



***Facultad
de
Ciencias***

**Especificación Gráfica de Procesos de
Recuperación de Datos en LUCA**
(Graphic Specification of Data Recovery
Processes in LUCA)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Ignacio Agüero Salcines

Director: Pablo Sánchez Barreiro

Co-Director: Sergio Herrera Iglesias

Febrero - 2018

Agradecimientos

Me gustaría dar agradecimientos a mi familia y facultad, ya que sin ellos esto no habría sido posible nada de esto.

Es importante agradecer también a CIC Consulting Informático por permitirme la oportunidad de realizar el desarrollo del proyecto en su empresa, sin olvidarme de mis compañeros de LUCA, que han sido un gran apoyo durante el mismo.

Para finalizar, me gustaría agradecer a mi mentor Pablo, por guiarme durante el desarrollo del proyecto con eficacia y ayudarme a afrontar este trabajo de fin de grado.

Resumen

Las empresas actuales no utilizan un único sistema de información que de soporte a sus procesos de trabajo, sino un ecosistema de sistemas información que dan soporte a diferentes procesos de negocio ejecutados dentro de dicha organización. Como consecuencia de esta nueva situación, cuando un usuario quiere obtener una información concreta cuyos datos residen en varios de estos sistemas, necesita acceder a cada uno de estos sistemas, extraer de cada sistema la información que precisa, filtrarla y unificarla para finalmente obtener los datos requeridos.

Por ejemplo, una tienda de electrodomésticos podría tener sistemas informáticos diferentes para el departamento de atención al cliente, para el departamento técnico de postventa y para el departamento de compras y adquisiciones. Por tanto, para conocer el estado actual de una reparación, podríamos necesitar:

- Acceder al primer sistema para obtener el identificador de la incidencia y en qué fase de su gestión se encuentra.
- Comprobado que la incidencia está actualmente en reparación, recuperaríamos otro sistema el estado detallado de la reparación, comprobando que está a la espera de una pieza.
- Finalmente accederíamos al sistema de compra y adquisiciones para comprobar cuando está prevista la entrega de dicha pieza. Los sistemas de almacenamiento de la información pueden ser diversos, incluyendo desde un servicio web, una base de datos relacional, un repositorio de ficheros accesible vía FTP o una base de datos NoSQL.

El objetivo de este proyecto es facilitar dicho proceso de composición al usuario mediante el desarrollo de un mecanismo gráfico para la especificación de estos procesos de composición de consultas.

Palabras clave Recuperación de la Información, Especificación de Consultas, Fuentes de Datos Distribuidas, Vaadin, GoJS.

Preface

Currently, companies rarely use a single software system to support their business processes. Instead, several software systems, typically one per department or business unit, are used. As a consequence of this situation, when a user wants to retrieve a piece of information whose data are stored across these systems, she would need to access each one of these systems; extract the required information from each system; filter this information; and, finally, unify it to obtain the appropriate data.

For example, an appliance store could have different information systems: one for the customer service department; one for technical assistance department; and one for the purchasing and procurement department. Therefore, to know the current status of a repair, it might be needed:

- To access the customer service system to get the identifier of the identifier of the repair order and to know at what stage of the repair process this order is.
- After checking that order is currently being processed, it might be needed to access the information system of the technical assistance department to find the details of the repair status. In this case, it might be discovered that the technicians are waiting for a component to complete the repair.
- Finally, to know when this component is expected to be delivered, we would need to access the information system of the purchase and acquisition department.

Each one of these systems may store its data into a different kind of system, ranging from a web service to a relational database, a file repository accessible via FTP or a NoSQL database, among other options. The objective of this project is to facilitate this process of composition to the user through the development of a graphic mechanism for the specification of these processes of composition of queries.

Keywords: Information Retrieval, Specification of Queries, Distributed Data Sources, Vaadin, GoJS.

Índice general

1. Introducción	1
1.1. Introducción	1
1.2. LUCA	2
1.2.1. Motivación	2
1.2.2. Funcionamiento de LUCA	3
1.2.3. Limitaciones actuales de LUCA	6
1.3. Objetivos del Trabajo de Fin de Grado	6
1.4. Alcance del Proyecto	7
1.5. Sumario	8
2. Antecedentes	9
2.1. Go.JS	9
2.2. Patrón <i>Model-View-Presenter (MVP)</i>	16
2.3. Vaadin	16
2.3.1. Desarrollo de Aplicaciones Web con Vaadin	17
2.3.2. Arquitectura Interna de <i>Vaadin</i>	20
2.4. Arquitectura LUCA	22
2.5. Sumario	23
3. Desarrollo del Proyecto	24
3.1. Introducción	24
3.2. Desarrollo del Editor Gráfico	26
3.3. Integración del editor con <i>Vaadin</i>	28
3.4. Integración del editor con LUCA	30
3.5. Gestión, Ejecución y Persistencia de los Procesos	32
3.5.1. Ejecución de Procesos	36
3.6. Sumario	37
4. Sumario, Conclusiones y Trabajos Futuros	38
4.1. Sumario	38
4.2. Resultados	39
4.3. Conclusiones	39
4.4. Trabajos Futuros	40

A. Tratamiento de eventos de la clase Diagram Component	41
---	----

Índice de figuras

1.1. Menu Principal LUCA	3
1.2. Vista de Gestión de Consultas	4
1.3. Creación de una Consulta	4
1.4. Vista de Ejecución de Consultas	5
2.1. Editor gráfico creado en GoJS	10
2.2. Asignación de la variable \$	10
2.3. Creación de un diagrama en GoJS	11
2.4. Declaración del patrón del Nodo	12
2.5. Función MakePort	13
2.6. Patrón Nodo con Puertos	14
2.7. Declaración del patrón del Link	14
2.8. Creación de la Paleta de Nodos	15
2.9. Patrón Modelo-Vista-Presentador	16
2.10. Árbol de Proyectos	17
2.11. Interfaz de Usuario Vaadin	18
2.12. Contenedor TreeGrid	18
2.13. Contenedor TreeGrid Collapsible	19
2.14. Contenedor TreeGrid Measurable	19
2.15. Arquitectura Interna Vaadin	20
2.16. Arquitectura de LUCA	22
3.1. Flujo de Desarrollo	24
3.2. Primer Contacto Editor	25
3.3. Template Prueba Editor Inicial	26
3.4. Link Tool	27
3.5. Componente Vaadin	28
3.6. Conector	29
3.7. Ejemplo de vista	30
3.8. Ejemplo de presentador	31
3.9. Modelo de Datos	32
3.10. Interfaz controladora	33
3.11. Implementación interfaz controladora	34
3.12. Fichero SQL	34

3.13. Ejemplo Test	35
3.14. Creación de un proceso	35
3.15. Ejecución de un proceso	36
3.16. Ejemplo Servicio de ejecución	37
A.1. Evento de elemento movido	41
A.2. Evento de edición de texto	42
A.3. Evento de doble selección	42

Capítulo 1

Introducción

Índice

1.1. Introducción	1
1.2. LUCA	2
1.2.1. Motivación	2
1.2.2. Funcionamiento de LUCA	3
1.2.3. Limitaciones actuales de LUCA	6
1.3. Objetivos del Trabajo de Fin de Grado	6
1.4. Alcance del Proyecto	7
1.5. Sumario	8

1.1. Introducción

Este documento contiene la memoria del Trabajo Fin de Grado realizado por Ignacio Agüero Salcines para la obtención del título de Grado en Ingeniería Informática. Dicho trabajo Trabajo Fin de Grado fue realizado en la empresa CIC Consulting Informático, entre los meses de Agosto y Diciembre de 2017 y 2018. Durante ese periodo el alumno realizó una carga total de trabajo de 500 horas aproximadamente, bajo la supervisión de Sergio Herrera Iglesias dentro del Departamento de Movilidad.

El presente trabajo se enmarca concretamente dentro del proyecto LUCA. El objetivo general de dicho proyecto es proporcionar una interfaz de acceso uniforme a un conjunto de fuentes de datos heterogéneas. El proyecto LUCA, el cual se describirá en mayor profundidad en la siguiente sección, comenzó en 2015, está actualmente en fase de producción con la versión 2.1.4 activa, está distribuida en cinco clientes de sectores tan diversos como la administración pública o la banca.

Dado que para entender los objetivos de este proyecto es necesario antes conocer cuáles son los objetivos de LUCA y cómo funciona dicha herramien-

ta, la siguiente sección proporciona una breve introducción a la herramienta LUCA.

1.2. LUCA

1.2.1. Motivación

En los últimos años, el volumen de datos recogidos y manipulados por las empresas ha aumentado de forma vertiginosa. Estos datos se han ido almacenando en diferentes tipos de fuentes conforme las empresas crecían y sus sistemas evolucionaban y se fusionaban. Como resultado de este proceso no es extraño actualmente encontrar empresas que tengan sus datos almacenados en sistemas tan dispares como bases de datos relacionales, hojas XML o repositorios FTP.

Como consecuencia de esta nueva situación, cuando un usuario quiere obtener una información concreta cuyos datos residen en varios de estos sistemas, éste necesita acceder a cada uno de estos sistemas, extraer de cada sistema la información que precisa, y finalmente filtrarla y unificarla para finalmente obtener los datos requeridos.

Por ejemplo, una cadena de venta de electrodomésticos podría tener sistemas informáticos diferentes para el departamento de atención al cliente, para el departamento técnico de postventa y para el departamento de compras y adquisiciones. Por tanto, para conocer con precisión el estado actual de una reparación, podríamos necesitar:

1. Acceder al sistema de atención al cliente para obtener el identificador de la incidencia y en qué fase de su gestión se encuentra.
2. Una vez corroborado que la incidencia está actualmente siendo atendida, recuperaríamos del sistema de gestión de reparaciones el estado detallado de la reparación. Como resultado de esta operación, supongamos que averiguamos que la reparación está a la espera de recibir una pieza que se ha de sustituir.
3. Finalmente, para poder hacer una estimación de cuando podría estar lista la reparación, accederíamos al sistema de compra y adquisiciones para averiguar cuando está prevista la entrega de la pieza solicitada.

Como hemos comentado anteriormente, a cada uno de estos sistemas podría accederse de manera diferente. Por ejemplo, el primero podría consultarse utilizando un servicio web. La información del segundo podría recuperarse accediendo directamente a una base de datos relacional, mientras que la información del tercero se obtendría analizando órdenes de compra en formato *pdf* almacenadas en un repositorio de ficheros compartido. Por

tanto, el usuario, para poder realizar este proceso, necesita conocer las particularidades de cada sistema y de su forma de acceso.

Para aliviar esta situación, dentro de la empresa CIC, se está desarrollando una aplicación denominada LUCA, a la cual contribuye este Trabajo Fin de Grado. Para facilitar este proceso de recuperación de información, LUCA proporciona un lenguaje común para todas las fuentes de datos a unificar, permitiendo al usuario abstraerse de los detalles de cada fuente.

