FACULTAD DE CIENCIAS

UNIVERSIDAD DE CANTABRIA Facultad de Ciencias

Proyecto Fin de Carrera

Desarrollo de widgets integrables en aplicaciones de análisis y visualización de datos

Data analysis and visualization widgetset development

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Autor: Fernando Magaldi Calleja
Director: Pablo Sánchez Barreiro
Codirector: Max Tuni San Martín

Septiembre - 2015

Agradecimientos

En primer lugar como no podía ser de otro modo, mi gratitud es hacia mis padres. Desde pequeño me enseñaron que la constancia y la voluntad son grandes compañeros de viaje y que todo esfuerzo por duro que parezca, con el tiempo acaba teniendo su recompensa.

Sin ellos y su apoyo incondicional a lo largo de todos estos años no habría conseguido llegar a donde estoy hoy. Gracias a su esfuerzo y los valores en los que me educaron, he podido convertirme en la persona que soy.

También he de agradecer el apoyo recibido por parte de mis jefes en la Empresa Predictia, Daniel San Martín y Max Tuni. Desde un primer momento se mostraron dispuestos a resolverme cualquier duda y dedicarme el tiempo que fuera necesario. Con ellos tuve mi primer contacto con el mundo empresarial y consiguieron gracias a su amabilidad y comprensión que no fuera un salto brusco, sino todo lo contrario, incluso en algún momento divertido.

Asimismo, tengo que hacer mención a todos aquellos profesores que han pasado por mi vida y tantas cosas me han enseñado.

Por último, me gustaría dedicar unas palabras a todos mis compañeros de clase. Cinco años son muchos, hemos compartido bastantes momentos juntos, momentos difíciles, de tensión y nervios, pero también de alegría y felicidad. Echando la vista atrás, parece que fuera ayer cuando siendo unos críos entrábamos en la universidad sin conocernos, hablando con timidez los unos con los otros. Con el tiempo se fueron forjando vínculos de amistad entre nosotros y es por ello que más que simples compañeros, les puedo considerar amigos.

Aunque algunos de nosotros ya hemos entrado en una nueva etapa en la vida y la distancia que nos separa puede ser grande, puedo afirmar con seguridad que el recuerdo que tengo y tendré sobre ellos es muy gratificante.

Resumen

El proyecto de fin de carrera consiste en desarrollar una serie de widgets que permiten analizar y visualizar información de manera intuitiva para el usuario. Principalmente, se desarrollaron widgets de filtrado de información y visualización a través de mapas, así como configuración del metadato de la aplicación.

Palabras clave: widget, Vaadin, gestión de metadato , filtrado de información, mapas.

Summary

The final career project lays out the development of a widgetsets that analyze and display information to the user intuitively. Mainly, these widgets can filter and display information on maps and manage metadata configuration of the application.

Keywords: widget, Vaadin, metadata management, information filtering, maps.

Índice general

Índice de ilustraciones

1 Introducción

El propósito del presente proyecto es el desarrollo de un conjunto de módulos suficientemente genéricos y reutilizables para la gestión y consulta de datos almacenados en arquitecturas empresariales distribuidas sobre la web.

Dicho conjunto quedará a disposición de la empresa Predictia, la cual lo utilizará para el desarrollo de aplicaciones de análisis de datos dentro de diferentes dominios.

Estos módulos de control gráfico, pasarán a ser llamados a partir de ahora *widgets*, pues en el ámbito web, se conocen de esta manera.

Normalmente, las aplicaciones empresariales o sistemas de información almacenan una serie de datos organizados en diversas entidades u objetos de dominio.

Dado que la mayoría de los sistemas de información empresariales, se desarrollan actualmente como sistemas distribuidos sobre la Web, el conjunto de widgets a desarrollar, también serán distribuidos sobre la Web conforme a arquitecturas multicapa empresariales.

Para favorecer la amigabilidad de su interfaz gráfica, se hará uso de las técnicas RIA (***R**ich **I**nternet **A**pplication*), cuya meta es trasladar las comodidades y facilidades que ofrecen las aplicaciones gráficas de escritorio a la web, incluyendo mejoras tanto en la experiencia visual, como en la conectividad y despliegue instantáneo de la aplicación, agilizando la comunicación, pues el intercambio de información solo se produce cuando se necesitan datos externos. Estos widgets se deberán integrar además con los frameworks y principios arquitectónicos de la empresa *Predictia*, que será la que utilizará dicho módulo.

La idea básica de esta serie de widgets consiste, en primer lugar en mostrar las entidades almacenadas en la aplicación donde se haya integrado, para una vez seleccionado un objeto de dominio, indicar qué acción se desea realizar sobre dicho objeto, y posteriormente, en el caso de ser oportuno, listar o representar los resultados obtenidos por dicha acción.

Pongamos un ejemplo, dado el supuesto que nuestra aplicación se dedica a gestionar los diferentes pacientes atendidos por el conjunto de hospitales de una comunidad autónoma en concreto.

Si el usuario deseara conocer la cantidad de pacientes que han sido atendidos en un rango de fechas en concreto, debería realizar la siguiente secuencia:

- En primer lugar, seleccionaría la entidad *Paciente*, la cual suponemos que tiene un atributo *FechaConsulta*
- Una vez seleccionada dicha entidad, el usuario indicaría que desea obtener un listado de pacientes, de entre todas las métricas posibles.
- El widgets mostraría entonces un cuadro de diálogo donde se pudiese seleccionar los diferentes atributos que tiene la entidad paciente y aplicar distintos tipos de filtros sobre ellos.
- En este caso, el usuario seleccionaría el atributo *FechaConsulta* e indicaría que desea aplicar un filtro tipo “Dentro de Rango”.

- Una vez especificado el filtro, o los filtros, en caso de que se quieran especificar varios, se procesaría la consulta, convirtiéndola a la secuencia de acciones que fuese necesaria, y se obtendrían los resultados.
- Finalmente, dichos resultados se visualizarían de forma amigable al usuario, para su fácil compresión mediante gráficas, mapas y tablas.

1.1 Objetivos del proyecto de fin de carrera

De toda la funcionalidad vista en el ejemplo, los objetivos principales a desarrollar en el proyecto son tres:

- El primero, es la selección de los diferentes tipos de filtros aplicables a los datos de entrada, según sea el tipo de variable a procesar (Dato numérico, de tipo discreto, texto, fechas, etc) mediante una interfaz amigable que permita al usuario con una serie de clics realizar estos filtros o consultas complejas sin necesidad de conocer el lenguaje de consulta a la base de datos del que dispone y sus particularidades

  .
- El segundo objetivo, consiste en la gestión del propio metadato (definición del dato en sí) de la base de datos. Por ejemplo, podremos definir el tipo de una variable, así como otras características: su rango, si admite nulos, los posibles estados de los que dispone, etc. También se podrá configurar toda la información referente al posicionamiento geográfico asociado a estas variables.

- El último de los objetivos se centra en la visualización de los datos sobre un mapa, una vez han sido procesados, es decir, una vez han sido filtrados y listados o agrupados eligiendo alguna de las métricas existentes.

  Para ello se han definido una serie de widgets encargados de mostrar la componente geográfica asociada a los posibles valores de la variable analizada sobre un mapa. Esta representación geográfica podrá trabajar tanto con coordenadas (puntos en el mapa) como con polígonos. Además se dispone de gráficos sectoriales si se va a representar varias dimensiones sobre cada elemento en el mapa.

Todo ello ha sido desarrollado mediante el framework de desarrollo Vaadin basado en el lenguaje Java, el cual se explicará con más detenimiento en la siguiente sección.

2 Antecedentes tecnológicos

2.1 Vaadin ¿Qué es?

Tradicionalmente, la creación de aplicaciones web visuales ha requerido una gran cantidad de tiempo, conocimiento y código repetitivo. Muchos framework de Java actualmente existen para hacer la vida del desarrollador mejor y más fácil, entre los que se encuentra Vaadin.

Vaadin es un framework Open Source para el desarrollo de aplicaciones web modernas o también llamadas RIA, que posee una librería con numerosos módulos con los que construir aplicaciones web profesionales, además de incorporar programación dirigida por eventos.

Vaadin soporta dos modelos distintos de programación: en el lado del servidor y en el lado del cliente. La programación del lado del servidor es la más potente, permitiendo olvidar el desarrollo web con HTML y JavaScript, y programar las interfaces de usuario de una forma más cercana al clásico desarrollo de las interfaces de una aplicación de escritorio, casi como si estuviéramos usando las librerías de java AWT, Swing o SWT, pero de forma más fácil.

Este tipo de programación permite que la mayor parte de la lógica y por tanto la mayor carga del trabajo recaiga sobre el servidor.

