se construyó la clase Model encargada de crear la conexión a la base de datos y de la lógica del negocio para todo el módulo encargado de la gestión de usuarios. Cada uno de los métodos públicos del Model es uno de los servicios web del módulo de Gestión de Usuarios y es llamado por la capa del controlador.

```
1| public class su010_gu111Model {
2|
3|     private static final Logger LOG =
   Logger.getLogger(su010_gu111Model.class.getName());
4|
5|
6|     /**
7|      * Edita un usuario con los parametos dados.
8|      *
9|      * @param user Usuario
10|     * @return true si lo edita, false de lo contrario.
11|     */
12|    public boolean editarUsuario(Usuario user) {
13|        SessionFactory sessionFactory = Conexion.getSessionFactory();
14|        try (Session session = sessionFactory.getCurrentSession()) {
15|            Transaction transaction = session.beginTransaction();
```

```
16|             UsuarioDAO dao = new UsuarioDAO(session, transaction);
17|             dao.actualizarUsuario(user);
18|             transaction.commit();
19|             Conexion.closeSessionAndUnbindFromThread();
20|             return true;
21|         } catch (HibernateException | NullPointerException ex) {
22|             LOG.log(Level.SEVERE, null, ex);
23|             return false;
24|         }
25|     }
26|
27|     /**
28|      * Crea un nuevo usuario en la tabla Usuarios
29|      *
30|      * @param user Usuario a crear
31|      * @return true si lo crea, false de lo contrario.
32|      */
33|     public boolean crearUsuario(Usuario user) {
34|         SessionFactory sessionFactory = Conexion.getSessionFactory();
35|         try (Session session = sessionFactory.getCurrentSession()) {
36|             Transaction transaction = session.beginTransaction();
37|             UsuarioDAO dao = new UsuarioDAO(session, transaction);
38|             dao.crearUsuario(user);
39|             transaction.commit();
40|             Conexion.closeSessionAndUnbindFromThread();
41|             return true;
42|         } catch (HibernateException | NullPointerException ex) {
43|             LOG.log(Level.SEVERE, null, ex);
44|             return false;
45|         }
46|
47|     }
48|
49|     /* Métodos para los demás servicios */
50|
51| }
```

3.4.2 Vista

La interfaz gráfica del módulo es sencilla, se diseñó una tabla que contiene la información básica de los usuarios del sistema con una serie de botones para las diferentes acciones que se pueden realizar.

```
1| <div ng-controller="MainController as mn">
2|     <div class="col-xs-10 col-xs-push-1">
3|         <div class="panel panel-default">
4|             <div class="panel-heading">
5|                 <h3 class="text-uppercase"><b>Gesti&oacute;n de
   usuarios</b></h3>
6|             </div>
7|             <div class='panel-body'>
8|                 <div class="row">
9|                     <div class="col-sm-6 form-inline">
10|                        <label>Mostrar <select class="input-sm form-
   control" ng-model="mn.pagina" ng-readonly="true">
```

```
11|                                <option ng-repeat="page in mn.pages"
   value="{{page.value}}">{{page.value}}</option>
12|                            </select> registros
13|                        </label>
14|                    </div>
15|                    <div class="col-sm-6 form-inline" style="text-
   align: right">
16|                        <label>Buscar: <input type="text" ng-
   model="query" class="form-control input-sm"/></label>
17|                    </div>
18|                </div>
19|                <table id="table" name="Usuarios" search-watch-
   model="query" st-table='mn.displayedCollection' st-safe-
   src='mn.respuesta' class='table table-bordered table-condensed table-
   hover table-responsive table-striped'>
20|                    <thead>
21|                        <tr>
22|                            <th class='text-center vertical-center'
   st-sort='username'>Usuario</th>
23|                            <th class='text-center vertical-center'
   st-sort='estadoSesion'>Estado</th>
24|                            <th class='text-center vertical-center'
   st-sort='nombre'>Nombre</th>
25|                            <th class='text-center vertical-center'
   st-sort='email'>E-Mail</th>
26|                            <th class='text-center vertical-center'
   st-sort='rol'>Rol</th>
27|                            <th class='text-center vertical-center'
   st-sort='fechaCreacion'>Fecha Creación</th>
28|                            <th class='text-center vertical-center'
   st-sort='ultimoAcceso'>Ultimo Ingreso</th>
29|                            <th class='text-center'></th>
30|                            <th class='text-center'></th>
31|                    </thead>
32|                    <tbody>
33|                        <tr ng-repeat='res in
   mn.displayedCollection'>
34|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.username}}</td>
35|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.estadoSesion}}</td>
36|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.nombre|uppercase}}</td>
37|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.email|lowercase}}</td>
38|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.rol}}</td>
39|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.fechaCreacion|date:"EEEE,
   d MMMM, y"}}</td>
40|                            <td class='col-xs-2 text-center vertical-
   center' style="vertical-align:middle">{{res.ultimoAcceso| date:"EEEE,
   d MMMM, y"}}</td>
41|                            <td class='col-xs-1'>
42|                                <button type='button' ng-
   click='mn.editatElemento(res)' class='btn btn-sm btn-success'>
43|                                    <i class='glyphicon glyphicon-
   edit'></i> Editar Usuario
44|                                </button>
45|                            </td>
46|                            <td class='col-xs-1'>
```

```
47|                                 <button type='button' ng-
   click='mn.eliminar(res)' class='btn btn-sm btn-danger'>
48|                                     <i class='glyphicon glyphicon
   glyphicon-remove'></i> Eliminar Usuario
49|                                 </button>
50|                             </td>
51|                         </tr>
52|                     </tbody>
53|                     <tfoot>
54|                         <tr>
55|                             <td colspan="9" class="text-center">
56|                                 <div st-pagination="" st-items-by-
   page="mn.pagina" st-displayed-pages="7"></div>
57|                             </td>
58|                         </tr>
59|                     </tfoot>
60|                 </table>
61|                 <div class='panel-body'>
62|                     <div class='right'>
63|                         <button type='button' ng-
   click='mn.insertarElemento()' class='btn btn-raised btn-primary'>
64|                             <i class='glyphicon glyphicon-plus'></i>
   Crear Usuario Nuevo
65|                         </button>
66|                     </div>
67|                 </div>
68|             </div>
69|         </div>
70|     </div>
71| </div>
```