1.2.2. Funcionamiento de LUCA

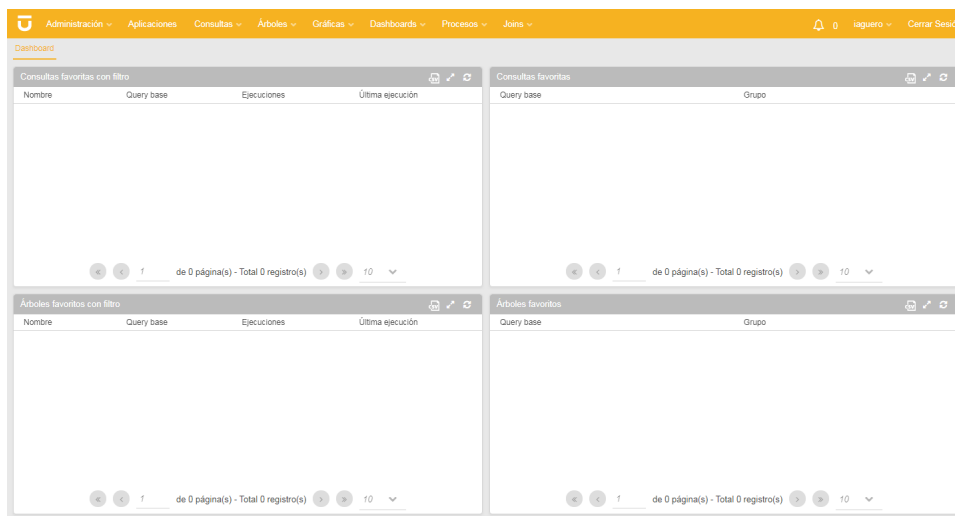


Figura 1.1: Menu Principal LUCA

A continuación, se detalla brevemente el funcionamiento de *LUCA*. Para ello, utilizaremos como ejemplo una consulta a la base de datos de LUCA, esta consulta obtendrá los procesos en función de un estado (los estados pueden ser en edición, publicado, es decir, preparada para ser ejecutada, o borrado).

En el menu principal de LUCA, nos aparece una vista con un conjunto de consultas que el usuario ha marcado como favoritas (Figura 1.1), es decir, que suele utilizar con frecuencia. Como se dijo anteriormente, se utilizará la consulta de los procesos por estado.

CAPÍTULO 1. INTRODUCCIÓN

NOMBRE	DETALLE	ESTADO	SISTEMA	TIPO	GRUPO	USUARIO EDICIÓN	ÚLTIMA EDICIÓN
Combo Estados Proceso	Combo Estados Proceso	Publicada	LUCA	BD MySQL	LUCA_TEST	iaguero	14-02-2018 09:51:44
Datos Usuarios	Datos Usuarios Listable	Publicada	LUCA	BD MySQL	LUCA_TEST	iaguero	14-02-2018 09:52:03
Email Usuario	Obtiene el email del usuario a partir de su username	Edición	LUCA	BD MySQL	LUCA_TEST	iaguero	14-02-2018 09:50:53
Get Procesos Por Estado	Get Procesos Por Un Estado	Publicada	LUCA	BD MySQL	LUCA_TEST	iaguero	14-02-2018 09:51:25

Figura 1.2: Vista de Gestión de Consultas

En la vista de gestión de consultas (Figura 1.2) se pueden realizar las operaciones propias de la gestión, como es crear, modificar, eliminar o ejecutar entre otras.

No obstante, para que dichas consultas pueden ser ejecutadas, es necesario un usuario con conocimientos suficientes para ello, al que denominaremos en adelante el *creador de consultas*.

VARIABLES 1

Clave	Descripción	Tipo	Seleccionable	Multiselección	Opcional	Formato fecha
Estado	Estado del proceso	string	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

SQL

```
1 SELECT lp.NAME,lp.DESCRPTION,aj.NAME as 'AsyncJob',su.USERNAME as 'User' FROM luca.LUCA_PROCESSES lp
2
3 inner join luca.ASYNC_JOBS aj on aj.id = lp.ASYNC_JOB_ID
4 inner join luca.SECURITY_USERS su on su.id = lp.EDITION_USER_ID
5
6 where 1=1
7 #if($Estado)
8 AND lp.PROCESS_STATE_ID=:Estado
9 #end
10
```

Figura 1.3: Creación de una Consulta

Para poder realizar esta tarea, el creador de consultas accedería a la interfaz dedicada a esta tarea (ver Figura 1.3). En esta interfaz, definiría

primero las variables de entrada y salida de la consulta (Figura 1.3, etiqueta 1). A continuación, especificaría cómo llevar a cabo dicha consulta a bajo nivel. Para ello puede utilizar una serie de facilidades y primitivas proporcionadas por LUCA. Para la consulta ejemplo, utilizando estas facilidades, se especifica que se desea obtener el nombre, la descripción, el nombre de la tarea asíncrona y el usuario, y se establece como variable de entrada el estado del proceso.

Tras guardar la consulta se puede ejecutar directamente desde la ventana de creación, sin embargo, si la consulta ya ha sido ejecutada previamente y el usuario la ha publicado (tras ejecutar una consulta desde la ventana de creación, si se ha ejecutado correctamente se le permite al usuario cambiar su estado de edición a publicada, esto significa que la consulta está preparada para ser ejecutada), el usuario puede ir a la ventana de ejecución de consultas, y desde ahí ejecutarla.

La principal ventaja que aporta LUCA es que el proceso de ejecución de consultas es opaco para el usuario que la ejecuta. El usuario sólo tiene que proporcionar los parámetros necesarios de entrada y seleccionar un formato de salida. Por tanto, el proceso de ejecución de consultas es exactamente el mismo con independencia de la fuente a la cual se accede.



NAME	DESCRIPTION	ASYNCJOB	USER
Ejecuciones Luca Hoy	Ejecuciones Luca Hoy, Sin variables de entrada	prueba 1-Job	laguero
Proceso Salidas Repetidas Mismo Hilo	Proceso Salidas Repetidas Mismo Hilo	Proceso Salidas Repetidas Mismo Hilo-Job	laguero
Proceso Salidas repetidas	Proceso con nombres de salida repetidas	Proceso Salidas repetidas-Job	laguero
Proyecto de alta en el sistema	Proyectos activos	Proyecto de alta en el sistema-Job	shemera
Prueba IDS	Prueba IDS	dytjn-Job	shemera
Prueba columnas repetidas	Prueba columnas repetidas	Prueba columnas repetidas-Job	laguero
prueba condicion	prueba condicion	prueba condicion-Job	laguero

Figura 1.4: Vista de Ejecución de Consultas

Por ejemplo, en el caso de la consulta *Get Procesos Por Estado* sería necesario proporcionar el estado de los procesos de los que queremos obtener información. Una vez introducido dicho estado, se seleccionaría el botón de ejecución de la consulta (Figura 1.4, etiqueta 2). Finalmente, se muestra el resultado de la consulta, el cuál puede ser visualizado de diferentes formas en función del recurso al que se llama. En nuestro caso, se muestra como una tabla ya que es una consulta a base de datos (Figura 1.4, etiqueta 3).

Lo importante de este proceso es que, una vez definida la consulta, esta se puede ejecutar fácilmente sin conocer los detalles internos de la misma, incluso hasta el tipo de sistema al que se accede.

1.2.3. Limitaciones actuales de LUCA

Actualmente, LUCA proporciona mecanismos para permitir al usuario recuperar de manera uniforme información de diferentes fuentes de datos. No obstante, LUCA por el momento sólo es capaz recuperar información de una única fuente de datos a la vez. Por tanto, cuando es necesario combinar información procedente de distintas fuentes, el propio usuario es el que debe realizar dicho proceso de composición a mano, ejecutando él cada consulta, y utilizando las salidas de cada una de ellas como entradas para las siguientes.

Un ejemplo de dicho proceso de composición sería la necesidad de un dependiente de una tienda de electrodomésticos de obtener la edad de los usuarios que compraron lavadoras durante el mes pasado. Actualmente, la secuencia de consultas que debería de realizar serían las siguientes:

- Primero necesitaría obtener el registro de compras del mes pasado del sistema.
- Después, tras guardar dicho registro, tendría que, uno por uno, seleccionar los que se corresponden con lavadoras.
- Una vez que el usuario tiene las lavadoras compradas el mes pasado, éste tendría que extraer que usuarios han comprado las lavadoras.
- Por último, debería buscar en el sistema cada usuario que ha realizado la compra, a partir del nombre obtenido en el punto anterior, y anotar su edad.

En adelante, estas cadenas de consultas para obtener un resultado concreto las denominaremos *procesos*. El problema actual de LUCA, tal como ilustra el ejemplo anterior, es que no soporta el concepto de *proceso*. Por tanto, para ejecutar un proceso, el usuario tiene que realizar una larga y compleja secuencia de acciones.

1.3. Objetivos del Trabajo de Fin de Grado

El objetivo general de este Trabajo Fin de Grado es integrar en LUCA el concepto de *proceso*. Para ello, hay que dar soporte a dos cuestiones diferentes: (1) la ejecución de los procesos; y (2) la especificación de procesos. Por tanto, el objetivo general de este trabajo se descompone en estos dos subobjetivos principales.

El primer objetivo implica poder tratar procesos en LUCA de la misma forma que se trata las consultas. Es decir, los procesos deberán aparecer como en las consultas bajo una pestaña de gestión y otra de ejecución. Obviamente, la complejidad de ejecutar una proceso es mayor que la de ejecutar una consulta, ya que necesitamos ejecutar varias consultas, guardar

resultados intermedios y utilizar estos resultados como entradas para otras consultas.

El segundo objetivo, que es el que implica una mayor complejidad, consiste en facilitar la especificación de procesos en LUCA. Para que un proceso pueda ser ejecutado, primero debe ser especificado, indicando qué consultas lo componen y cómo se relacionan. De acuerdo con los deseos expresados por los responsables del proyecto LUCA y la empresa CIC, dicho mecanismo de especificación debía ser gráfico, permitiendo así componer consultas de manera visual mediante la interconexión de las salidas de unas con las entradas de otras.

1.4. Alcance del Proyecto

Para refinar estos dos grandes objetivos en una serie de requisitos más concretos, se llevó a cabo en primer lugar una reunión con el Jefe y el Gerente del proyecto. El objetivo de dicha reunión era conocer LUCA en profundidad. A continuación, se nos proporcionaron unos documentos técnicos con los requisitos técnicos tanto para la ejecución de procesos como para el desarrollo del componente gráfico de especificación de procesos. Estos documentos pueden encontrarse en el Anexo adjunto a la memoria. Por tanto, dado que la fase de Ingeniería de Requisitos para este proyecto ya había sido realizada por la propia empresa, no era necesario repetirla dentro de este proyecto.

Por otro lado, el proyecto a desarrollar debía integrarse dentro del proyecto LUCA, el cual ya tenía una arquitectura perfectamente definida. Por tanto, el presente proyecto simplemente debía seguir dichas pautas arquitectónicas, no siendo necesario tampoco realizar una fase de diseño arquitectónico del mismo.

Consecuentemente, el proyecto queda reducido a las fases de diseño microarquitectónico pruebas y despliegue, aunque como es lógico, ésta última fase sería responsabilidad última de la dirección técnica del proyecto LUCA, al ser éstos los encargados de mantener la integridad del producto.

Estas fases de diseño y prueba se realizaron mediante un esquema de desarrollo iterativo e incremental, similar a de las metodologías ágiles, pero sin llegar a adoptar una elementos propios de un proyecto ágil, tales como un *Product Backlog* o la limitación de la duración de las iteraciones a un tiempo concreto.

El desarrollo del producto se dividió en varias iteraciones. En cada iteración, se añadían nuevas funcionalidades al producto, que iba creciendo en complejidad. Al final de cada iteración, se mostraba el producto a los responsables del proyecto LUCA, los cuales daban su aprobación para continuar con el desarrollo y sugerían los cambios y modificaciones que consideraban oportunos.

1.5. Sumario

Este capítulo ha presentado los objetivos del proyecto descrito en este documento. El objetivo general de dicho proyecto era permitir que la herramienta LUCA soportase el concepto de *proceso*, entendiéndose por *proceso* la ejecución de una serie de consultas concatenadas de recuperación de la información donde las salidas de ciertas consultas sirven como entradas para otras.