En el lado del cliente, Vaadin usa AJAX (**A**synchronous **J**avaScript **A**nd **X**ML) para la comunicación entre el navegador y el servidor, utilizando como soporte GWT (Google Web Toolkit), el cual proporciona un compilador de Java a JavaScript, así como un marco de interfaz de usuario con todas las funcionalidades. Con este enfoque, Vaadin es completamente Java en ambos lados.

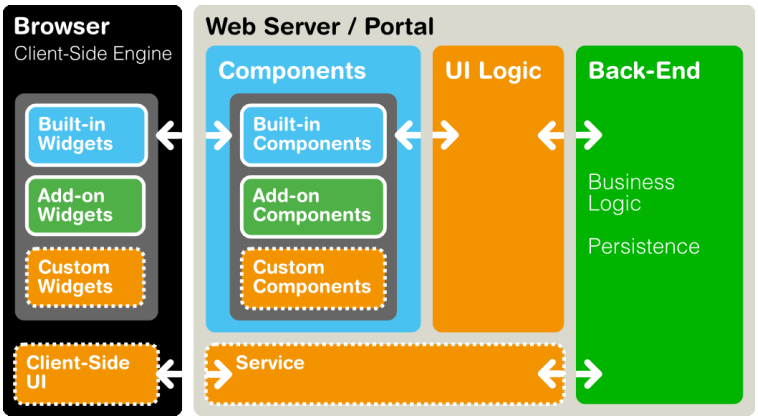

Ilustración 2.1 Arquitectura de las aplicaciones Vaadin (Grönroos, 2014)

Ilustra la arquitectura básica de las aplicaciones web hechas con Vaadin. La arquitectura de la aplicación del lado del servidor se basa en el framework del lado del servidor (server-side framework) y en un motor del lado del cliente (client-side engine). Este motor se ejecuta en el navegador como código JavaScript, renderizando la interfaz de usuario, y entregando la información de las interacciones de usuario al servidor. La lógica de interfaz de usuario (UI) de una aplicación se ejecuta como un Servlet Java en el servidor de aplicaciones Java.

Las aplicaciones Vaadin por defecto funcionan en modo SPA (*Single Page Application*), es decir, todo el contenido, páginas web, formularios, etc de la aplicación son cargados en una sola página web, perdiendo el soporte de navegación (poder desplazarse adelante/atrás mediante el navegador, guardar una página concreta de la aplicación como favorita, etc), pero basta con configurar unos simples parámetros, para poder disponer de ello, como si de una página web normal se tratara

El entorno Vaadin define una clara separación entre la estructura de la interfaz de usuario y su apariencia, y permite desarrollarlos separadamente, es por ello, que el diseño de la aplicación se personaliza fácilmente con CSS o plantillas HTML de manera que el resultado final es puro HTML5.

2.1.1 Desarrollo en el lado del servidor

Ahora que tenemos una idea general de lo que es Vaadin, vamos a centrarnos en cómo sería desarrollar simplemente la parte del lado del servidor.

Para ello, Vaadin trae implementados un conjunto de widgets que, junto con un pequeño número de addons (widgets personalizados creados por la comunidad de usuarios) nos facilitan este desarrollo, pues a partir de ahora desarrollarán widgets creados mediante la composición de widgets simples que ya incorporan tanto la lógica de cliente como de servidor.

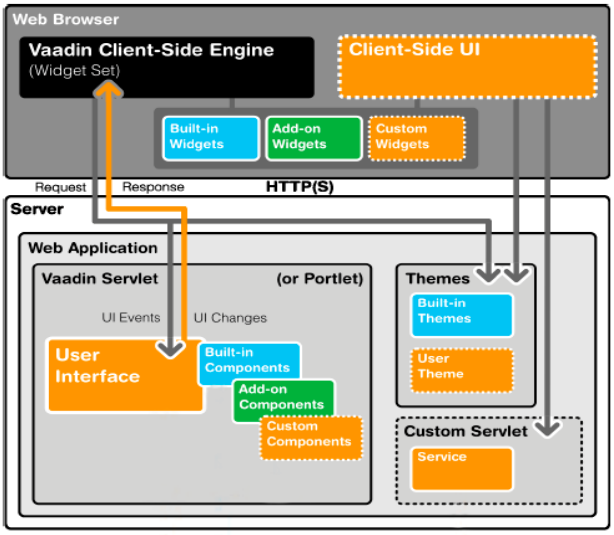

Ilustración 2.2 Arquitectura de ejecución en Vaadin (Grönroos, 2014)

Muestra una ilustración básica de las comunicaciones del lado del cliente y del lado del servidor, en un escenario de ejecución donde la página con el código del lado del cliente (motor o aplicación) ha sido inicialmente cargada por el navegador.

Una aplicación web Vaadin se ejecuta como un servlet en un servidor web Java, sirviendo peticiones HTTP. El servlet recibe las peticiones del cliente y las interpreta como eventos para una sesión de usuario concreta. Los eventos se asocian con los componentes de la interfaz de usuario y se propagan tanto en el lado del servidor como en el cliente. Si la lógica de la UI hace cambios en los componentes de la interfaz de usuario del lado del servidor, el servlet los renderiza en el navegador web generando una respuesta. El motor del lado del cliente que se ejecuta en el navegador recibe las respuestas y las utiliza para hacer los cambios que sean pertinentes en la página del navegador.

Los widgets de Vaadin emplean la técnica AJAX para obtener desde el cliente la información necesaria del servidor para renderizar la interfaz de usuario en el navegador.

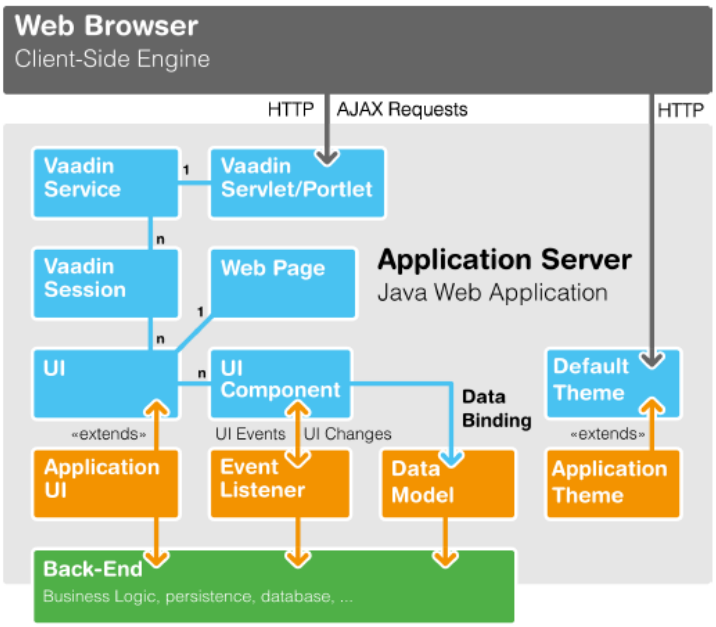

Ilustración 2.3 Arquitectura de las aplicaciones en el lado del servidor (Grönroos, 2014)

La interfaz de usuario de la aplicación se define a partir de clase de "com.vaadin.ui.*UI*", que permite definir en su inicialización los componentes de la interfaz de usuario. La entrada de usuario se maneja con oyentes de eventos, aunque también es posible enlazar los componentes de la interfaz de usuario directamente a los datos. El estilo visual de la aplicación se define en temas vía fichero CSS o SCSS. Los iconos, otras imágenes, y ficheros descargables son manejados como recursos, que pueden ser externos o servidos por el propio servidor de aplicaciones

Una UI representa un fragmento de HTML, en el cual, una aplicación de Vaadin se ejecuta en una página. Normalmente ocupa la página por completo, pero también puede ser sólo parte de ella.

De forma habitual, cuando el usuario abre una nueva página con la URL de la *UI* de Vaadin, un nuevo objeto de tipo *UI* es automáticamente creado para él, compartiendo sucesivas peticiones del mismo navegador, así como la misma sesión de usuario.

A continuación, se mostrará un pequeño ejemplo sobre cómo crear una UI:

```
1 @Title("Hello Window")
2 public class HelloWorld extends UI {
3     @Override
4     protected void init(VaadinRequest request) {
5           VerticalLayout content = new VerticalLayout();
6           setContent(content);
7           content.addComponent(new Label("Hello World!"));
8     }
9 }
```

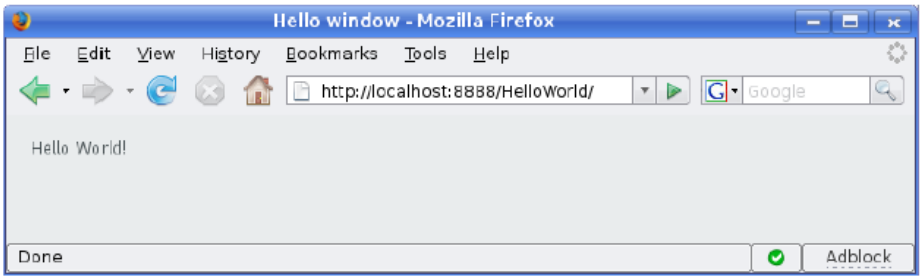

Ilustración 2.4 Ejemplo sencillo con la demostración de "hello world" (Grönroos, 2014)

2.2 Google Web Toolkit

GWT es un conjunto de herramientas de desarrollo para crear y optimizar aplicaciones complejas basadas en el navegador. Su objetivo es permitir el desarrollo productivo de las aplicaciones web de alto rendimiento sin que el desarrollador tenga que ser un experto en las peculiaridades de cada navegador o en JavaScript.