Además del HTML de la tabla, se crearon dos modales para los formularios de crear y actualizar datos de usuario.

```
1|  <div class="row">
2|      <div class="col-lg-12">
3|          <div class="tabs-container">
4|              <uib-tabset>
5|                  <uib-tab>
6|                      <uib-tab-heading>
7|                          Actualizar Datos del Usuario
   {{selected.username}}
8|                      </uib-tab-heading>
9|                      <div class="panel-body">
10|                         <form name="datosUsuario">
11|                             <fieldset>
12|                                 <input-text titulo="Nombre"
   valor="selected.nombre" col="6" readonly="false"
   required="true"></input-text>
13|                                 <input-text titulo="Email"
   valor="selected.email" col="6" readonly="false"
   required="true"></input-text>
14|                                 <select-datos titulo="Rol"
   valor="selected.rol" datos="mn.roles" col="6" required="true"
   value="nombre" label="descripcion"></select-datos>
15|                             </fieldset>
16|                             <div class="col-xs-12 text-center">
```

```
17|                                  <button class="btn btn-raised btn-
    success" ng-click="guardar(datosUsuario)">Editar</button>
18|                                  <button class="btn btn-raised btn-
    info" type="submit" ng-
    click="cancel()"><strong>Cerrar</strong></button>
19|                                </div>
20|                            </form>
21|                        </div>
22|                    </uib-tab>
23|                </uib-tabset>
24|            </div>
25|        </div>
26|</div>
```

3.4.3 Controlador

Las funciones del controlador están divididas en dos partes. Una parte está escrita en Java encargada de controlar qué páginas se cargan y qué servicios se tienen disponibles en cada módulo. La otra parte está escrita en Javascript y controla los eventos producidos por el usuario, el momento de mostrar la información en pantalla y la llamada a los servicios web.