Antes de entrar a profundizar y refinar dicho objetivo, se ha descrito la herramienta LUCA, detallando sus objetivos generales, su estado actual y sus limitaciones. Seguidamente, se han introducido los objetivos concretos de este proyecto, consistentes en aliviar parte de las limitaciones actuales de LUCA. Finalmente se ha definido el alcance concreto del proyecto, resaltando que las fases de Ingeniería de Requisitos y Diseño Arquitectónico quedan fuera del mismo.

Capítulo 2

Antecedentes

Índice

2.1. GoJS	9
2.2. Patrón <i>Model-View-Presenter (MVP)</i>	16
2.3. Vaadin	16
2.3.1. Desarrollo de Aplicaciones Web con Vaadin	17
2.3.2. Arquitectura Interna de <i>Vaadin</i>	20
2.4. Arquitectura LUCA	22
2.5. Sumario	23

2.1. GoJS

GoJS [7] es una biblioteca de JavaScript para implementar editores gráficos dentro de interfaces web. GoJS facilita la implementación de tareas tales como definición de símbolos gráficos, gestión de paletas de símbolos, arrastrar y soltar (*drag and drop*), copiar y pegar, edición de etiquetas de texto asociadas a símbolos gráficos, menús contextuales, función de deshacer o la gestión de eventos de ratón, entre muchos otros elementos.

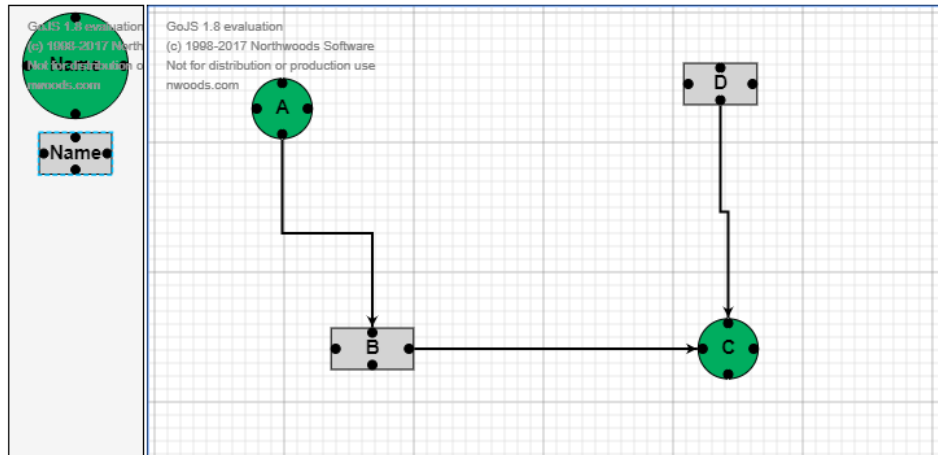


Figura 2.1: Editor gráfico creado en GoJS

Para ilustrar el funcionamiento de GoJS, a continuación se mostrará a modo de ejemplo cómo crear un editor gráfico para diseñar unos pseudo diagramas de flujo compuestos por círculos y rectángulos interconectados por flechas. Dicho editor se muestra en la Figura [2.1](#).

```
1  var $ = go.GraphObject.make;
```

Figura 2.2: Asignación de la variable \$

Para comenzar, cabe destacar que, es necesario reservar dos secciones de la página HTML que lo contiene. Una sección albergará la paleta con los elementos gráficos del editor, que en este caso serán sólo el círculo y el rectángulo, y otra sección para el diagrama o lienzo sobre el que se depositarán los elementos gráficos.

A continuación, se debe realizar una serie de acciones a nivel de Javascript para proporcionar tanto a la paleta de dibujo como al área de dibujo del comportamiento deseado. En primer lugar, crearemos una variable \$ que dé acceso al entorno GoJS. Esto se realiza mediante la llamada a la sentencia `make` de librería *GraphObject* de GoJS (Figura [2.2](#), Línea 1).

```
1 myDiagram =
2 $(go.Diagram, "myDiagramDiv",{
3   grid: $(go.Panel, "Grid",
4     $(go.Shape, "LineH", {
5       stroke: "lightgray",
6       strokeWidth: 0.5
7     } ),
8     $(go.Shape, "LineH", {
9       stroke: "gray",
10      strokeWidth: 0.5,
11      interval: 10
12    } ),
13     $(go.Shape, "LineV", {
14       stroke: "lightgray",
15       strokeWidth: 0.5
16    } ),
17     $(go.Shape, "LineV", {
18       stroke: "gray",
19       strokeWidth: 0.5,
20       interval: 10
21    } )
22  ),
23   allowDrop: true
24 });
```

Figura 2.3: Creación de un diagrama en GoJS

Seguidamente, se personaliza la sección HTML reservada al diagrama, denominada `myDiagramDiv` (Figura 2.3), asignándole un *grid* o cuadrícula de fondo (Figura 2.3, Líneas 3-22), especificando los colores de las líneas de la cuadrícula (mediante la propiedad *stroke*) y dándole la capacidad de poder arrastrar elementos sobre él (Figura 2.3, Línea 23).

```

1 myDiagram.nodeTemplate =
2   $(go.Node, "Spot",{
3     locationSpot: go.Spot.Center
4     selectable: true,
5     selectionAdornmentTemplate: nodeSelectionAdornment
6     resizable: true,
7     resizeObjectName: "PANEL",
8     resizeAdornmentTemplate: nodeResizeAdornment
9     rotatable: true,
10    rotateAdornmentTemplate: nodeRotateAdornment
11  },
12  new go.Binding("location").makeTwoWay(go.Point.stringify),
13  $(go.Panel, "Auto",
14    {
15      name: "PANEL"
16    },
17    $(go.Shape,
18      {
19        portId: "",
20        fromLinkable: true, toLinkable: true, cursor: "pointer",
21      },
22      new go.Binding("figure"),
23      new go.Binding("fill")),
24    $(go.TextBlock,
25      {
26        maxSize: new go.Size(50, 50),
27        editable: true
28      },
29      new go.Binding("text").makeTwoWay()
30    )
31  )
32 );

```

Figura 2.4: Declaración del patrón del Nodo

A continuación deberemos definir lo que en GoJS se conoce como la *plantilla de nodos* o *node template*. Esta plantilla define el comportamiento genérico de todos los nodos que compondrán nuestro diagrama. Estos nodos, en nuestro caso, serán rectángulos y círculos. La Figura 2.4 muestra cómo se crea el patrón que servirán de esqueleto para todos los nodos de nuestro diagrama.

Para crear la definición un nodo lo primero es declarar el nodo con *go.Node* (Figura 2.4, Línea 2). A continuación, especificamos, mediante la definición de ciertas propiedades, cómo se comportará el nodo (Figura 2.4, Líneas 3-10). En nuestro caso, se indica que los nodo, con carácter general, podrán ser seleccionados, redimensionados y rotados (Figura 2.4, Líneas 10-10, respectivamente). Se establece que los nodos aparecerán localizados por defecto en el centro del diagrama en el momento de su creación (Figura 2.4, Línea 43). Además, se indica que la propiedad *location*, que establece la posición del nodo, queda expuesta para su modificación desde el exterior (Figura 2.4, Línea 12).

Siempre que se crea un nodo, se reserva un espacio sobre el que se van a situar los diferentes elementos del nodo. En nuestro ejemplo, este espacio recibe el nombre de *panel* (Figura 2.4, Línea 13). El panel actúa como contenedor de la figura (*shape*) (Figura 2.4, Línea 17), que es la encargada de definir la forma del nodo. En nuestro caso, dicho panel podrá poseer diferentes formas (como serán el círculo y el rectángulo) y colores gracias a los *bindings*, que son los que hacen que estas propiedades sean modificables programáticamente (Figura 2.4, Líneas 22-23). Además, cada nodo poseerá un identificador propio, denominado *portId* (Figura 2.4, Línea 19) y todos los nodos pueden recibir conexiones tanto de entrada como de salida (*toLinkable* y *fromLinkable*) (Figura 2.4, Línea 20). Por último, cada nodo poseerá una etiqueta de texto que permitirá especificar su nombre, el cual también puede ser modificado externamente (Figura 2.4, Líneas 24-31).

Una vez definidas las propiedades básicas de un nodo, el siguiente paso es definir cómo enlazar dichos nodos. Para ello debemos definir dos elementos: *puertos* y *enlaces*. Los enlaces son los elementos que permiten conectar nodos, y los puertos de un nodo son los puntos de dicho nodo desde donde puede partir un enlace o a dónde puede llegar un enlace. En la Figura 2.1, estos puertos aparecen como unos círculos negros en los extremos de las figuras.

```
1  function makePort(name, spot) {  
2      return $(go.Shape, "Circle",  
3          {  
4              desiredSize: new go.Size(7, 7),  
5              alignment: spot,  
6              alignmentFocus: spot,  
7              portId: name,  
8              fromLinkable: true, toLinkable: true,  
9              cursor: "pointer"  
10         });  
11 }
```

Figura 2.5: Función MakePort

```
1 myDiagram.nodeTemplate =  
2 $(go.Node, "Spot",  
3   { locationSpot: go.Spot.Center },  
4   ...  
5   makePort("T", go.Spot.Top),  
6   makePort("L", go.Spot.Left),  
7   makePort("R", go.Spot.Right),  
8   makePort("B", go.Spot.Bottom)  
9 );
```

Figura 2.6: Patrón Nodo con Puertos

Para añadir puertos a los nodos se ha creado una función una función llamada *makePort*, la cual se muestra en la Figura 2.5. Este fragmento de código define un puerto como una forma circular con un identificador (Figura 2.5, Línea 07), una posición (Figura 2.5, Líneas 05-06) y un tamaño (Figura 2.5, Línea 04). Finalmente, usando esta función, añadimos cuatro llamadas a a misma al final de la plantilla de nodos, con el objeto de crear un puerto a cada lado del eje de cada nodo (Figura 2.6, Línea 04).

```
1 myDiagram.linkTemplate =  
2 $(go.Link,  
3   $(go.Shape,  
4     { isPanelMain: true, strokeWidth: 2 } ),  
5   $(go.Shape,  
6     { toArrow: "Standard", stroke: null }  
7   )  
8 );
```

Figura 2.7: Declaración del patrón del Link

Una vez definidos los puertos, especificamos cómo se comportarán los enlaces entre nodos (Figura 2.7). En nuestro caso, se declara que los enlaces irán de un nodo principal *isPanelMain* a uno secundario, se define el grosor del enlace y se establece que el enlace poseerá una flecha al final del mismo (*toArrow: "Standard"*).

```
1 myPalette =
2   $(go.Palette, "myPaletteDiv",
3   {
4     nodeTemplateMap: myDiagram.nodeTemplateMap,
5     model: new go.GraphLinksModel([
6       {
7         text: "Name",
8         figure: "Circle",
9         fill: "#00AD5F"
10      },
11      {
12        text: "Name",
13        figure: "Rectangle",
14        fill: "lightgray"
15      }
16    ])
17   })
18 );
```

Figura 2.8: Creación de la Paleta de Nodos

Por último, se procede a crear la paleta que contendrá los círculos y los rectángulos, tal como se muestra en la Figura 2.8. Como se puede observar, la paleta se sitúa sobre una sección HTML denominada *myPalletteDiv* (Figura 2.8, Línea 2), la cual fue reservada con anterioridad. Para la definición de los nodos que estarán dentro de la paleta se utiliza el mismo patrón e nodos creado con anterioridad (Figura 2.8). Finalmente, por último, se declara el *modelo*, que es el conjunto de figuras concretas que poseerá dicha paleta. Para crear los elementos del modelo, se instancia la plantilla de nodos y se cambian sus propiedades con objeto de crear objetos diferentes. En nuestro caso, creamos círculos de color verde (Figura 2.8, Línea 06-10) y rectángulos de color gris (Figura 2.8, Línea 11-15), ambos con *Name* como nombre por defecto, el cual podrá ser luego editado.