Por ello, GWT nace con la misión de mejorar radicalmente la experiencia web de los usuarios, al permitir a los desarrolladores utilizar las herramientas de Java para crear JavasScript.

Respecto a Vaadin, el marco de Vaadin del lado del cliente está basado en Google Web Toolkit (GWT). Los módulos del lado del cliente se desarrollan en Java y se compilan a JavaScript con el compilador de Vaadin, que es una extensión del compilador de GWT. El marco del lado del cliente también oculta mucha de la manipulación del DOM (Document Object Model) de HTML y habilita el manejo de eventos del navegador en Java.

GWT es esencialmente una tecnología del lado del cliente, normalmente usada para desarrollar la lógica de la interfaz de usuario en el navegador web. Los módulos del lado del cliente puros, siguen necesitando comunicarse con el servidor usando llamadas *RPC* (***R**emote **P**rocedure **C**all***)** y serializando algunos datos. El modo de desarrollo dirigido por el servidor en Vaadin oculta de manera efectiva todas las comunicaciones entre

cliente y servidor y permite manejar la lógica de la interacción del usuario en una aplicación del lado del servidor. Esto hace la arquitectura de las aplicaciones web basadas en *AJAX* mucho más sencilla.

En resumen, GWT es un compilador, preprocesador, enlazador y optimizador completo, que permite separar la mantenibilidad del código de la efectividad del ejecutable para no comprometerlas, de tal manera que el código resultante esté muy comprimido, ofuscado y optimizado, permitiendo una rápida ejecución en el navegador pero a la vez, que el código fuente pueda ser legible en el entorno de desarrollo elegido, como por ejemplo Eclipse.

2.3 Spring framework

Spring en una plataforma Java que sirve para construir aplicaciones modernas de nivel empresarial con un amplio catálogo de funcionalidades.

Spring maneja la infraestructura de tal manera que permite al desarrollador centrarse en su aplicación mientras mantiene un control total en el despliegue de las partes que necesita.

Si bien las características fundamentales de Spring Framework pueden ser usadas en cualquier aplicación desarrollada en Java, existen variadas extensiones para la construcción de aplicaciones web sobre la plataforma Java EE. A pesar de que no impone ningún modelo de programación en particular, este framework se ha vuelto popular en la comunidad al ser considerado una alternativa, sustituto, e incluso un complemento al modelo EJB (Enterprise JavaBean).

Dicho modelo creado por Oracle detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor, que son, precisamente, los EJB.

Por tanto, el objetivo de los EJB es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, etc) para centrarse en el desarrollo de la lógica de negocio en sí.

El marco de desarrollo Spring está compuesto por distintos módulos que se pueden agregar a las aplicaciones Java, siendo el desarrollador el encargado de utilizar los que necesite.

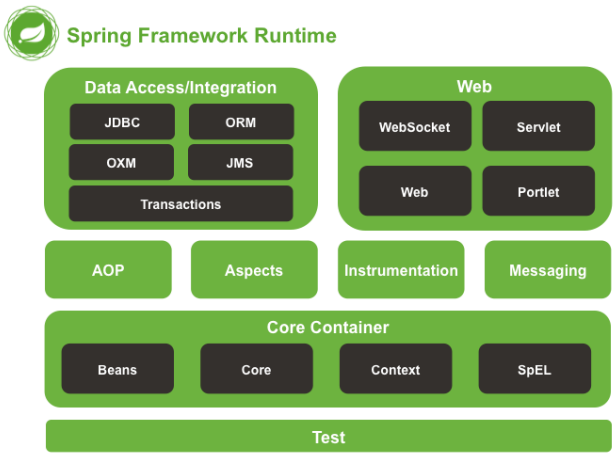

Ilustración 2.5 Módulos por los que está compuesto Spring (Spring-framework, 2015)

El contenedor del núcleo proporciona las características fundamentales del marco de trabajo de Spring, tales como la inversión de control y la inyección de dependencias.

(La inyección de dependencias surge como solución al problema que causa tener múltiples objetos relacionados entre sí mediante composición, pues para enlazar dichos objetos, se tendría que inyectar en uno de ellos una instancia del otro)

Es por ello, que los objetos de nuestra aplicación no buscan o crean sus dependencias, sino que es Spring quién se encarga de administrarlas, instanciando los objetos, inyectándolos mediante reflexión y gestionando su ciclo de vida.

El módulo de instrumentación y *AOP* (***A**spect-**O**riented **P**rogramming*), proveen una implementación para este paradigma de programación, permitiendo desacoplar el código de las funcionalidades transversales, eliminando por tanto la repetición innecesaria de código. Este módulo proporcionar por ejemplo los loggers, la seguridad, el manejo de las transacciones, etc.

La mensajería, proporciona un registro configurable de objetos receptores de mensajes.

El acceso a datos provee una capa de abstracción sobre *JDBC*, abstrae el código de acceso a datos de una manera simple. Cuenta con una capa de excepciones sobre los mensajes de error provistos por cada servidor específico de base de datos, además provee capas de integración para APIs de mapeo objeto - relacional, incluyendo, JDO, Hibernate e iBatis. Usando el paquete ORM es posible usar esos mapeadores en conjunto con otras características que Spring ofrece, como la administración de transacciones mencionada con anterioridad.

El módulo web, proporciona clases especiales orientadas al desarrollo web, tales como funcionalidad multipartes (para realizar la carga de archivos) junto con un framework basado en HTTP y servlets, que provee herramientas para la extensión y personalización de aplicaciones web.

El módulo de test, da soporte para el desarrollo de unidades de prueba e integración.

3 Arquitectura

3.1 Metodología

La metodología aplicada para el desarrollo del proyecto podría considerarse como desarrollo incremental, empleando metodologías de tipo ágil. Al tratarse Predictia de una empresa pequeña, las comunicaciones pueden realizarse cara a cara y por tanto se da una mayor importancia a la generación de código simple y en la medida de lo posible auto-explicativo frente a la creación de documentación técnica, pues es más primordial la creación de software funcional de manera frecuente que la generación de documentación extensiva.

Respecto al modelo incremental usado durante el desarrollo del proyecto, conviene matizar ciertos aspectos.

Es bien sabido, que el modelo incremental tiene como principio realizar una serie de desarrollos en cascada o secuenciales incluyendo: análisis de requisitos, diseño, implementación y pruebas, en donde cada secuencia sirve para completar una pequeña parte del sistema, antes de comenzar con la siguiente iteración.

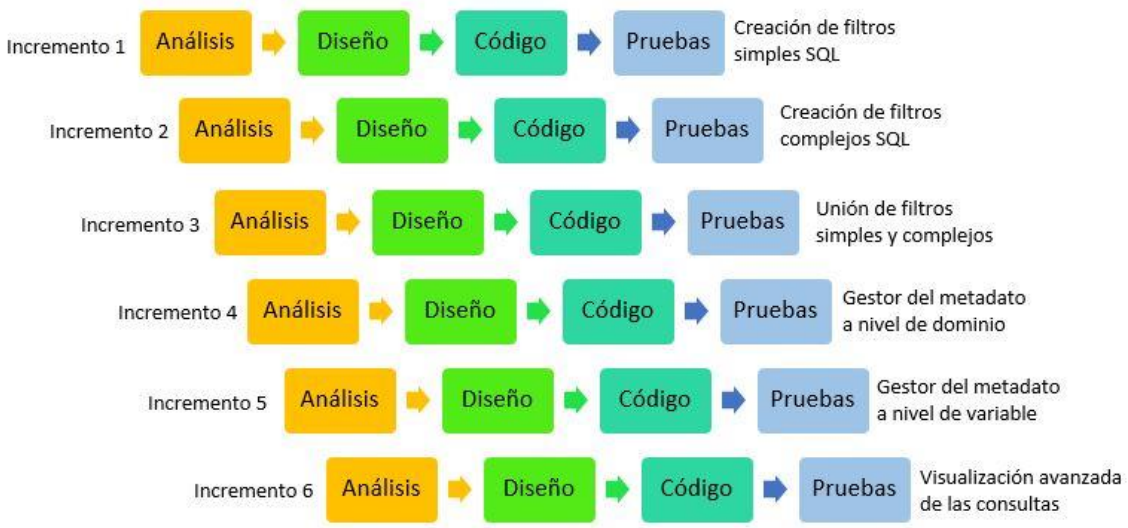

Ilustración 3.1 Ejemplo del modelo de desarrollo incremental

En nuestro caso, al tratarse del desarrollo de widgets, relativamente independientes entre sí, en cada iteración o incremento, se ha producido un nuevo widget a añadir al software, dando como resultado un incremento en la aplicación final.