El controlador en Java se implementó en una clase Controller que tiene dos métodos públicos. El primero es llamado *usuario* y recibe una petición HTTP bajo el verbo **GET**, analiza si el usuario tiene permisos para acceder al contenido y responde con la el documento solicitado.

```
1| @Stateless
2| @Path("")
3| public class su010_gu111Controller {
4|
5|     Autorizacion auth;
6|     LoginModel user;
7|     su010_gu111Model model;
8|     Gson gson;
9|
10|    public su010_gu111Controller() {
11|        auth = new Autorizacion();
12|        user = new LoginModel();
13|        model = new su010_gu111Model();
14|        gson = new GsonBuilder()
15|                .setDateFormat("yyyy-MM-dd")
16|                .setPrettyPrinting()
17|                .serializeNulls()
18|                .create();
19|    }
20|
21|    @GET
22|    @Path("{tipo}")
23|    public Response usuarios(
```

```
24|             @Context HttpServletRequest request,
25|             @PathParam("tipo") String tipo
26|     ) throws URISyntaxException {
27|         for (Cookie cookie : request.getCookies()) {
28|             System.out.println(cookie.getValue());
29|         }
30|         if (auth.verificarPermisos(request.getCookies(),
    Sets.newHashSet("ALL"))) {
31|             switch (tipo) {
32|                 case "principal":
33|                     return Response.ok(new
    Viewable("/principal")).build();
34|                 case "modal_insertar":
35|                     return Response.ok(new
    Viewable("/modal_insertar"))
36|                                     .build();
37|                 case "modal_editar":
38|                     return Response.ok(new Viewable("/modal_editar"))
39|                                     .build();
40|                 case "modal_ver":
41|                     return Response.ok(new
    Viewable("/modal_ver")).build();
42|                 case "script.js":
43|                     return Response.ok(new
    Viewable("/script.js")).build();
44|                 default:
45|                     return Response.status(204).build();
46|             }
47|
48|         }
49|         return Response.ok()
50|
          .entity(gson.toJson(auth.accesoDenegado())).build();
51|     }
52| }
```

El otro método de la clase Controller es denominado *funciones* y se encarga de responder con un JSON a los servicios llamados. Este método recibe una petición HTTP tipo **POST**, junto con algunos parámetros, valida que el usuario tenga acceso a los servicios y responde de acuerdo a lo solicitado.

```
1|  @POST
2|  @Path("funciones/{metodo}")
3|  public Response funciones(
4|          @Context HttpServletRequest request,
5|          @PathParam("metodo") String metodo,
6|          @FormParam("USUARIO") String usuario,
7|          @FormParam("CLAVE") String clave,
8|          @FormParam("NOMBRE") String nombre,
9|          @FormParam("ESTADO") String estado,
10|         @FormParam("CODARE") String codare,
11|         @FormParam("NRODOC") String nrodoc,
12|         @FormParam("EMAIL") String email,
13|         @FormParam("ROL") String rol,
14|         @FormParam("DURACION") String duracion
15| ) throws URISyntaxException, SQLException, ClassNotFoundException {
16|     if (auth.verificarPermisos(request, 111)) {
```

```
17|         Usuario usr;
18|         String username = auth.obtenerUsername(request);
19|         switch (metodo) {
20|             case "listar":
21|                 return
   Response.ok().entity(model.listarUsuarios()).build();
22|             case "roles":
23|                 return
   Response.ok().entity(model.listarRoles()).build();
24|             case "borrar":
25|                 if (model.borrarUsuario(usuario)) {
26|                     JsonObject o = new JsonParser()
27|                                 .parse("{\"MENSAJE\":\"SE HA BORRADO
   AL USUARIO CORRECTAMENTE\"}").getAsJsonObject();
28|                     return
   Response.ok().entity(gson.toJson(o)).build();
29|                 } else {
30|                     JsonObject o = new JsonParser()
31|                                 .parse("{\"MENSAJE\":\"HA OCURRIDO UN
   ERROR AL BORRAR AL USUARIO.\"}").getAsJsonObject();
32|                     return
   Response.ok().entity(gson.toJson(o)).build();
33|                 }
34|/*Casos de editar, cambiar clave y agregar usuarios omitidos por
   brevedad*/
35|             default:
36|                 return Response.ok().entity("403").build();
37|         }
38|     }
39|     return Response.status(Response.Status.FORBIDDEN).build();
40|}
```