Una vez definidos estos elementos, nuestro editor gráfico queda implementado, y GoJS se encargará de dibujar la paleta y el diagrama, así como, de pintar los elementos sobre el área de dibujo cuando éstos sean seleccionados en la paleta, permitiendo arrastrarlos, redimensionarlos o renombrarlos, entre otras funciones.

2.2. Patrón *Model-View-Presenter* (MVP)

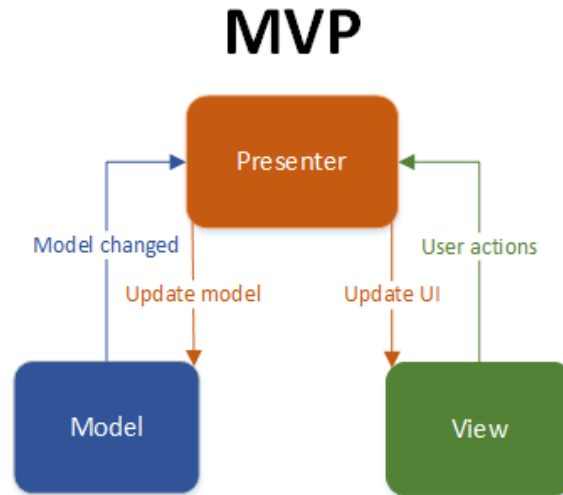


Figura 2.9: Patrón Modelo-Vista-Presentador

Este patrón está orientado a crear interfaces de usuario y su objetivo es separar la lógica de la aplicación de los detalles estéticos de la interfaz. Se compone de tres módulos independientes: *modelo*, *vista* y *presentador* (ver Figura 2.9).

El *modelo* es el encargado de gestionar los datos de la aplicación, y que de alguna forma serán mostrados al usuario. La *vista* es una interfaz de usuario, normalmente gráfica, que se encarga de mostrar datos al usuario de la manera más amigable posible. Por último el presentador se sitúa entre el *Modelo* y la *Vista* y se encarga de conectar ambos elementos. Los eventos realizados sobre la interfaz de usuario se delegan en el presentador, que es el que decide qué cambios se deberán realizar sobre la interfaz gráfica. Para ello puede acceder a datos al modelo, o solicitar la modificación de los mismos.

De esta forma se separa lo que es la lógica de navegación y procesamiento de eventos de los detalles de cómo se organiza exactamente una interfaz de usuario. De este modo, se aísla dicha lógica de cambios estéticos que pudiesen producirse en dicha interfaz.

La siguiente sección mostrará como *Vaadin*, un framework para el desarrollo de aplicaciones web, implementa de una manera concreta este patrón.

2.3. Vaadin

Vaadin [3] es un *framework* para el desarrollo de aplicaciones *web* avanzadas, también conocidas como *Rich-Internet Applications* (RIA) [4]. El

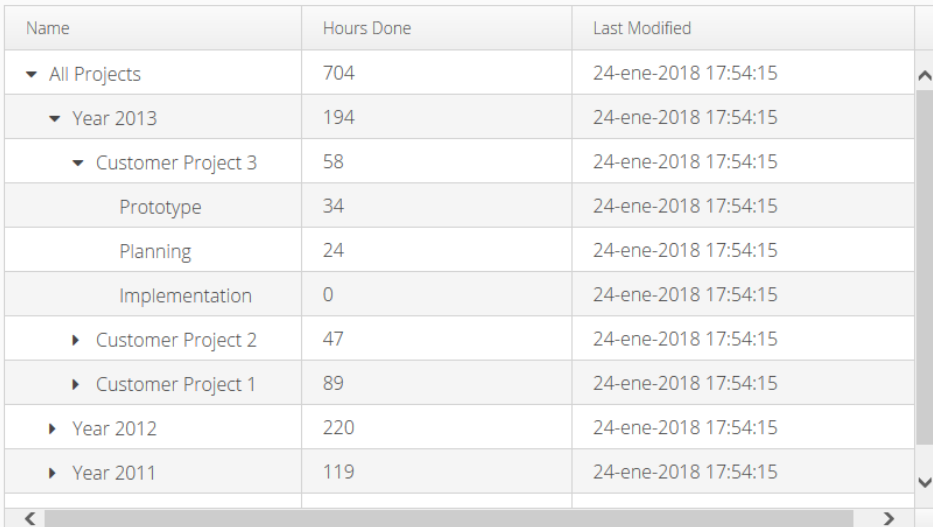
objetivo del paradigma *RIA* es desarrollar aplicaciones *web* con interfaces avanzadas que les haga asemejarse a las aplicaciones de escritorio. La principal ventaja que aporta *Vaadin* es que permite escribir aplicaciones en código Java, como si fuesen de escritorio, y luego este código es transformado para que funcione en entornos web.

Una de las características diferenciadores de *Vaadin* es que, al contrario de las librerías de JavaScript tradicionales, *Vaadin* también contempla la parte del servidor, por lo se generan tanto las llamadas al servidor desde la interfaz gráfica (*front-end*) como la recepción y tratamiento de esas llamadas en la parte del servidor (*back-end*).

Para abstraer al usuario de elementos relacionados con HTML o Javascript, Vaadin utiliza los llamados *componentes*. Un componente representa un elemento gráfico o *widget*. Para el desarrollo de los componentes, Vaadin proporciona una serie de clases reutilizables que contienen los infraestructura necesaria para facilitar su traducción a código HTML y Javascript. Para crear *componentes*, los desarrolladores de Vaadin deben simplemente extender estas clases.

A continuación, para introducir al lector en el funcionamiento de *Vaadin*, se mostrará primero cómo construir una pequeña aplicación web utilizando *Vaadin*, y a continuación se detallará el funcionamiento de su arquitectura utilizando como ejemplo dicha aplicación web.

2.3.1. Desarrollo de Aplicaciones Web con Vaadin



Name	Hours Done	Last Modified
▼ All Projects	704	24-ene-2018 17:54:15
▼ Year 2013	194	24-ene-2018 17:54:15
▼ Customer Project 3	58	24-ene-2018 17:54:15
Prototype	34	24-ene-2018 17:54:15
Planning	24	24-ene-2018 17:54:15
Implementation	0	24-ene-2018 17:54:15
▶ Customer Project 2	47	24-ene-2018 17:54:15
▶ Customer Project 1	89	24-ene-2018 17:54:15
▶ Year 2012	220	24-ene-2018 17:54:15
▶ Year 2011	119	24-ene-2018 17:54:15

Figura 2.10: Árbol de Proyectos

```

1  @Override
2  protected void init(VaadinRequest request) {
3
4      final VerticalLayout layout = new VerticalLayout();
5      layout.setSpacing(true);
6      layout.setMargin(true);
7
8      final TreeGrid grid = new TreeGrid();
9      grid.setWidth(800, Unit.PIXELS);
10     grid.setHeight(450, Unit.PIXELS);
11
12     JobContainer container = new JobContainer();
13     grid.setContainerDataSource(container);
14
15     layout.addComponent(grid);
16     setContent(layout);
17 }

```

Figura 2.11: Interfaz de Usuario Vaadin

```

1  public class JobContainer extends HierarchicalContainer
2  implements Collapsible, Measurable {
3
4      static final String PROPERTY_NAME = "Name";
5      static final String PROPERTY_HOURS = "Hours done";
6      static final String PROPERTY_MODIFIED = "Last modified";
7
8      public JobContainer() {
9          addContainerProperty(PROPERTY_NAME, String.class, "");
10         addContainerProperty(PROPERTY_HOURS, Integer.class, 0);
11         addContainerProperty(PROPERTY_MODIFIED, Date.class, new Date());
12
13         ...
14     }
15
16     private Object addItem(Object[] values) {...}
17     private Object addChild(Object[] values, Object parentId) {...}
18     private void setProperties(Item item, Object[] values) {...}
19     private void addChildren(Object itemId) {...}
20     private boolean removeChildrenRecursively(Object itemId) {...}
21
22     @Override
23     public boolean hasChildren(Object itemId) {...}
24 }
25

```

Figura 2.12: Contenedor TreeGrid

La Figura 2.10 muestra la interfaz de una pequeña aplicación web consistente en un árbol de tareas. Para construir este ejemplo en *Vaadin*, en primer lugar definimos su interfaz gráfica. Para crear dicha interfaz gráfica, nos basamos en un componente gráfico, o *widget*, denominado *TreeGrid* (Figura 2.11, Línea 8). Como puede observarse, este componente gráfico se usa

directamente desde código Java, tal como se crearía una interfaz Java de escritorio, utilizando elementos propios de Java como los *layouts* (Figura 2.11, Líneas 4-6) y no siendo necesario escribir nada en Javascript o HTML. Este componente mostrará los datos proporcionados por el contenedor de datos *JobContainer* (Figura 2.11, Líneas 12-13).

```
1 public class JobContainer
2     ...
3     private Map<Object, Boolean> expandedNodes = new HashMap<>();
4
5     @Override
6     public void setCollapsed(Object itemId, boolean collapsed) {
7         expandedNodes.put(itemId, !collapsed);
8         if (collapsed) {
9             removeChildrenRecursively(itemId);
10        } else {
11            addChildren(itemId);
12        }
13    }
14
15    @Override
16    public boolean isCollapsed(Object itemId) {
17        return !Boolean.TRUE.equals(expandedNodes.get(itemId));
18    }
19 }
```

Figura 2.13: Contenedor TreeGrid Collapsible

```
1 public class JobContainer
2     ...
3     @Override
4     public int getDepth(Object itemId) {
5         int depth = 0;
6         while (!isRoot(itemId)) {
7             depth++;
8             itemId = getParent(itemId);
9         }
10        return depth;
11    }
12 }
```

Figura 2.14: Contenedor TreeGrid Measurable

La clase *JobContainer* (Figura 2.12) es la que proporcionará los datos que se muestran en el *grid*. Esta clase extiende de una clase de Vaadin llamada *HierarchicalContainer* y se encarga de implementar toda la lógica para almacenar de forma jerárquica los nodos. Además, implementa las interfaces de Vaadin *Collapsible* (Figura 2.13) y *Measurable* (Figura 2.14), encargadas

de contraer el árbol de elementos y de calcular la profundidad del elemento en la jerarquía, respectivamente.

Tal como se puede apreciar, no es necesario ningún conocimiento de las tecnologías web tales como *HTML* [2], *CSS* [2], *Javascript* [2], *HTTP* [6] o *AJAX* [5] para construir el ejemplo de la Figura 2.10 utilizando *Vaadin*.