3.2 Visión general

En esta sección se va a describir la arquitectura utilizada por la empresa Predictia para el desarrollo del proyecto.

La elección de las diferentes tecnologías que forman parte de la arquitectura, está basada en estándares ampliamente extendidos y aceptados en el mercado, así como en la propia experiencia que supone haber experimentado con antelación las ventajas e inconvenientes de cada una de ellas.

La arquitectura propuesta se basa en un modelo de tres capas, compuesto por: la capa de acceso a datos, la capa de servicios y la capa web.

La capa de acceso a datos, se encarga principalmente de almacenar y recuperar objetos de dominio persistentes, la capa de servicios engloba las reglas de negocio del sistema, y la capa web contiene las vistas y los controladores.

Para el desarrollo de esta arquitectura, se han utilizado las siguientes tecnologías:

- Jooq y Neo4j en lo relacionado con la persistencia de datos
- Spring para la inyección de dependencias, la integración de servicios, el acceso a los datos , la gestión de transacciones, la autenticación y la programación orientada a aspectos
- Vaadin para el renderizado de la interfaz de usuario en la capa web

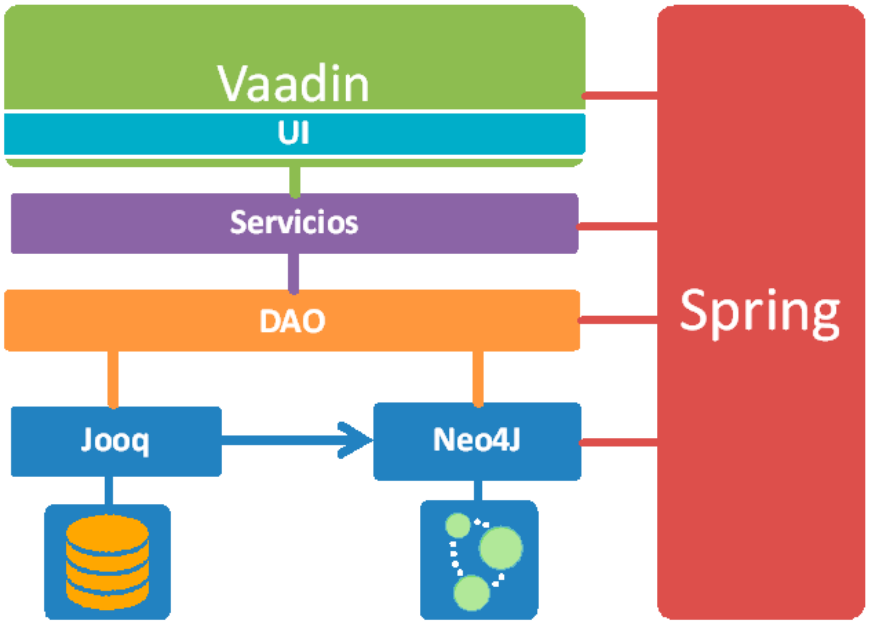

Ilustración 3.2 Arquitectura del proyecto

3.3 Capa de acceso a datos:

Comencemos explicando la arquitectura de abajo a arriba, es decir, por la capa de acceso a datos.

En esta capa, nos encontramos con dos tipos de bases de datos distintas, por un lado tenemos una BBDD clásica de tipo relacional en donde se encuentran almacenados los datos que el cliente proporcione. Para su manejo se ha optado por el framework *JOOQ* (***J**ava **O**bject **O**riented **Q**uerying*), que no es más que es una ligera biblioteca de software de mapeo de bases de datos en Java que implementa el patrón de registro activo. Su propósito es ser a la vez relacional y orientada a objetos al proporcionar un lenguaje de dominio específico para construir las consultas a partir de clases generadas desde un esquema de base de datos (ilustración 3.3).

Ilustración 3.3 Equivalencia de la construcción de una consulta SQl mediante JOOQ (Neo4j, 2015)

El otro tipo de BBDD del que se dispone es una base de datos orientada a grafos, utilizada para almacenar la información del metadato (información sobre el propio dato en sí) de la base de datos relacional.

A diferencia de una base de datos relacional, en las orientadas a grafos, la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se pueda usar teoría de grafos para recorrer la base de datos, ya que ésta puede describir atributos de los nodos (entidades) y las aristas (relaciones).

La implementación elegida de BBDD de tipo grafos es Neo4j, que es un motor de persistencia embebido, implementado en java, basado en disco y completamente transaccional.

La elección de una BBDD "NoSQL", ha sido debido a la necesidad no sólo de almacenar datos "sueltos", sino que una parte importante de dichos datos eran las relaciones entre ellos, y en este caso, si pensamos en términos relacionales, el esquema se empieza a complicar rápidamente en cuanto el número de relaciones aumenta en gran medida, siendo aquí donde una base de datos orientada a grafos cobra sentido.

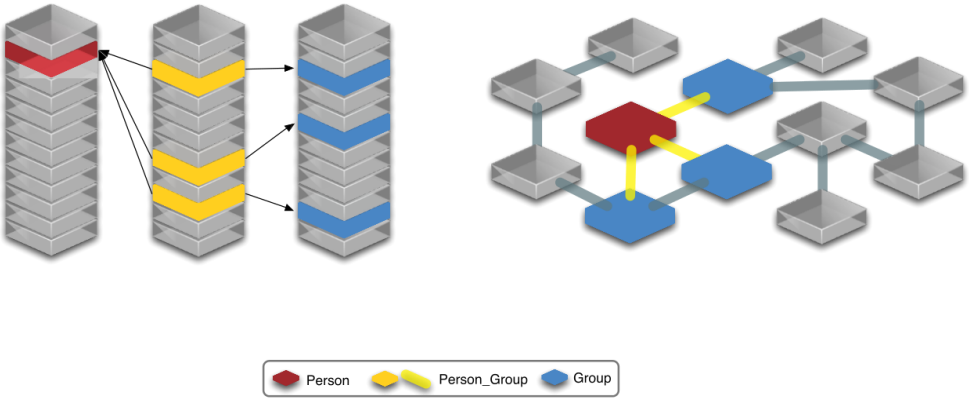

Ilustración 3.4 Diferencia entre el amacenamiento en una BBD relacional frente a una orientada a grafos (Neo4j, 2015)

Como se puede apreciar en la ilustración 3.4, para el desarrollo de la arquitectura se ha usado el patrón *DAO* (***D**ata **A**ccess **O**bject*) para conseguir que los componentes de la aplicación fueran transparentes en la medida de lo posible al actual sistema de persistencia o fuente de datos, es decir, abstraer y encapsular el acceso a los mismos.

3.4 Spring

El siguiente componente a tratar, siguiendo la disposición de capas de la ilustración 3.2 sería Spring, que aunque se encuentra disponible en todas ellas, conviene explicarlo antes de seguir adelante.

Como ya se ha explicado en los antecedentes tecnológicos, Spring está formado por diversos módulos que proveen un amplio rango de servicios, de los cuales, nosotros hemos utilizado los siguientes:

- Spring framework (el núcleo) para la inyección de dependencias y la gestión de servicios. Este módulo cohesiona los distintos elementos de los que conforman la arquitectura, es decir, es el "pegamento" que une las distintas partes de cada capa entre sí.
- Spring data para facilitar el acceso a las nuevas tecnologías de acceso a datos, como las bases de datos no relacionales, además de tener un soporte mejorado para las bases de datos relacionales.
- Spring security para la autenticación y el control de acceso, siendo fácilmente extensible para satisfacer las necesidades del cliente.
- Spring AOP para proveer lógica transversal y reutilizable entre diferentes clases, como la gestión de transacciones y el caché. Esto se consigue gracias a su integración con Apache *AspectJ*, librería que implementa el paradigma de programación orientada a aspectos

3.5 Infraestructura

La elección de la infraestructura software en el desarrollo de un proyecto, siempre es un punto importante a tener en cuenta, ya que de su elección dependerá en cierta medida la calidad del software desarrollado. Si los distintos módulos que conforman la infraestructura están bien integrados entre ellos y además aportan facilidades al desarrollador que de otro modo no tendría, esto implica que la opción escogida ha sido la correcta.

De entre todas las alternativas disponibles, ya sea por familiaridad de uso o por motivos técnicos de la propia empresa, Predictia eligió la infraestructura representada en la ilustración 3.5.