En el controlador de Javascript lo primero que se hizo fue crear el módulo de Angular y crear un interceptor para que toda petición que salga del módulo tenga un encabezado con la autorización.

```
1| angular.module('caos', [
2|     'smart-table',
3|     'ngCsv',
4|     'oitozero.ngSweetAlert',
5|     'ngSanitize',
6|     'ui.bootstrap',
7|     'ngAnimate'
8| ]).controller('MainController',
   MainController).factory('sessionInjector', function () {
9|     var sessionInjector = {
10|         request: function (config) {
11|             var cookie = getCookie("LOGIN").replace(new RegExp('"',
   'g'), '');
12|             config.headers['Authorization'] = cookie;
13|             return config;
14|         }
15|     };
16|     return sessionInjector;
17| }).config(['$httpProvider', function ($httpProvider) {
18|         $httpProvider.interceptors.push('sessionInjector');
19|     }]);
```

Luego se creó el controlador principal, este se ejecuta apenas se inicialice el módulo. Se encarga de construir el estado inicial de la página para ello primero se crea una función que hace una petición HTTP al servidor y que retorna una promesa. La petición busca consumir el servicio para listar a todos los usuarios del sistema y una vez sea resuelta de manera positiva, asigna la respuesta a una variable de Javascript y la muestra en pantalla automáticamente gracias a una característica de Angular llamado Enlace de datos de dos vías.

```
1| function MainController($scope, $q, $http, $uibModal, SweetAlert,
   Notificar) {
2|     'use strict';
3|     var vm = this;
4|     $scope.selected = {};
5|
6|      /**
7|      * Lista los usuarios de la plataforma
8|      * @returns {$q@call;defer.promise}
9|      */
10|    function listarUsuarios() {
11|        var defered = $q.defer(), promise = defered.promise;
12|        $http({
13|            cache: true,
14|            method: 'POST',
15|            url: '/su010_gu111/funciones/listar',
16|            headers: {'Content-Type': 'application/x-www-form-
   urlencoded'}
17|        }).success(function (data) {
18|            defered.resolve(data);
19|        }).error(function (err) {
20|            defered.reject(err);
21|        });
22|        return promise;
23|    }
24|
25|    /**
26|     * LISTA TODOS LOS ROLES EXISTENTES
27|     * @returns {$q@call;defer.promise}
28|     */
29|    function listarRoles() {
30|        var defered = $q.defer();
31|        var promise = defered.promise;
32|        $http({
33|            cache: true,
34|            method: 'POST',
35|            url: '/su010_gu111/funciones/roles',
36|            headers: {'Content-Type': 'application/x-www-form-
   urlencoded'}
37|        }).success(function (data) {
38|            defered.resolve(data);
39|        }).error(function (err) {
40|            defered.reject(err);
41|        });
42|        return promise;
43|    }
44|
```

```
45|     /**
46|      * Elimina a un usuario
47|      * @param {type} usuario
48|      * @returns {$q@call;defer.promise}
49|      */
50|     function eliminarUsuario(usuario) {
51|         var defered = $q.defer();
52|         var promise = defered.promise;
53|         $http({
54|             cache: true,
55|             method: 'POST',
56|             url: '/su010_gu111/funciones/borrar',
57|             data: $.param({USUARIO: usuario}),
58|             headers: {'Content-Type': 'application/x-www-form-
   urlencoded'}
59|         }).success(function (data) {
60|             defered.resolve(data);
61|         }).error(function (err) {
62|             defered.reject(err);
63|         });
64|         return promise;
65|     }
66|
67|     vm.pagina = 10;
68|     vm.pages = [{value: 10}, {value: 20}, {value: 50}, {value: 100}];
69|     vm.respuesta = [];
70|     vm.roles = [];
71|
72|     listarUsuarios().then(function (json) {
73|         vm.respuesta = json;
74|     });
75|     listarRoles().then(function (json) {
76|         vm.roles = json;
77|     });
78| }
```