2.3.2. Arquitectura Interna de *Vaadin*

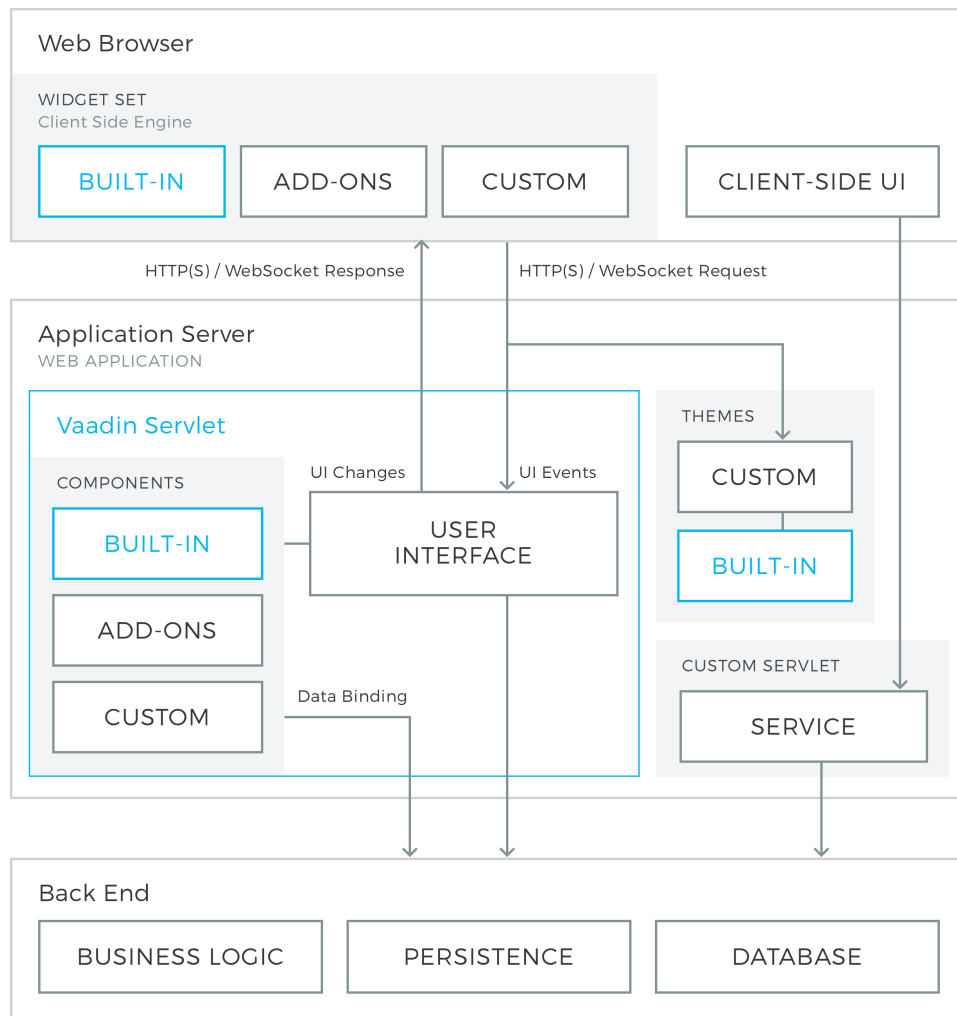


Figura 2.15: Arquitectura Interna Vaadin

Las clases descritas en el apartado anterior se transforman, utilizando *Vaadin*, en código *Javascript* que pueda ser procesado en un navegador web, más una serie de llamadas *AJAX* a una serie de funciones desplegadas en el

servidor que se encargan de gestionar la interacción con el servidor, de acuerdo al patrón *Model-View-Presenter (MVP)* (ver Sección 2.2). La Figura 2.15 muestra el funcionamiento general de este esquema.

Tal como se puede observar en la Figura 2.15, el código generado por *Vaadin* se estructura en tres módulos: (1) *WebBrowser*; (2) *Back End*; y, (2) *Application Server*.

El primer módulo, *Web Browser*, es el que correría en el lado del cliente y contiene simplemente los elementos necesarios para mostrar los controles gráficos avanzados, recoger los eventos de usuario y comunicarse con el servidor de la aplicación. Este módulo ejerce de *Vista*.

El segundo módulo, *Back End* es el que contiene la lógica de negocio de la aplicación, incluyendo objetos de negocio y la infraestructura necesaria para persistirlos en bases de datos. Este módulo ejerce de *Modelo*.

El tercer módulo, *Application Server*, es el que hace de puente entre la vista y el modelo, recibiendo los eventos que se realizan sobre la interfaz de usuario y respondiendo de manera adecuada. Este módulo ejerce de *Presentador*.

Cada evento (*UI Events* en Figura 2.15) realizado sobre la interfaz de usuario (*Web Browser*) se delega mediante una petición *AJAX* sobre *HTTP* al módulo *Application Server*. Este módulo es el que decide cómo la aplicación debe reaccionar a dicho evento. Esta reacción podría implicar desde simple cambio en la interfaz gráfica hasta el procesamiento de una larga y compleja regla de negocio.

Por ejemplo, si se decide desplegar un año dentro del árbol de tareas de la Figura 2.10, este evento se delega en la capa de aplicación (*Application Server*), que devuelve los datos necesarios, para actualizar la interfaz gráfica. Si, por otro lado, quisiéramos añadir una nueva tarea a nuestro árbol de tareas, este evento se delegaría de nuevo en la capa de aplicación, la cual llamaría a la parte de negocio (*Back End*) para iniciar una transacción que, tras verificar que se puede añadir dicha tarea de acuerdo con las reglas definidas para el negocio, actualice el estado de los objetos de negocio de manera adecuada, y los refleje en la base de datos correspondiente.

2.4. Arquitectura LUCA

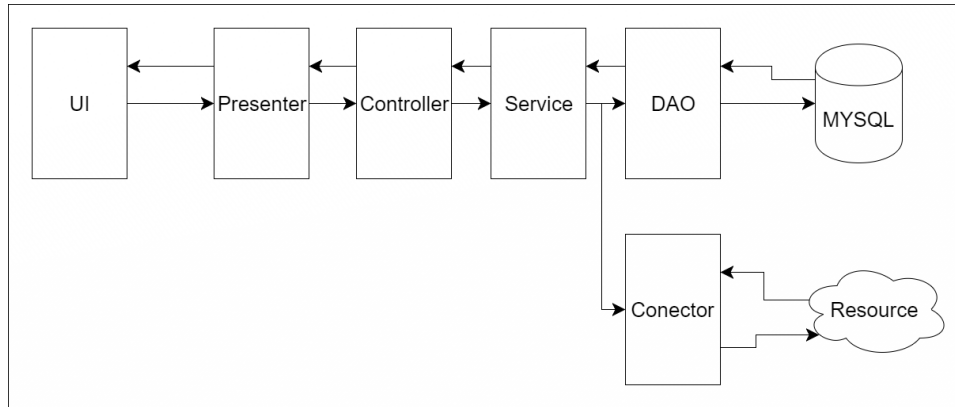


Figura 2.16: Arquitectura de LUCA

La Figura 2.16 muestra la arquitectura a alto nivel de *LUCA*. *LUCA* utiliza una arquitectura en capas, basada en la arquitectura inicial de *Vaadin*. Se distinguen tres capas principales, propias de los sistemas empresariales: *presentación*, *negocio* y *persistencia*.

La capa de *presentación* se organiza de acuerdo con el patrón MVP y se genera automáticamente desde ficheros Java que describen la interfaz de usuario gracias al uso del framework *Vaadin*.

La capa de *negocio* se subdivide en dos subcapas: *Access Control Layer* y *Business Layer*. La primera subcapa se encarga de comprobar que quién realiza peticiones sobre la capa de negocio tiene los permisos y privilegios necesarios para ejecutar dichas acciones. Una vez otorgado dicho acceso, la petición se pasa a la capa de negocio propiamente dicha (*Business Layer*), que es la que se encarga de actualizar los objetos del dominio de acuerdo con las reglas de negocio definidas. Para actualizar dichos objetos de dominio puede ser necesario recuperar ciertos elementos desde los sistemas donde se hallen almacenados, así como salvarlos en los mismos. Para realizar estas acciones, la capa de negocio se comunica con la capa de persistencia.

La capa de *persistencia* trata de ocultar los detalles de los diferentes sistemas donde se almacenan los datos a la capa de negocio, proporcionando una interfaz de alto nivel a través de la cual acceder a dichos datos. *LUCA* utiliza dos clases de sistema de almacenamiento claramente diferenciados.

Los datos propios de *LUCA*, tales como consultas existentes o usuarios creados, se almacenan en una base de datos relacional alojada en un gestor de bases de dato MySQL, al que se accede a través de una serie de clases DAO (*Data Access Objects*). Además, *LUCA* necesita acceder a múltiples sistemas, de naturaleza diversa, para recuperar los datos de las consultas

que ejecuta. Para facilitar el acceso a dichos sistemas, LUCA hace uso del concepto de *conector*.

Un conector LUCA es un componente que se encarga de manejar la interacción con una datos de una fuente de datos externa. LUCA posee componentes genéricos por cada tipo de fuente a la que puede conectarse. De este modo, LUCA proporciona conectores para comunicarse con bases de datos relacionales alojadas en *Oracle* a través de *JDBC (Java DataBase Connectivity)*, servicios REST o servicios SOAP, entre otros. De este modo, cuando queremos incorporar una nueva fuente de datos a LUCA, simplemente deberemos instanciar un conector del tipo adecuado y configurarlo de manera adecuada para acceder a dicha fuente.

2.5. Sumario

El presente capítulo ha presentado todos los conceptos y tecnologías necesarias para poder comprender el trabajo presentado en esta memoria. En primer lugar se ha introducido el framework GoJS, que es el que se ha utilizado para implementar el editor gráfico para la especificación de consultas concatenadas o procesos dentro de LUCA. Dado que este proyecto se integra dentro de la herramienta LUCA, se ha descrito la arquitectura de la misma, para lo que ha sido necesario introducir el patrón *Modelo-Vista-Presentador* y *Vaadin*, por ser esta la tecnología sobre la que se sustenta la herramienta LUCA.

Capítulo 3

Desarrollo del Proyecto

Índice

3.1. Introducción	24
3.2. Desarrollo del Editor Gráfico	26
3.3. Integración del editor con <i>Vaadin</i>	28
3.4. Integración del editor con LUCA	30
3.5. Gestión, Ejecución y Persistencia de los Procesos	32
3.5.1. Ejecución de Procesos	36
3.6. Sumario	37

3.1. Introducción

El presente capítulo detalla el proceso de desarrollo del proyecto realizado dentro de este Trabajo Fin de Grado. Tal como se ha comentado, el objetivo del proyecto era proporcionar al producto LUCA una herramienta gráfica para poder crear *procesos*, es decir, poder encadenar consultas entre sí en base a sus valores de entrada y de salida.

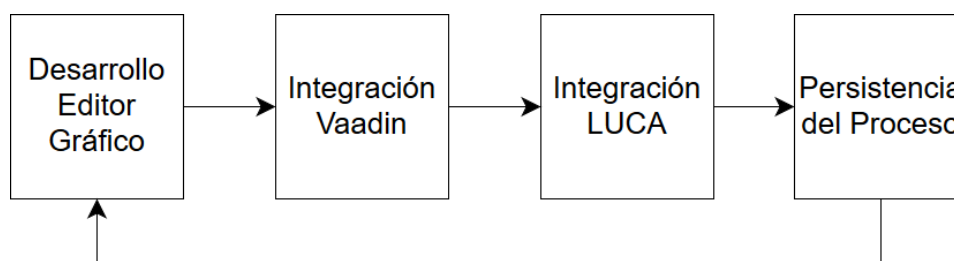


Figura 3.1: Flujo de Desarrollo

Para alcanzar este objetivo, se siguió el esquema de trabajo que se muestra en la Figura 3.1.