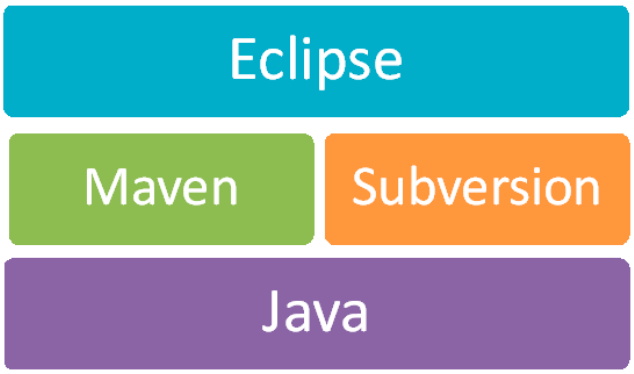

Ilustración 3.5 Infraestructura principal del proyecto

Como entorno de desarrollo, el elegido ha sido Eclipse, pues en cuanto a las herramientas disponibles para el desarrollo en Java pocos IDEs pueden hacerle frente, gracias al conjunto de características que lo componen, tales como:

- ✓ Editor de texto con resaltado de sintaxis.
- ✓ Compilación en tiempo real.
- ✓ Depuración del código
- ✓ Pruebas unitarias

A la hora de realizar la compilación y el empaquetado del código, siempre es costoso tener que tratar manualmente con la gestión de las dependencias de los distintos módulos o componentes externos, y por ello, se ha utilizado la herramienta Maven para automatizar dicha tarea.

Maven utiliza un *POM* (***P**roject **O**bject **M**odel*) para describir el proyecto software a construir, sus dependencias y el orden de construcción de los elementos.

A su vez, para poder tener un control de versiones del propio código desarrollado de tal forma que estuviera integrado dentro de Eclipse, se ha utilizado el módulo de Subversion (SVN).

Subversion es una herramienta de control de versiones basada en un repositorio cuyo funcionamiento se asemeja enormemente al de un sistema de ficheros. Utiliza el concepto de revisión para guardar los cambios producidos en el repositorio, lo que permite que pueda ser usado por diferentes personas de forma simultánea, fomentando la colaboración.

3.6 Estructura de la base de datos

Una de las partes más importantes en el desarrollo de una aplicación suele ser la disposición y estructura de la base de datos, pues sobre ella se construirán todos los demás elementos. Sin embargo, la aplicación que está desarrollando Predictia puede conectarse a diferentes bases de datos (cada una con diferentes estructuras), por lo que el modelo de datos no se puede apoyar en una estructura concreta de base de datos.

La aplicación se puede conectar a un conjunto cualquiera de datos, situados físicamente en una base de datos accesible por medio de un conector JDBC. Esta base de datos concreta con los datos de un cliente la denominaremos "Dominio". Los metadatos sobre este Dominio (el esquema de tablas y columnas) se almacenarán en la aplicación mediante una estructura de tipo grafo y podrán ser configuradas por parte del usuario, mediante el concepto "Variable".

Una variable representa una columna en una tabla en la base de datos, con la ventaja de poder ser reutilizable entre diferentes Dominios y contener información adicional como los posibles valores esperados en dicha columna, o por ejemplo información geográfica sobre cada valor concreto representado en una fila concreta de una columna.

Esta división entre datos y metadatos proporciona un nivel más de abstracción, permitiendo el uso de cualquier BBDD heterogénea como principal soporte de almacenamiento.

Todas las explicaciones sobre widgets que por complejidad o para mayor claridad necesitan de un ejemplo para su visualización se harán sobre una de las tres tablas mostradas en la siguiente figura:

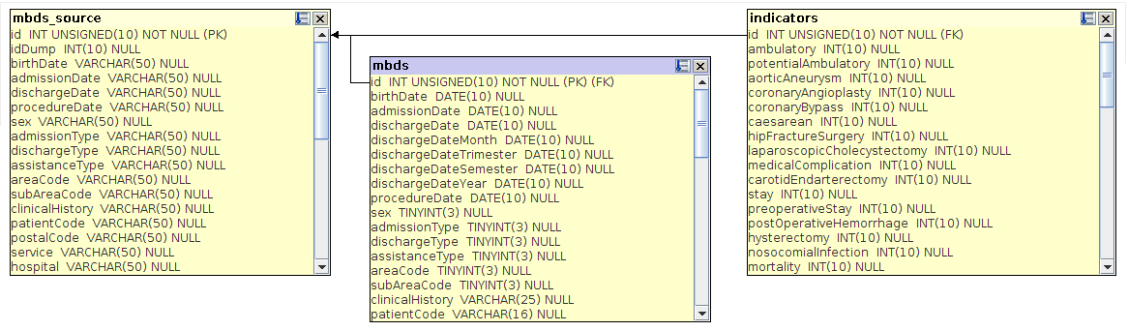

Ilustración 3.6 Representación de las principales tablas y atributos de la BBDD

Tal y como se aprecia en la ilustración superior, tanto la tabla con nombre “mbds” como “indicators” dependen de la tabla “mbds_source”. Esto es debido a que esta tabla contiene información de la base de datos sin procesar (todas las columnas son de tipo texto), mientras que “mbds” contiene la misma información pero procesada, con el tipo apropiado para cada columna.

Por otra parte, la tabla “indicators” contiene una serie de indicadores (métricas), asociados a cada registro de las tablas originales.

4 Formación de filtros

Como ya se ha descrito en la introducción, el primer objetivo del proyecto es la realización de consultas sobre los contenidos de una BBDD de la forma más amigable y simple posible, y para ello se necesita poder realizar filtros sobre la información a consultar.

Para un usuario medio, definir un filtro puede resultar engorroso, ya que además de conocer el modelo de la BBDD, es necesario conocer la sintaxis de consulta, pudiendo tener particularidades en función del motor que se esté empleando. Además, un usuario puede no recordar los posibles valores existentes en las diferentes columnas de la BBDD, por lo que en ocasiones para definir un filtro se necesita una consulta adicional para acceder a esta lista de valores.

Para evitar en la medida de lo posible estos inconvenientes, se han dispuesto una serie de widgets que proporcionan los elementos necesarios para realizar filtros sobre una consulta de una forma visual. Esto permite al usuario ir componiendo el filtro con tan solo rellenar o elegir de entre las opciones disponibles, las que le interese en ese momento.

Se ha decidido crear dos widgets independientes que permiten definir filtros dependiendo de su complejidad: el widget de filtro simple y el de filtro complejo.

Antes de nada, conviene resaltar, la definición de “complejidad” a la que nos referimos cuando se menciona dicho término, pues en este caso no hace referencia a la cantidad de parámetros que contiene la consulta, ya que estos pueden ser tantos como el usuario quiera, sino al número de operaciones que se han de realizar en la formación de la consulta. (Entendiéndose operación como una operación binaria “and”, “or”, etc que nos permite enlazar entre sí, las distintas variables a tratar dentro de la consulta)

El filtro simple sólo podrá tener un tipo de operación binaria, mientras que el complejo puede contener diferentes tipos de operaciones de forma anidada.

Con esta aproximación se simplifica la interfaz de usuario para la operación más frecuente (consultas simples), pero se permite un modo “avanzado” para definir consultas todo lo complejas que sea necesario.

4.1 Requisitos

4.1.1 Filtros simples

Implementar un widget que permitan realizar, de manera amigable al usuario, filtros sobre los diferentes tipos de datos disponibles en la aplicación (texto, numérico, fecha, discreto, polígono y coordenada geográfica). Todos ellos deben tener unas características comunes:

- Posibilidad de contener un valor nulo
- Posibilidad de realizar un filtro exclusivo
- La tabla y la columna (variable) sobre la que se realiza el filtro, se podrán seleccionar desde la ventana mediante listas desplegables.
- Disponer de ayudas de validación de la selección, ya que la introducción de información errónea por parte del usuario es inevitable.
- Todos los elementos visuales que sean necesarios generar aparecerán en una ventana modal que permitirá su cierre y redimensionado por parte del usuario.

Cuando el filtro tenga un conjunto conocido de estados, ya sea de variables discretas, coordenadas o polígonos, se mostrará el conjunto de estados a su vez en una lista desplegable.

En el widget de filtro simple, además de permitir definir un filtro con una condición sobre una variable, existirá la opción de filtro comparativo de variables. Al igual que el caso anterior se desplegará sobre una ventana modal, aunque ahora el filtro se compondrá de dos selectores desplegables de variables (cada uno con su tabla-columna), así como de otro selector que permita la elección del tipo de operación de comparación a realizar (mayor que, igual que, menor que, etc).

4.1.2 Filtros complejos

La aplicación tendrá un modo de filtrado avanzado que se compondrá de un conjunto de filtros simples (ver apartado anterior), que se podrán anidar mediante operadores AND u OR con ilimitados niveles. Para componer los filtros complejos se hará uso de una interfaz con soporte de "arrastrar y soltar".