Luego de listar y mostrar la información, se define la respuesta de la página al recibir un evento de clic sobre cada uno de los botones. La página se encargará de mostrar un modal cuando se necesite crear un usuario nuevo y cuando se necesite actualizarlo, cuando se intente eliminar el usuario tan solo se mostrará una alerta para confirmación.

```
1| vm.editatElemento = editatElemento;
2| function editatElemento(res) {
3|     $scope.selected = res;
4|     var modalInstance = $uibModal.open({
5|         templateUrl: "./modal_editar",
6|         controller: usuarioModalController,
7|         scope: $scope,
8|         size: 'lg',
9|         windowClass: "animated fadeIn"
10|     });
11| }
12| vm.insertarElemento = insertarElemento;
13| function insertarElemento() {
14|     $scope.selected = {};
15|     $uibModal.open({
16|         templateUrl: "./modal_insertar",
17|         controller: usuarioModalController,
```

```
        scope: $scope,
        size: 'lg',
        windowClass: "animated fadeIn"
    });
}
vm.eliminar = eliminar;
function eliminar(res) {
    SweetAlert.swal({
        title: "¡Atención!",
        text: "Se va a eliminar este usuario",
        type: "warning",
        showCancelButton: true,
        confirmButtonColor: "#DD6B55",
        confirmButtonText: "OK!",
        cancelButtonText: "Cancelar",
        closeOnConfirm: true},
    function (isConfirm) {
        if (isConfirm) {
            eliminarUsuario(res.username).then(function (json) {
                if (json.MENSAJE === "HA OCURRIDO UN ERROR AL BORRAR AL USUARIO.") {
                    Notificar.error();
                } else {
                    listarUsuarios().then(function (json) {
                        vm.respuesta = json;
                    });
                }
            });
        }
    });
}
```

Finalmente se crea un nuevo controlador para los modales que se encarga de controlar los eventos que sucedan dentro del modal, principalmente el llamado a los servicios web para insertar y actualizar al usuario.

```
function usuarioModalController($scope, $uibModalInstance, Notificar) {

    function editarUsuario(nombre, email, rol, usuario) {
        var defered = $q.defer();
        var promise = defered.promise;
        $http({
            cache: true,
            method: 'POST',
            url: '/su010_gu111/funciones/editar',
            data: $.param({NOMBRE: nombre, EMAIL: email, ROL: rol, USUARIO: usuario}),
            headers: {'Content-Type': 'application/x-www-form-urlencoded'}
        }).success(function (data) {
            defered.resolve(data);
        }).error(function (err) {
            defered.reject(err);
        });
        return promise;
    }

```

```
20|    function agregarUsuario(nombre, email, rol, usuario, clave) {
21|        var defered = $q.defer();
22|        var promise = defered.promise;
23|        $http({
24|            cache: true,
25|            method: 'POST',
26|            url: '/su010_gu111/funciones/agregar',
27|            data: $.param({NOMBRE: nombre, EMAIL: email, ROL: rol,
   USUARIO: usuario, CLAVE: clave}),
28|            headers: {'Content-Type': 'application/x-www-form-
   urlencoded'}
29|        }).success(function (data) {
30|            defered.resolve(data);
31|        }).error(function (err) {
32|            defered.reject(err);
33|        });
34|        return promise;
35|    }
36|
37|    $scope.guardar = function (form) {
38|        if (form.$valid) {
39|            editarUsuario($scope.selected.nombre,
   $scope.selected.email, $scope.selected.rol,
   $scope.selected.username).then(function (json) {
40|                if (json.MENSAJE === "SE HA EDITADO AL USUARIO
   CORRECTAMENTE") {
41|                    listarUsuarios().then(function (json) {
42|                        vm.respuesta = json;
43|                    });
44|                } else {
45|                    Notificar.error();
46|                }
47|            }, function (error) {
48|                console.log(error);
49|            });
50|            $uibModalInstance.close();
51|            $scope.selected = {};
52|        } else {
53|            Notificar.required(form.$error);
54|        }
55|    };
56|
57|    $scope.registrar = function (form) {
58|        if (form.$valid) {
59|            agregarUsuario($scope.selected.nombre,
   $scope.selected.email,
60|                    $scope.selected.rol, $scope.selected.username,
61|                    $scope.selected.password)
62|                    .then(function (json) {
63|                        if (json.MENSAJE === "SE HA CREADO AL USUARIO
   CORRECTAMENTE") {
64|                            listarUsuarios().then(function (json) {
65|                                vm.respuesta = json;
66|                            });
67|                        } else {
68|                            Notificar.error();
69|                        }
70|                    }, function (error) {
71|                        console.log(error);
72|                    });
73|            $uibModalInstance.close();
```

```
74|             $scope.selected = {};
75|         } else {
76|             Notificar.required(form.$error);
77|         }
78|     };
79|
80|     $scope.cancel = function () {
81|         $uibModalInstance.dismiss('cancel');
82|     };
83| }
```

Para finalizar, el código fuente completo de este y otro módulo, las clases compartidas, el contenedor donde se ejecutan y el script de la base de datos están publicados en un repositorio libre de GitHub bajo licencia MIT en la dirección https://github.com/LordShiroe/EnergyCorp.