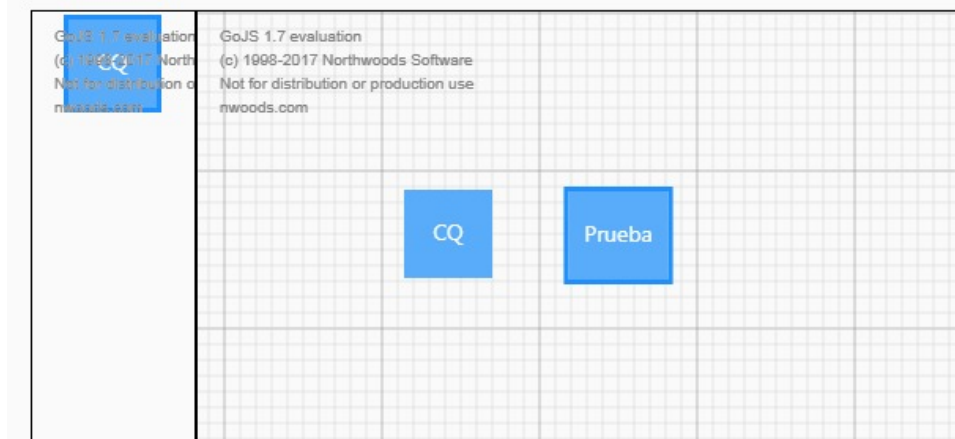


Figura 3.2: Primer Contacto Editor

La primera etapa consistió en realizar un primer contacto con la herramienta *GoJS*, para ello, se realizó una prueba de concepto donde se creaba un diagrama sencillo con cajas o nodos que se conectan entre sí (Figura 3.2).

Con la prueba de concepto funcionando, la segunda etapa consistió en integrar el diagrama en *GoJS* con el framework de *Vaadin*. De esta forma se obtenía un editor gráfico de procesos implementado en *Java*, que podía ser utilizado en cualquier proyecto *Vaadin*.

La tercera etapa se centró en integrar dicho proyecto genérico *Vaadin* en el producto LUCA. El objetivo era ser capaz de mostrar el editor y poder interactuar con él y recibir los eventos ocurridos, de acuerdo con el patrón *Modelo-Vista-Presentador* y la arquitectura de *Vaadin*.

Por último, la cuarta etapa consistió en añadir la capa de negocio y de persistencia al modelo de datos de LUCA, creado específicamente para persistir *procesos* y toda su jerarquía de clases.

Una vez acabadas las cuatro etapas de desarrollo, comenzó un curso de iteraciones para refinar el proyecto e ir dando la complejidad final al mismo, de forma que se fue completando toda la lógica de negocio, así como, el aspecto gráfico final.

Paralelamente a estas etapas, y en cuanto fue posible, se trabajó en la parte de ejecución de los procesos especificados desde el editor gráfico y almacenados en LUCA. Las siguientes secciones proporcionan detalles sobre la ejecución de cada etapa.

3.2. Desarrollo del Editor Gráfico

El editor gráfico utiliza *GoJS* para crear toda la interfaz visual. Este framework (como ya se explicó en el apartado de documentación) utiliza *templates* para definir el aspecto estético de los diferentes componentes. Por lo tanto, el primer paso, tras tener claro cómo debía de ser un primer boceto de interfaz, fue crear todos los *templates* de los que se iba a componer.

El segundo paso, una vez conseguida la estética y estructura de elementos deseada, fue comprobar la funcionalidad del mismo, desde asegurar que al interactuar con el editor se creaban los eventos oportunos, hasta comprobar que los elementos se mostraban en pantalla tal y como se quería.

```

1  myDiagram.nodeTemplate = $(go.Node, "Table", {
2    locationObjectName : "BODY",
3    locationSpot : go.Spot.Center,
4    selectionObjectName : "BODY"
5  }, new go.Binding("location", "loc", go.Point.parse)
6    .makeTwoWay(go.Point.stringify),
7
8    $(go.Panel, "Auto", {
9      row : 1,
10     column : 1,
11     name : "BODY",
12     stretch : go.GraphObject.Fill
13   },
14     $(go.Shape, "Rectangle", {
15       fill : "#58ACFA",
16       stroke : null,
17       strokeWidth : 0,
18       name : "rectangle",
19       minSize : new go.Size(56, 56)
20     }),
21     $(go.TextBlock, "CQ Default", {
22       margin : 10,
23       textAlign : "center",
24       font : "14px Segoe UI,sans-serif",
25       stroke : "white",
26       editable : true
27     }, new go.Binding("text", "name").makeTwoWay()
28   )
29   );
30

```

Figura 3.3: *Template* Prueba Editor Inicial

La Figura 3.2 y la Figura 3.3 muestran el primer boceto de editor. La primera se corresponde con una captura de la interfaz y la segunda con el *template* utilizado para poder crearla. Como se puede comprobar, para el primer contacto se creó un primer área desde la que arrastrar los diferentes elementos (en nuestro caso nodos), y el segundo se corresponde con el contenedor de dichos nodos. El template muestra como se crea un nodo

(Figura 3.3 Línea 1) y se le asigna un único panel (Figura 3.3 Línea 8) que contiene una rectángulo (Figura 3.3 Líneas 14-28) y una caja de texto dentro de dicho rectángulo (Figura 3.3 Líneas 21-27).

Con la primera prueba de concepto terminada, las siguientes iteraciones se centraron en otorgar nuevas funcionalidades al diseño:

- Permitir cambiar colores y tamaños de los elementos desde el exterior, es decir, desde una aplicación que utilice este componente gráfico. Para ello fue necesario declarar una serie de *bindings* y comprobar su correcto funcionamiento.
- Permitir crear elementos con diferentes formas a partir de formas pre-diseñadas en *GoJS* o a partir de imágenes *SVG*.

Durante el proceso gráfico de enlazado de elementos, fue necesaria la modificación del funcionamiento por defecto de GoJS. GoJS, cuando se crea un enlace entre dos elementos, crea un evento de enlazado y además añade al modelo de datos de *GoJS* un *link* entre los dos elementos. Este flujo contradice los principios del patrón *Model-View-Presenter*. De acuerdo con este patrón, en el flujo de creación de un enlace, *GoJS* sólo se tiene de encargar, en una primera instancia, de enviar el evento de enlazado. Sólo después, cuando se le confirme desde el *presentador* que todo es correcto, debería crear gráficamente dicho enlace. Sin embargo, como ya se ha comentado, GoJS por defecto crea el enlace y manda luego un evento que comunica su creación. Para conseguir el comportamiento adecuado, hubo que modificar la herramienta de enlazado (*LinkTool*, es la encargada de producir eventos y modificar el modelo durante el proceso de enlazado para que enviase el evento exclusivamente).

```
1  function NotPersistLinkingTool() {
2      go.LinkTool.call(this);
3      this.name = "NotPersistLinkingTool";
4      this.portGravity = 0;
5  }
6
7  go.Diagram.inherit(NotPersistLinkingTool, go.LinkTool);
8
9  NotPersistLinkingTool.prototype.insertLink =
10 function(fromnode, fromport, tonode, toport) {
11     this.archetypeLinkData = {
12         from: fromnode.data.key,
13         fromPort: fromport.data.key,
14         to: tonode.data.key,
15         toPort: toport.data.key
16     }
17     myDiagram.raiseDiagramEvent('LinkDrawn', this.archetypeLinkData);
18 };
```

Figura 3.4: Link Tool

En la figura 3.4 se explica como se realizó dicha modificación. Las líneas 1-7 describen la creación de un enlace (constructor) y cómo se sobrescribe la actual clase que permite componer enlaces. Por último las líneas 9-18 describen dos aspectos: por un lado, cómo se establecen los orígenes tanto de los nodos (líneas 12 y 14) como de las variables (líneas 13 y 15); y, por otro lado se lanza un evento para comunicar que se quiere crear un enlace (línea 17).

Debido a que esta herramienta se focaliza en representar información sobre una interfaz, las pruebas realizadas se definieron y ejecutaron de manera sistemática, pero no se automatizaron utilizando ningún *framework* tipo *Selenium* debido a su alta complejidad, inversión de tiempo e inestabilidad frente a los cambios.

3.3. Integración del editor con *Vaadin*

```
1  @JavaScript({
2      "vaadin://procesos/js/componentsLibrary.js",
3      "vaadin://procesos/js/connectorSample.js",
4      "vaadin://procesos/js/go.js"
5  })
6  public class DiagramComponent extends AbstractJavaScriptComponent {
7      public DiagramComponent() {
8          addFunction("ExternalObjectsDropped", new JavaScriptFunction() {
9              @Override
10                 public void call(JsonArray arguments) {}
11             ...
12         }
13     public void createNode(Node node)
14     {
15         List<Node> nodes = getState().getNodes();
16         nodes.add(node);
17         getState().setNodes(nodes);
18     }
19 }
```

Figura 3.5: Componente Vaadin

El objetivo principal del editor gráfico de procesos es proporcionar un editor gráficos que cualquier proyecto *Vaadin* pueda utilizar para modelar procesos. Para ello es necesario hacer que el editor gráfico creado en la sección anterior se integre con *Vaadin*, enviando los eventos que genere al *presentador* de *Vaadin* y reaccionando adecuadamente a las respuestas de éste.

En la Figura 3.5 se muestra el ejemplo creado para la primera integración. En ella se puede observar cómo se importan los diferentes ficheros *Javascript* necesarios para utilizar el editor creado previamente (Figura 3.5 Líneas 2-4).

A continuación se indica cómo se han de tratar los eventos que se espera que lleguen desde el editor gráfico (Figura 3.5 Líneas 9-12. Los eventos implementados se encuentran en el anexo apéndice A).

Por cada evento, se recibe un *array* con los objetos *JSON* que contienen los datos de cada evento. A continuación, se procesa dicho evento. Después se trata dicho evento en función de su objetivo. Por último se muestra como se define un método de creación de nodos. Este método interacciona con el estado del componente para llevar a cabo la creación un nodo (Figura 3.5 Líneas 18-23).

Para que lo anterior funcione, es necesario que exista una comunicación entre la herramienta *GoJS* y *Vaadin* a través del envío de eventos. Estos eventos son producidos por *GoJS* y trasladados al presentador de *Vaadin*.

```

1  var diagramComponent = new mylibrary.DiagramComponent(element);
2
3  this.onStateChange = function() {
4      diagramComponent.setNodes(this.getState().nodes);
5  };
6
7  var self = this;
8
9  myDiagram.addDiagramListener('ExternalObjectsDropped', function(
10     properties) {
11     self.ExternalObjectsDropped(properties.subject.Ca.key.Vd);
12 });
13 ...

```

Figura 3.6: Conector

Para que esta comunicación entre *GoJS* y *Vaadin* sea posible, es necesario un fichero que realice las funciones de *conector*. El objetivo de este conector será recoger los eventos producidos en el editor gráfico, redirigirlos a *Vaadin*. A continuación, siguiendo las órdenes de *Vaadin*, el conector realizará los cambios pertinentes sobre la interfaz del editor gráfico. La Figura 3.6 muestra el *conector* creado para la prueba de concepto. En él se puede ver cómo se crea el diagrama a partir de la librería importada de *GoJS* (Figura 3.6 Línea 1). A continuación se declara la actuación ante un cambio en el estado (Figura 3.6, Líneas 3-5). Por último, se muestra un ejemplo de cómo se manda un evento al componente de *Vaadin* (Figura 3.6 Líneas 9-11), referenciado por la variable *self*. De esta forma se produce un diálogo entre ambos conceptos en base a eventos y cambios en la interfaz gráfica.

En cada iteración se fueron incorporando tanto al presentador de *Vaadin* como al conector la gestión de los nuevos eventos que se fueron añadiendo al editor gráfico.

Debido a la complejidad que existe para comprobar que las acciones que

se realizan desde el proyecto del editor a nivel *Java* se muestran gráficamente en el editor, no se han automatizado las pruebas de este componente, aunque si se han definido y ejecutado manualmente de una manera sistemática.