El menú estará formado por 5 capas:

- La 1º capa tendrá 2 botones para la formación de los dos tipos de filtros simples: "simple" o "comparación".
- La 2º capa contendrá en forma de etiquetas seleccionables las diferentes operaciones binarias que unirán varios filtros simples en uno complejo.
- La 3º capa, cumplirá la función de contener los filtros simples creados de forma temporal, a la espera de su posterior utilización en el filtro complejo.
- La 4º capa contendrá una previsualización del filtro complejo formado hasta el momento.
- La 5º capa tendrá la función de una papelera y servirá para eliminar cualquier elemento arrastrable no deseado.

Para la creación de las diferentes condiciones o filtrados simples sobre variables que conformarán el filtro complejo, se seguirán los requisitos expresados con anterioridad en el apartado "requisitos de filtros simples", pero con la peculiaridad que en cuanto se creen, serán añadidos automáticamente a la capa dispuesta para tal fin.

4.2 Desarrollo del widget de filtrado simple

Ahora que los requisitos están claros, comencemos con la explicación de cómo se han llevado a cabo.

Debido a la dependencia que existe entre los filtros simples y todos los demás componentes vamos a empezar por aquí, partiendo desde el principio, mostrando cómo está realizado cada elemento, qué función cumple, y su relación con los demás.

4.2.1 Selector de columna de la BBDD a filtrar

Una de las partes más importantes en el filtrado, como es obvio, es la elección de cada una de las condiciones que conformarán dicho filtro. Y a su vez la parte fundamental en la creación de las condiciones, es la selección de la variable sobre la que actuar.

Este será nuestro punto de partida.

El primer paso a realizar es la elección de la tabla objetivo de entre las posibles, así como la variable a tratar por el filtro. Esta funcionalidad ha sido encapsulada en un widget cuya representación visual son dos listas desplegables emparentadas entre ellas, de tal forma que la segunda lista de opciones a mostrar dependa de lo seleccionado en la primera.

Vaadin por defecto ya trae implementada una versión base de una lista desplegable, y sobre ella se han realizado las mejoras para llegar al fin descrito.

La primera modificación ha consistido en crear una nueva lista desplegable que admitiera genéricos y tratara internamente el tipo pasado, pudiendo devolver los objetos seleccionados con el tipo concreto. Seguidamente, se ha automatizado la creación de estas listas, pues en nuestro caso, queríamos que muchas de las opciones ya estuvieran asignadas, tales como la no posibilidad de agregar nuevos elementos por parte del usuario más allá de los mostrados o que las selecciones se actualizarán en el acto.

```
1  public static <T> TypedComboBox<T> getTypedComboBox(String caption,
2  List<T> values, Function<T, String> func) {
3     TypedComboBox<T> typedComboBox = new TypedComboBox<T>(caption);
4     typedComboBox.setAvailableElements(values);
5     for(T t : values)
6           typedComboBox.setItemCaption(t, func.apply(t));
7     typedComboBox.setImmediate(true);
8     typedComboBox.setInvalidAllowed(false);
9     typedComboBox.setNullSelectionAllowed(false);
10    typedComboBox.setNewItemsAllowed(false);
11    return typedComboBox;
12 }
```

En el constructor de este método, se puede apreciar el paso de una función como argumento. Esto es debido a que las listas desplegables por defecto usan como representación del valor de cada objeto el método “toString” y es necesario definir para cada objeto el valor que queremos mostrar.

Sobre esta clase de partida se ha implementado la lista desplegable dependiente.

Como recibe la lista desplegable padre de la que depende, la lista hija puede suscribirse al evento de “valor cambiado” del padre y generar su contenido en función de la selección.

```
1  public class DependantTypedComboBox<T, P> extends TypedComboBox<T> {
2
3      public DependantTypedComboBox(String caption, Function<T, String>
4      itemCaptionFunction, TypedComboBox<P> parent, Function<P, Collection<T>>
5      childElementsFactory) {
6              super(caption, itemCaptionFunction);
7              this.parent = parent;
8              this.childElementsFactory = childElementsFactory;
9              parent.addValueChangeListener(new   Property.ValueChangeListener()
10 {
11                     @Override
12                     public void valueChange(Property.ValueChangeEvent event) {
13                             update();
14                     }
15             });
16     }
17
18     private final TypedComboBox<P> parent;
19     private final transient Function<P, Collection<T>> childElementsFactory;
20
21     public void update(){
22             P element = parent.getSelectedElement();
23             if(element != null){
24                     setAvailableElements(childElementsFactory.apply(element));
25             }else{
26                     setAvailableElements(nullParentElements());
27             }
28     }
29
30     protected Collection<T> nullParentElements(){
31             return Collections.emptyList();
32     }
33 }
```

Por último, el widget propiamente en sí, hace uso de estas clases explicadas hasta ahora, para generar ambas listas desplegables con todas las características ya establecidas y listo para funcionar.

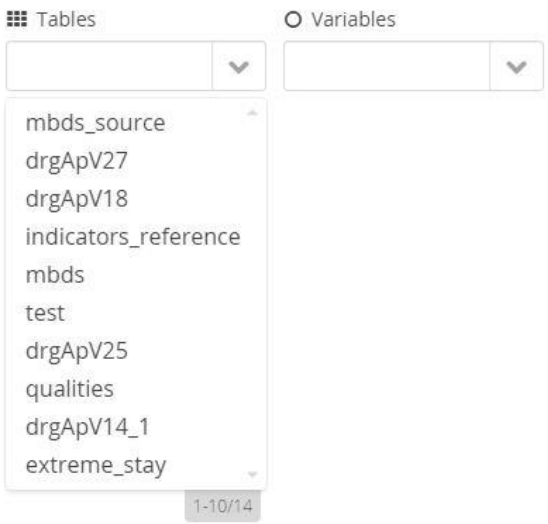

Ilustración 4.1 Widget de selección compuesto por listas desplegables dependientes

4.2.2 Condiciones de filtrado según el tipo de columna

En un principio, esta aplicación es capaz de tratar 6 tipos diferentes de variables, intentando abarcar con ellas todas las posibilidades, siendo estos tipos los siguientes:

Tipo texto, tipo rango numérico, tipo fecha, tipo discreto, tipo coordenada y tipo polígono.

Como en un futuro puede ser probable que se amplíe esta lista, para permitir que el sistema admita la ampliación con el menor número de cambios posible, se ha decidido implementar el patrón visitante para la formación de las condiciones.

De primeras debemos tener en cuenta que cada variable tiene un tipo asociado a ella, y por tanto, tendrán que ser mostrados los componentes que mejor la definan en cada caso, siendo aquí donde entra en juego el patrón visitante.

La forma de proceder es la siguiente:

El usuario selecciona una tabla y una variable del widget de las listas desplegables, a continuación se genera un evento que llama al visitante con la variable en cuestión y éste devuelve el componente asociado al tipo de variable recibida como argumento.

Consiguiendo de este modo que las variables y su tipado sean tratadas de forma transparente al usuario.

```
1  public AbstractVariableFilterComponent createVariableFilterComponent(final
       Column column, final LayoutWithButton layoutWithButton){
2      final CreateFilterLayout createFilterLayout = this;
3      VariableVisitor<AbstractVariableFilterComponent> visitor =
        new VariableVisitor<AbstractVariableFilterComponent>() {
4            @Override
5            public AbstractVariableFilterComponent visit(Variable variable) {
6                  return new TextVariableFilterComponent(createFilterLayout,
                    column, layoutWithButton);
7            }
8            @Override
9            public AbstractVariableFilterComponent visit(
             DateVariable variable) {
10                 return new DateVariableFilterComponent(createFilterLayout,
                    column, layoutWithButton);
11           }
12           @Override
13           public AbstractVariableFilterComponent visit(
             DiscreteVariable variable) {
14                 return new DiscreteVariableFilterComponent(
                   createFilterLayout, column, layoutWithButton);
15           }
16           @Override
17           public AbstractVariableFilterComponent visit(
             NumberVariable variable) {
18                 return new ContinuousVariableFilterComponent(
                   createFilterLayout, column, layoutWithButton);
19           }
20           @Override
21           public AbstractVariableFilterComponent visit(
             MultipleVariable variable) {
22                 return new TextVariableFilterComponent(
23                 createFilterLayout, column, layoutWithButton);
24           }
25           @Override
26           public AbstractVariableFilterComponent visit(
             LocationVariable arg0) {
27                 return          new          LocationVariableFilterComponent(
                   createFilterLayout, column, layoutWithButton);
29           }
30           @Override
31           public AbstractVariableFilterComponent visit(
             PolygonVariable arg0) {
32                 return          new          PolygonVariableFilterComponent(
                   createFilterLayout, column, layoutWithButton);
33           }
34     };
35     AbstractVariableFilterComponent vfc = column.getVariable().
       accept(visitor);
37     vfc.addLayoutComponents();
```

```
38      return vfc;
39  }
```

En el código superior expuesto, intervienen ciertas clases que todavía no se han explicado, pues conviene ir entendiendo el funcionamiento de la aplicación paso a paso y una vez que los conceptos hayan sido mencionados, se tratará dicho código con mayor detenimiento.