4. Conclusiones y recomendaciones

4.1 Conclusiones

El patrón de diseño MVC permite dividir el sistema en tres capas separadas permitiendo un bajo acoplamiento. Cada capa es una entidad que puede ser cambiada o modificada sin afectar a la totalidad del sistema. En web el MVC se ve implementado de diferentes formas, dependiendo del Framework con el que se trabaje, esto es especialmente cierto en el Frontend.

Con el interés de diseñar una aplicación MVC web a partir de una aplicación de legado con una arquitectura diferente, es necesario realizar una serie de pasos y análisis para dividir el problema y sacarle el máximo partido a la nueva arquitectura, de modo tal que se puedan suplir las falencias de la arquitectura anterior.

4.2 Recomendaciones

Se recomienda realizar una actualización a todos los servicios de Enelar para hacerlos RESTful. Dicho proceso toma bastante tiempo y se debe realizar de forma gradual pero hacerlo asegura facilidad de lectura, uso y modificación de los servicios diseñados, esto es importante para el soporte y mantenimiento de cualquier sistema de información.

Bibliografía

[1] G. Alor-Hernandez, V. Rosales-Morales, and L. O. Colombo-Mendoza, “Basic Concepts on RIAs,” in *Application Development and Design: Concepts, Methodologies, Tools and Applications*, First., Information Resource Management Association, 2017, p. 1611.

[2] A. Osmani, *Learning Javascript Desingn Patterns*. O’reilly Media, 2012.

[3] colaboradores de Wikipedia, “Smalltalk.” [Online]. Available: https://es.wikipedia.org/w/index.php?title=Smalltalk&oldid=90177048. [Accessed: 14-Jan-2017].

[4] M. Fowler, “GUI Architectures,” 2006. [Online]. Available: https://martinfowler.com/eaaDev/uiArchs.html.

[5] colaboradores de Wikipedia, “Front-end y back-end,” *Wikipedia*. .

[6] M. Godbolt, *Frontend Architecture for Design Systems*, First Edit. O’Reilly Media, Inc., 2016.

[7] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[8] Mozilla Foundation, “Javascript,” *12 de Mayo*, 2016. [Online]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript.

[9] colaboradores de Wikipedia, “AngularJS.” [Online]. Available: https://es.wikipedia.org/w/index.php?title=AngularJS&oldid=93940329. [Accessed: 20-Jun-2010].

[10] colaboradores de AngularJs, “Understanding Controllers.” [Online]. Available: https://docs.angularjs.org/guide/controller. [Accessed: 12-Apr-2017].

[11] Tutorialpoints, “AngularJS - Services.” [Online]. Available: https://www.tutorialspoint.com/angularjs/angularjs_services.htm.

[12] Oracle Corporation, “Connecting with DataSource Objects.” [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html. [Accessed: 14-Apr-2017].

[13] colaboradores de Wikipedia, “Data access object,” *Wikipedia*. Wikipedia, La enciclopedia libre.

[14] colaboradores de Wikipedia, “Plain Old Java Object,” *Wikipedia*. .

[15] Elena, J. F. Gir, and G. Rossi, “Testing de migración de aplicaciones distribuidas a entornos Web,” Universidad Nacional de la Plata, 2007.

[16] B. Meyer, *CompositeIStructured Design*. New York: Van Nostrand Reinhold, 1978.

[17] C. Fontela, “Modularidad, cohesión y acoplamiento,” 2009. [Online]. Available: https://cysingsoft.wordpress.com/2009/06/23/modularidad-cohesion-y-acoplamiento-carlos-fontela/. [Accessed: 20-May-2017].

[18] colaboradores de Wikipedia, “Javadoc,” *Wikipedia*. Wikipedia, La enciclopedia libre., 2015.

[19] colaboradores de Wikipedia, “Prueba unitaria,” *Wikipedia*. Wikipedia, La enciclopedia libre., 2016.

[20] C. Rada, “Las pruebas de software y el aseguramiento de calidad,” 2014. [Online]. Available: https://camilorada.wordpress.com/tag/pruebas-integracion/. [Accessed: 05-Jun-2017].