3.4. Integración del editor con LUCA

El objetivo de hacer el editor gráfico como un componente propio de *Vaadin*, era facilitar su integración con el proyecto LUCA. Gracias al esfuerzo realizado en la sección anterior, esta integración consiste ahora en simplemente incluirlo en una vista, conforme al esquema del patrón *Mode-View-Presenter*. Una vez que la vista alberga dicho componente, al que denominaremos a partir de ahora *DiagramComponent*, es necesario registrarlo con el presentador. De esta forma el componente ya está preparado para ser utilizado y mostrar el diagrama. Por último sería necesario mandar alimentar el editor con los datos necesarios para su alimentación, como las consultas que se pueden utilizar como bloques para la definición de procesos.

```
1  public class ProcessConfigurationView {
2
3      private VerticalLayout content;
4      private DiagramComponent diagramComponent;
5
6      public VerticalLayout getContent()
7      {
8          if(content == null)
9          {
10             content = new VerticalLayout();
11             content.setSizeFull();
12             content.setSpacing(true);
13             content.addComponent(getSplitPanel());
14          }
15          return content;
16      }
17
18      public DiagramComponent getDiagramComponent()
19      {
20          if(diagramComponent == null)
21          {
22             diagramComponent = new DiagramComponent();
23             diagramComponent.setSizeFull();
24          }
25          return diagramComponent;
26      }
27  }
```

Figura 3.7: Ejemplo de vista

La figura 3.7 muestra una de las vistas utilizadas en el proyecto. En esta vista se declara el componente creado (editor gráfico, Figura 3.7 Líneas 4 y 18-26) y es insertado en el contexto de la vista (Figura 3.7 Línea 13).

```

1 public class ProcessConfigurationPresenter extends
2 AbstractPresenter<ProcessConfigurationView>{
3
4     private static final Logger LOGGER =
5     LoggerFactory.getLogger(ProcessConfigurationPresenter.class);
6
7     @Override
8     protected void init()
9     {
10         super.init();
11         getView().getDiagramComponent().getDiagramListener().
12         addLinkDrawnListener(new LinkDrawnListenerImpl());
13     }
14
15     @Override
16     public void queryAdded(CustomQuery customQuery)
17     {
18
19         LucaSubProcessBox subProcessBox =
20         new LucaSubProcessBox(customQuery);
21         subProcessBox.setId(customQuery.getId().toString());
22         subProcessBox.setName(customQuery.getName());
23         subProcessBox.setLocation(new Location(100, 100));
24
25         try
26         {
27             String typeResource = getDatasourceType(customQuery);
28             getView().getDiagramComponent().
29             addSubProcessBox(subProcessBox, typeResource);
30             LOGGER.debug("Added CustomQuery [{}]", customQuery.getName());
31         }
32         catch (BadSubProcessBoxException | SubProcessBoxExists e)
33         {
34             LOGGER.error("Invalid subprocess box: " + e.getMessage());
35         }
36     }
37 }

```

Figura 3.8: Ejemplo de presentador

El presentador de la figura 3.8 utiliza la vista descrita anteriormente y realiza una acción de inserción de un subprocesso utilizando el editor gráfico. En su método constructor define los eventos que desea escuchar del editor gráfico (Figura 3.8 Líneas 7-13). El método de inserción de subprocessos (Figura 3.8 Líneas 15-35) recibe la consulta que albergará dicho subprocesso y se divide en dos partes, en la primera se declara el subprocesso que será enviado al editor gráfico para mostrar el elemento (Figura 3.8 Líneas 19-23), por último, en la segunda parte se realiza la llamada al editor gráfico para conseguir lo descrito anteriormente (Figura 3.8 Líneas 25-35).

En las siguientes iteraciones en esta etapa, se creó la vista de gestión para la creación, edición y eliminación de procesos. Una vez que LUCA era capaz de soportar el concepto de proceso, éstos debían gestionarse del mismo modo que las consultas. Además se enriqueció la vista de gestión

El objetivo de este paso era que LUCA pudiese almacenar los procesos definidos en una base de datos, de manera que éstos fuesen perdurables. Para ello lo primero era definir un modelo conceptual de datos para los procesos. Dicho modelo conceptual se muestra en la Figura 3.9. En esta Figura, las clases sombreadas son clases propias del producto LUCA, a las que referencian las clases creadas en este proyecto.

El modelo consta de una clase principal llamada *Process*, esta es la encargada de albergar todos los datos necesarios para poder crear un proceso, tiene establecida un estado (para determinar si es un proceso preparado para ser ejecutado o no, o si ha sido eliminado), se compone de una lista de las variables de entrada y de salida, se describen a raíz del subproceso al que pertenecen y la variable de entrada o de salida que conecta. El proceso contiene a su vez una lista de subprocesos. Un *SubProceso* tiene asociada una consulta, que será la que se ejecute una vez definido el *process*. Cada enlace entre subprocesos y entre el subproceso y el proceso, se persiste gracias a un *SubProcessRelation*, este simula un enlace, y por lo tanto contiene la información de las variables de los procesos que conecta, además, existe otra variante de él, el *RelationCondition*, el cuál informa de un enlace hacia un nodo de condición, por lo tanto también alberga la variable del nuevo subproceso al que conecta.

Este modelo conceptual de datos fue implementado en *Java*, utilizando POJOs (*Plain Old Java Objects*) y posteriormente anotado usando *JPA* (*Java Persistence API*) [1] para indicar cómo debían crearse las tablas de la base de datos relacional que daría soporte a dicho modelo. De esta forma, es *JPA* quién automáticamente creará el esquema de base de datos, así como las interfaces de los repositorios pertinentes para manipular los objetos de negocio definidos por nuestro modelo conceptual de datos. Además, esta herramienta asegura el correcto funcionamiento de la capa de persistencia, por lo tanto no siendo necesario crear pruebas a este nivel.

Una vez creado este modelo conceptual de datos, la capa de negocio se subdividió en dos subcapas. La primera era la capa de *control de acceso* y se encargaría de verificar que quién desea utilizar la capa de negocio posea los privilegios y permisos necesarios para hacerlo. La segunda subcapa era la propia capa de negocio, que gestiona los procesos y los diferentes elementos que posee el modelo. Para ello esta capa se comunica con una capa de persistencia que utiliza por debajo *JPA* para la comunicación con la base de datos relacional.

<pre>1 @PreAuthorize(ProcessPermission.AUTH_PROCESS_CREATE) 2 Process saveProcess(Process process);</pre>

Figura 3.10: Interfaz controladora

```
1  @Autowired
2  private ProcessService processService;
3
4  @Override
5  public Process saveProcess(Process process){
6      return processService.saveProcess(process);
7  }
```

Figura 3.11: Implementación interfaz controladora

La Figura 3.10 muestra cómo se verifican los permisos y privilegios del usuario que utiliza el servicio. Para llevarlo a cabo, se comprueba previo a la ejecución del método, que el usuario tiene en este caso los permisos para crear un proceso. Una vez verificados esos permisos, se ejecuta el método. Internamente, se llama al servicio propio de la capa de negocio para realizar la operación de persistencia (Figura 3.11).

En las consecutivas iteraciones se fueron añadiendo las funcionalidades necesarias para poder persistir el modelo de datos conforme este se iba enriqueciendo en las sucesivas iteraciones.

```
1  INSERT INTO
2  LUCA_PROCESSES(id ,DESCRIPTION,NAME...)
3  VALUES(2 , 'process_test_desc' , 'process_test_name' ,...
```

Figura 3.12: Fichero *SQL*

Para la capa de servicio si fue posible definir pruebas automatizadas, ya que se crearon test para todos los métodos asociados a la persistencia del modelo. Para realizarlas se utilizó conjuntamente la capa de repositorio, de forma que fue necesario introducir datos en la base de datos previos a cada test. Esto se realizó mediante un fichero *SQL* (Figura 3.12) que al inicio de cada test cargaba diversos datos en la base de datos y al final de cada test la vaciaba. De esta forma dicha base de datos queda preparada para la consecutiva ejecución de pruebas sin depender de las anteriores.

```

1  @Test
2  @Sql(value = "classpath:ProcessTestsIT.sql")
3  public void testGetProcesses()
4  {
5      PageRequest pageRequest = null;
6      ProcessFilter filter = new ProcessFilter();
7      List<Process> processes = processService.
8          getProcess(filter, pageRequest).asItemsList();
9
10     assertTrue(processes.size() == 2);
11 }

```

Figura 3.13: Ejemplo Test

La Figura 3.13 muestra el caso de prueba para la obtención de todos los *Procesos* existentes en la base de datos. En éste se puede comprobar como se importa el fichero *SQL* (Figura 3.13, Línea 2), y luego se simula una petición a la base de datos a través del servicio apropiado (Figura 3.13, Líneas 5-8). Finalmente se comprueba que se han recibido todos los datos esperados (Figura 3.13, Línea 10). En este ejemplo se usa el concepto de filtro, el cuál permite (como su propio nombre indica) filtrar en función de las propiedades de un *proceso*. Por ejemplo, si en el filtro se introduce un identificador, solo se traerá el *proceso* coincidente con dicho identificador.

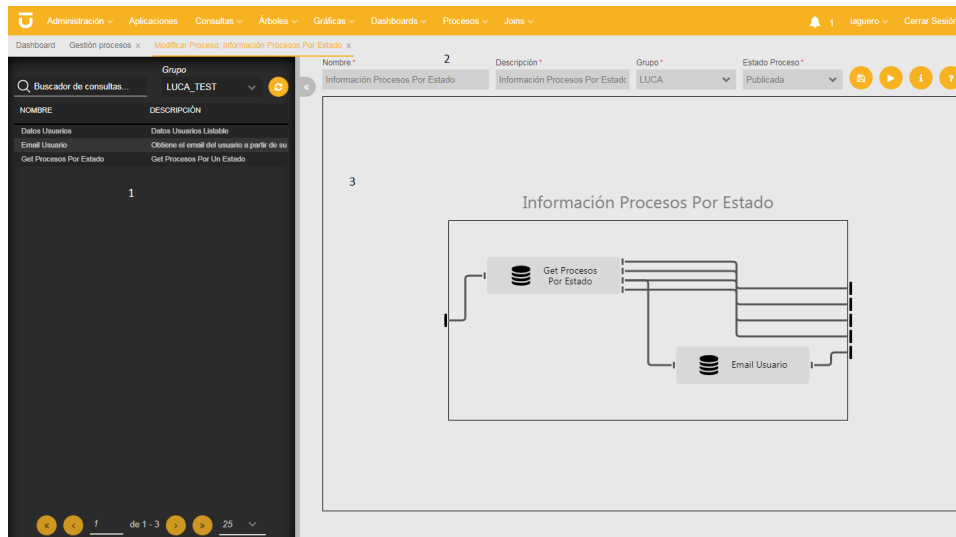


Figura 3.14: Creación de un proceso

La Figura 3.14 ilustra, para cerrar la descripción del editor gráfico, la interfaz de creación de un proceso. La interfaz se compone de tres seccio-

nes bien diferenciadas, por una parte tenemos una tabla con las diferentes consultas que pueden ser utilizadas en el proceso. Esta tabla además, puede ser filtrada en base al grupo al que queremos que pertenezcan las consultas (Figura 3.14, etiqueta 1). La sección superior de la figura (Figura 3.14, etiqueta 2) muestra todas las características necesarias para crear un proceso. La última sección alberga el diagrama con el que el usuario puede interactuar para construir el proceso utilizando las diferentes consultas que son arrastradas desde la tabla de consultas (Figura 3.14, etiqueta 3).