El resultado de la llamada al visitante produce entre otros, los siguientes menús:

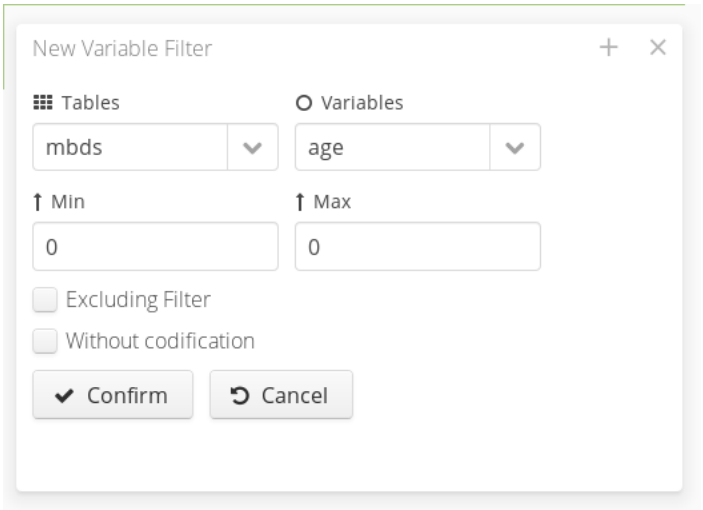

Ilustración 4.2 Variable de filtrado de tipo numérico

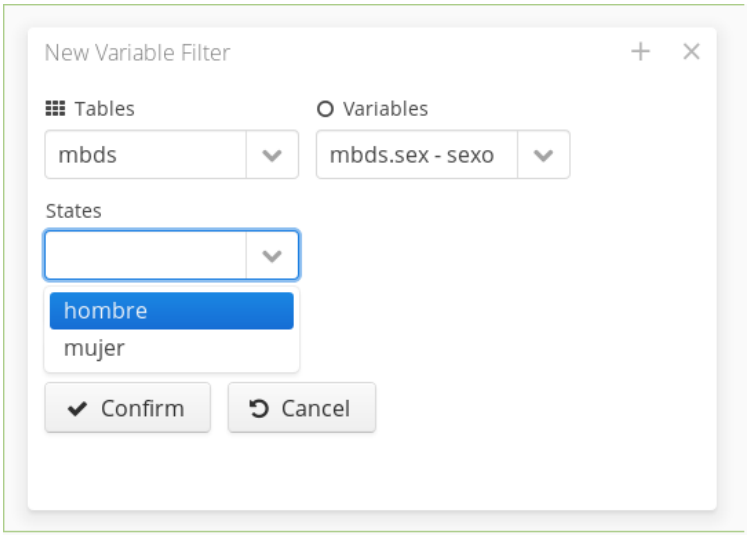

Ilustración 4.3 Variable de filtrado de tipo discreta

Permite el filtrado de la variable mediante la selección de los estados deseados gracias a una lista desplegable que contiene todos los posibles. Estos estados una vez son seleccionados, se muestran en forma de tokens sin repetición con la posibilidad de borrado en caso de equivocación.

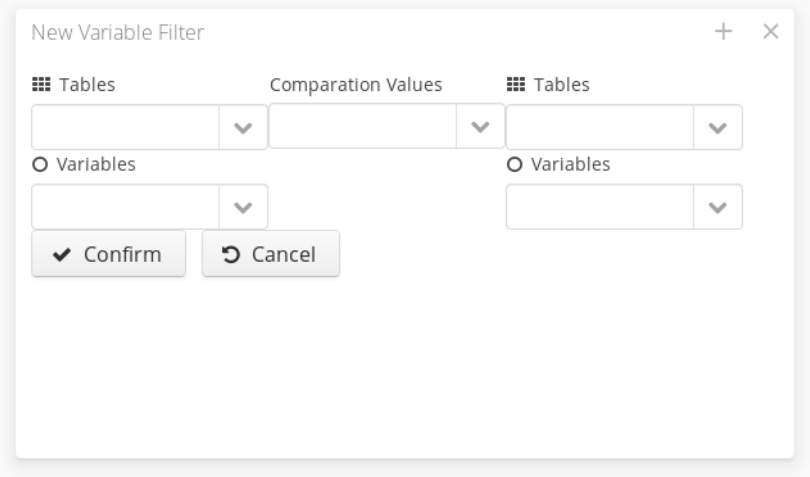

Ilustración 4.4 Generación de condición de tipo comparativa

4.2.3 Representación gráfica del filtro

Una vez terminada la condición de filtrado, ya sea simple o comparativa, es necesario crear una representación en forma gráfica equivalente a la información introducida.

Dicha representación de filtro dependiente del tipo de variable ha sido dispuesta sobre una capa contenedora (Layout de Vaadin), que tiene además en su interior, una etiqueta para mostrar la información y uno o dos botones según las operaciones que se puedan realizar (editar y borrar).

Para que la representación fuera en todo momento coherente con el modelo que hay por detrás, se optó por realizar un conversor. Este conversor es el encargado de transformar el modelo a una etiqueta equivalente visualmente, puesto que la operación contraria no es necesaria, ya que siempre hay una referencia al objeto de negocio al que apunta, mediante un enlace de datos o databinding.

Una vez llegado a este punto es posible explicar los elementos que forman parte del código del patrón visitante.

Como argumentos recibe un "Column" y un "LayoutWithButton". El primero hace referencia a la variable elegida del widget de las listas desplegables, y el segundo a la representación visual que tendrá esta condición, explicada en el párrafo anterior.

Dentro del método se crea una variable final de tipo "CreateFilterLayout".

Esta variable hace referencia a la clase encargada de generar las distintas capas y ventanas modales sobre las que se introducirán y añadirán los elementos posteriormente creados.

En resumen, el proceso es el siguiente: primero se crea el envoltorio sobre el que irán los controles para crear una condición de filtrado, que en nuestro caso, es una ventana modal dispuesta con una serie de capas vacías a la espera de los controles. Al mismo tiempo se crea un componente con una etiqueta asociada para representar la futura condición de filtrado.

Sobre esta ventana modal se añade el resultado del visitante, dando como resultado, los controles específicos de la variable seleccionada.

Posteriormente, una vez aceptada la condición, el objeto de modelo del visitante se enlazará con la etiqueta y se generará su representación visual.

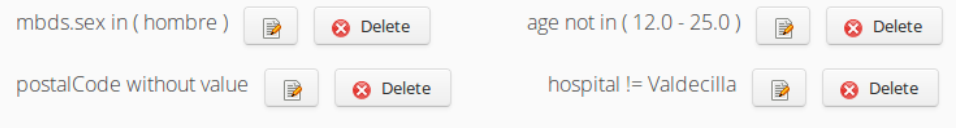

Ilustración 4.5 Representación de las condiciones de filtrado

El resultado final del widget de filtrado simple, con todo lo mostrado hasta el momento tiene el aspecto de la ilustración 2.6

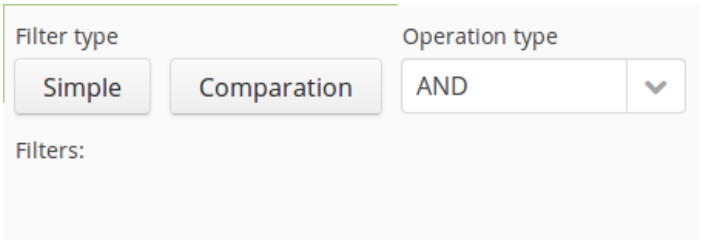

Ilustración 4.6 Widget para la generación de filtros simples

4.2.4 Generación de la condición de consulta SQL a partir del filtro

Una vez aceptado el filtro, se reunirán todos los datos introducidos y procesados hasta el momento, generándose una traducción equivalente a dicha consulta en el motor de BBDD asociado. Para ello, se emplea un servicio creado por la empresa Predictia que se encarga crear la consulta SQL, gracias al cual no ha sido necesario tener que programar a bajo nivel en lo que a creación de consultas se refiere. Dicho servicio toma como argumento de entrada una instancia de Filter, que puede ser de tipo AndFilter, OrFilter o ComplexFilter, en último caso tendría a su vez dentro otros filtros. Debido a esta naturaleza recursiva soporta filtros arbitrariamente complejos. Las instancias de filtros AndFilter y OrFilter se componen a su vez de lista de AbstractVariableFilter, clase abstracta que tiene implementaciones concretas asociadas a cada uno de los diferentes tipos de Variables. De esta manera todos los elementos de la interfaz de usuario tienen su objeto de modelo correspondiente y son soportados como entrada a este servicio, que internamente generará un SQL válido para el motor de BBDD concreto que se esté empleando.