3.5.1. Ejecución de Procesos

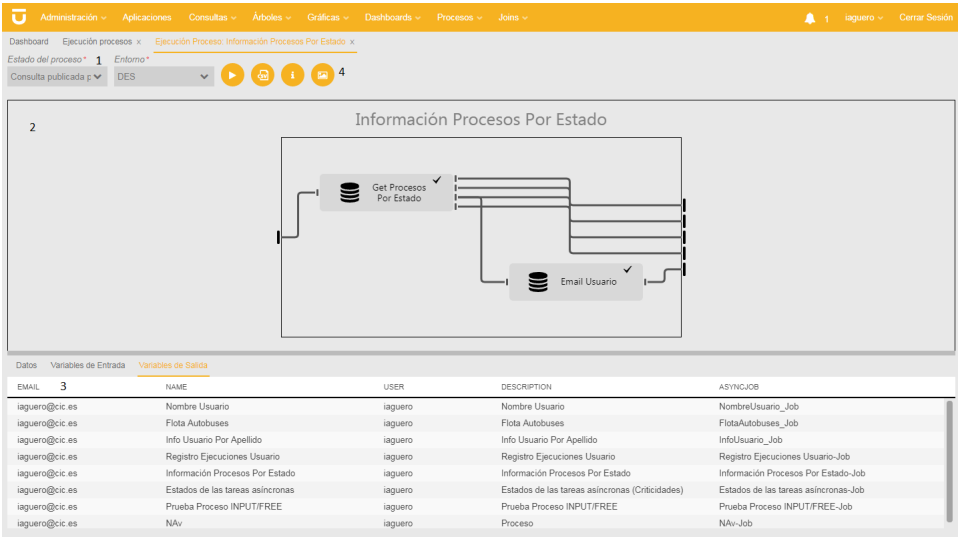


Figura 3.15: Ejecución de un proceso

Para finalizar, la Figura 3.15 muestra la ejecución de un proceso. Esta figura se puede dividir en tres secciones. La primera consta de los campos necesarios para poder ejecutar un proceso (Figura 3.15, etiqueta 1). En la segunda sección se muestra el diagrama del proceso (esta vez no es editable), en él, durante la ejecución, se podrá ir visualizando el flujo de ejecución del proceso, viendo que consultas o enlaces se han ejecutado correctamente o no (Figura 3.15, etiqueta 2). La última sección muestra el resultado de ejecutar el proceso, se visualiza una tabla con el conjunto de resultados (Figura 3.15, etiqueta 3). Además de estas secciones, el usuario si lo desea puede ocultar el diagrama para solo visualizar la tabla de resultados (Figura 3.15, etiqueta 4).

```
1  @Service
2  public class ProcessExecutorImpl implements ProcessExecutor{
3
4      @Override
5      public ResultExecution executeProcess(AbstractState state ,
6      Set<InputVariable> inputVariables ,
7      EnviromentExecution enviromentExecution){
8
9          ResultExecution resultExecution =
10             execute(inputVariables , enviromentExecution);
11
12             ResultExecution sortResultExecution =
13             buildResponse(resultExecution);
14
15             return sortResultExecution;
16         }
17     }
```

Figura 3.16: Ejemplo Servicio de ejecución

La ejecución se delega en un servicio de ejecución (Figura 3.15, Líneas 9-10), cuyo cometido reside en ir recorriendo todas las consultas e ir ejecutándolas. Detallando más, cuando ejecuta una consulta, recoge sus resultados para ser enviados a la siguiente consulta a través de los enlaces del subproceso, una vez que la siguiente consulta ha recibido los datos de todas sus variables de entrada, se ejecuta la nueva consulta, creando así un bucle iterativo hasta que los resultado llegan a las variables de salida del proceso y se establecen sus valores finales. Una vez recibidos los resultados, se ordenan y procesan para que el presentar que los reciba pueda utilizarlos tal y como él los espera (Figura 3.15, Líneas 12-13).

3.6. Sumario

El presente capítulo ha presentado el flujo del desarrollo del proyecto cuyo objetivo era proporcionar al producto LUCA una herramienta gráfica para poder crear *procesos*. El desarrollo se dividió en cuatro etapas iterativas (Figura 3.1): Desarrollo del editor gráfico, integración del editor con *Vaadin*, integración del editor con LUCA y por último la gestión, ejecución y persistencia de los procesos.

Capítulo 4

Sumario, Conclusiones y Trabajos Futuros

4.1. Sumario

El presente documento ha descrito el trabajo realizado para integrar el concepto de *proceso* en LUCA. Gracias a este trabajo es posible ejecutar en LUCA procesos de recuperación de la información que requieran de la ejecución de varias consultas encadenadas, donde las salidas de ciertas consultas sirvan como entradas de otras.

Dado que el trabajo presentado se enmarca dentro del proyecto LUCA, en primer lugar se ha descrito dicho proyecto en profundidad; detallando su objetivo, que es el de proporcionar mecanismos de acceso uniforme a fuentes de datos heterogéneas, y sus limitaciones. Entre dichas limitaciones se encontraba la carencia de un soporte adecuado para ejecutar procesos de recuperación de la información que precisasen de la ejecución de varias consultadas encadenadas. Antes del desarrollo del proyecto, el usuario debía ejecutar cada consulta del conjunto de consultas encadenadas manualmente, guardar sus resultados y manipularlos adecuadamente para posteriormente poder usarlos como entradas para otras consultas.

Para paliar esta deficiencia, los directores de LUCA habían elaborado una especificación de requisitos para un nuevo módulo de LUCA que soportase el concepto de proceso, incluyendo tanto su especificación, la definición del proceso, como su ejecución. Dado que la especificación de requisitos ya estaba hecha, y la arquitectura definida, este proyecto fin de carrera se circunscribe a las fase de diseño detallado, implementación y pruebas.

Una vez definidas la motivación y el alcance del proyecto, se describieron una serie de tecnologías necesarias para entender su funcionamiento. En primer lugar se explicó el funcionamiento del framework *GoJS* cuyo objetivo es facilitar la creación de editores gráficos en Javascript, el cual fue utilizado en el desarrollo del proyecto para la creación del editor gráfico de procesos. A

continuación, se describió la arquitectura de LUCA, que es el proyecto raíz donde se integra el módulo desarrollado en este Trabajo Fin de Grado. Dado que LUCA está construido sobre *Vaadin*, un framework para el desarrollo de aplicaciones web enriquecidas desde código Java, y que *Vaadin* sigue el patrón *Model-View-Presenter*, antes de describir la arquitectura de LUCA se introdujeron ambos elementos.

Finalmente, se describieron las diversas fases del desarrollo del presente proyecto. El proyecto se ha desarrollado de manera iterativa, moviéndonos en cada iteración desde la capa de presentación a la capa de persistencia.

4.2. Resultados

Gracias a este proyecto, LUCA soporta actualmente el concepto de *proceso*. Más concretamente, gracias a este proyecto, LUCA soporta ahora las siguientes funcionalidades.

1. Especificación gráfica de *procesos*, mediante la definición de las consultas que componen dicho proceso y su interconexión.
2. Alteración de la ejecución del flujo de ejecución de un proceso en función de los valores de las salidas.
3. Comprobación de la corrección de la conexión entre salidas y entradas de las consultas que componen un proceso.
4. Ejecución de procesos.
5. Ejecución paso a paso de los procesos.
6. Exportar los resultados de los procesos.

4.3. Conclusiones

La experiencia durante el desarrollo del proyecto ha sido más que satisfactoria, ya que los conocimientos que he aprendido a lo largo del grado, tales como los patrones de diseño, las diferentes arquitecturas software o el despliegue de servicios entre muchos otros, han sido indispensables para poder llevarlo a cabo. Además, debido al uso de nuevos frameworks y herramientas, he sido capaz de ampliar mis conocimientos, a los que he incorporado la utilización de frameworks como *Vaadin* y *GoJS*.

La oportunidad de llevar a cabo el proyecto en la empresa CIC Consulting Informático, me ha permitido adquirir un hábito de trabajo, así como de responsabilidades y fijaciones a horarios que no se puede aprender de ninguna otra forma. Además, el apoyo recibido por el equipo de LUCA ha sido excepcional y he podido aprender mucho con ellos.

4.4. Trabajos Futuros

Con el proyecto integrado en LUCA, se espera poder realizar el trabajo realizado en futuras versiones del producto. Concretamente, se plantea la posibilidad de soportar procesos anidados, es decir, la capacidad de utilizar procesos ya definidos como si fuesen consultas simples para la definición de un nuevo proceso. Es decir, permitir que los procesos puedan contener procesos. También se plantea la posibilidad de crear consultas desde una consola gráfica, utilizando el editor gráfico, ya que en algunos casos este estilo de especificación puede resultar más conveniente.

Apéndice A

Tratamiento de eventos de la clase Diagram Component

A continuación se muestran el resto de eventos que son tratado en el ejemplo del editor gráfico y su integración con Vaadin.

```
1  addFunction("SelectionMoved", new JavaScriptFunction()
2  {
3
4      @Override
5      public void call(JsonArray arguments)
6      {
7
8          List<Node> nodes = getState().getNodes();
9          for (Node node : nodes)
10         {
11             node.update();
12         }
13         System.out.println("SelectionMoved at " + arguments.toJson());
14     }
15 });
```

Figura A.1: Evento de elemento movido

APÉNDICE A. TRATAMIENTO DE EVENTOS DE LA CLASE DIAGRAM COMPONENT

```
1  addFunction("TextEdited", new JavaScriptFunction()
2  {
3
4      @Override
5      public void call(JsonArray arguments)
6      {
7
8          Map<String, String> map = new HashMap<>();
9          if(arguments != null)
10         {
11             try
12             {
13                 String srt = arguments.toJson();
14                 map = JsonUtils.jsonToMap(
15                     srt.substring(1, srt.length() - 1));
16             }
17             catch (JSONException e)
18             {
19                 e.printStackTrace();
20             }
21         }
22         List<Node> nodes = getState().getNodes();
23         for (Node node : nodes)
24         {
25             if(map.get("oldValue").equals(node.getName()))
26             {
27                 node.setName(map.get("newValue"));
28             }
29         }
30         System.out.println("TextEdited at " + arguments.toJson());
31     }
32 });
```

Figura A.2: Evento de edición de texto

```
1  addFunction("ObjectDoubleClicked", new JavaScriptFunction()
2  {
3
4      @Override
5      public void call(JsonArray arguments)
6      {
7
8          System.out.println("ObjectDoubleClicked at "
9          + arguments.toJson());
10     }
11 });
```

Figura A.3: Evento de doble selección

Bibliografía

- [1] CAULES, C. A. *Arquitectura Java JPA Domain Driven Design*. arquitecturajava, 2014. 33
- [2] GÓMEZ, M. R. *Curso de Desarrollo Web: HTML, CSS y JavaScript*. ANAYA MULTIMEDIA, 2017. 20
- [3] HOLAN, J., AND KVASNOVSKY, O. *Vaadin 7 Cookbook*. Packt Publishing, 2013. 16
- [4] KAY, R. Rich internet application. <https://www.computerworld.com/article/2551058/networking/rich-internet-applications.html>. Visto en: 02-14-2018. 16
- [5] OCAÑA, A. S. *Programación Ajax y jQuery*. No Starch Press, 2014. 20
- [6] SHIFLETT, C. *HTTP Developer's Handbook*. Sams Publishing, 2003. 20
- [7] SOFTWARE, N. Gojs. <https://gojs.net/latest/index.html>. Visto en: 02-14-2018. 9