4.3 Desarrollo del widget de filtrado complejo

Siguiendo con las especificaciones propuestas, este widget visualmente hablando ha sido dividido en 5 secciones para que cumpla totalmente con los requisitos.

Como la parte fundamental o primigenia tiene su base en los filtros simples, toda la nueva estructura y por tanto la jerarquía de clases desarrollada, se ha expandido en torno a todo lo implementado con anterioridad, permitiendo de este modo centrarnos en el nuevo componente y su funcionalidad sin necesidad de echar la vista atrás.

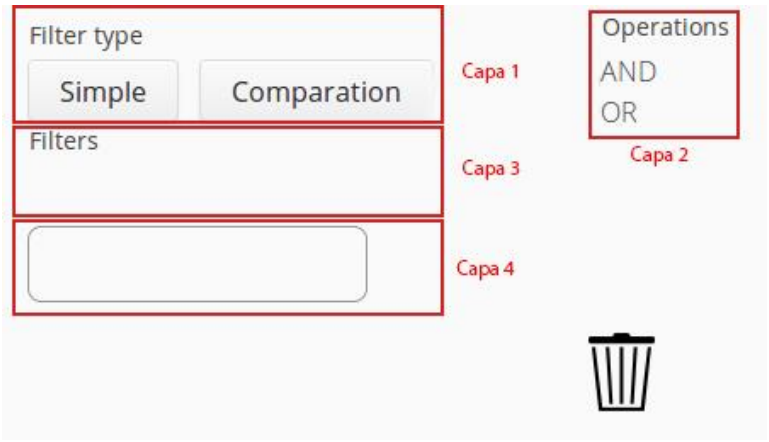

Ilustración 4.7 División en capas del widget de filtrso complejos

Tal y como se puede apreciar en la ilustración 4.7, la capa 1 ha sido reaprovechada completamente.

Sin embargo las capas 2 y 3 han tenido que ser modificadas respecto a la versión anterior, añadiendo unas variaciones en su comportamiento que permitan ser usadas en ambos casos.

Estas variaciones son:

En la capa 2, se ha sustituido al selector desplegable de operaciones por un par de etiquetas representando a dichas operaciones, las cuales, además pueden ser arrastradas/soltadas para formar el filtro según se quiera.

En la capa 3, siguiendo este criterio, todas las etiquetas generadas también podrán ser movidas mediante arrastrar y soltar, y a su vez, dejarán de incluir el botón de eliminar, puesto que esta operación ahora se traslada a un nuevo módulo representado por el icono de un cubo de basura.

4.3.1 Capa de previsualización de filtro

Esta capa (capa 4 de la ilustración 4.7) podríamos definirla como la encargada de todo lo concerniente en cuanto a la formación de filtros complejos se refiere.

Visualmente hablando, es un contenedor sobre el que podrán ser soltadas en su interior las diversas operaciones o condiciones de filtrado que darán lugar al filtro como tal.

Por tanto a medida que el usuario vaya rellenando el filtro, esta capa servirá para previsualizar el estado de la consulta hasta su finalización.

Cuando el usuario arrastre una etiqueta de operación a su interior, ésta automáticamente se transformará en un rectángulo capaz de contener dentro de ella, tanto condiciones de filtrado como a su vez otras operaciones, pudiendo llegar con este procedimiento a la complejidad que el filtro requiera.

4.3.2 Anidamiento de filtros

El encadenamiento de operaciones se puede realizar de dos maneras: en serie uniendo unas detrás de otras o enlazándolas unas dentro de otras de forma recursiva.

El propósito de estos encadenamientos es mantener cierto orden lógico, o lo que es lo mismo, una jerarquía por niveles.

Poniendo un ejemplo más fácil de entender del porqué de esta jerarquía, podríamos verlo como el uso de paréntesis en una operación matemática, en donde dependiendo de la precedencia o no de paréntesis, deberíamos ejecutar primero unas operaciones u otras.

Trasladando este símil a nuestro caso, nuestras operaciones binarias serían los paréntesis y las condiciones de filtrado las operaciones matemáticas.

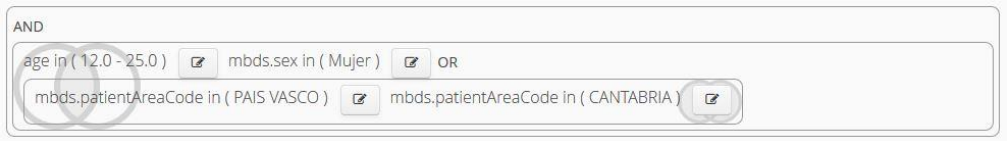

Ilustración 4.8 Ejemplo de consulta compleja

Haciendo la equivalencia de paréntesis sobre un filtro real, partimos de la consulta mostrada en la ilustración 4.8 y su traducción en una consulta sería:

age in (12,25)
AND
sex = 'Mujer'
AND
(*patientAreaCode* = 'CANTABRIA' **OR** *patientAreaCode* = 'PAIS_VASCO')

4.3.3 Implementación

Entrando un poco en los detalles de la implementación a nivel interna, vamos a resaltar aquellos componentes cuya funcionalidad sea crítica para entender el proceso al completo.

Como widget principal tenemos la capa contenedora. Dicha capa para poder mantener este orden lógico por niveles, posee un par de atributos fundamentales: una lista de condiciones de filtrado y otra lista de capas contenedoras del mismo tipo que el padre, permitiendo de este modo la recursividad entre capas.

Al tratarse de un widget contenedor de elementos visuales, aparte de poder crear/actualizar/eliminar dichos elementos, debe ser capaz de manejar y convertir entre la representación visual de los elementos introducidos y el objeto de modelo asociado a ellos que contendrá los datos en sí.

Para poder convertir un filtro completo a su representación equivalente, aparte de saber la jerarquía de operaciones presentes en él, es necesario transformar cada una de las condiciones de filtrado contenidas por cada tipo de operación. Por suerte este proceso ya fue resuelto tiempo atrás con los filtros simples

```
1 private LayoutWithButton convertAbstractFilterToLayoutWithButton
      (AbstractVariableFilter filter) {
2     ObjectProperty<AbstractVariableFilter> property =
      new ObjectProperty<AbstractVariableFilter>(filter);
3     AbstractVariableFilterConverter avfc =
      new AbstractVariableFilterConverter(cfl.getMessageSource());
4     Label lb = new Label();
5     lb.setConverter(avfc);
6     lb.setPropertyDataSource(property);
7     return new LayoutWithButton(cfl,lb);
8 }
```

Esta función recibe como argumento una instancia de cualquier tipo de filtro disponible, (por ello se usa la clase base abstracta, para que no haya limitación alguna) a continuación genera un enlace de datos sobre dicho tipo (para mantener actualizada en todo momento la representación) y por último asigna este enlace de datos junto con un conversor de objeto de modelo a texto, a la etiqueta especificada.

4.3.4 Ejemplo de filtro complejo

La mejor forma para comprender el funcionamiento de la formación al completo, es mediante un ejemplo.

Partimos del caso que deseamos buscar a los pacientes varones del hospital de Valdecilla cuya edad esté comprendida entre los 12 y 25 años o entre los 40 y 50.

El primer paso sería poner una operación binaria como base de la consulta.

(Siempre aunque la consulta sólo requiera de una condición, es necesaria una operación que la envuelva por funcionamiento del sistema. Además una operación binaria sobre un solo operando no influye en el resultado)

En el momento que terminemos de arrastrar la operación y soltemos el elemento en el recuadro de filtrado, comprobaremos cómo dicha etiqueta se transforma en un rectángulo, permitiéndonos colocar más elementos en su interior.

Esta transformación sucede gracias al manejador asociado al evento de arrastrar y soltar, el cual, está enlazado tanto con las operaciones, como con la representación visual de las condiciones de filtrado.

Dicho manejador dependiendo del tipo de la instancia del objeto que ha generado el evento, podrá distinguir entre uno u otro elemento y realizar las operaciones oportunas.

Ahora, con la operación colocada, tendríamos que definir las diversas condiciones de filtrado tal y como se ha explicado en el apartado anterior de filtros simples, para posteriormente arrastrar e introducir una a una en el recuadro asociado a la operación.

Como en nuestro ejemplo tenemos una operación disyuntiva en la consulta, tendríamos que colocar sobre la operación base AND una nueva operación. Esta vez de tipo OR y dentro de ella, los filtros de rango de edad.

Si por casualidad, alguno de los elementos dentro de la consulta, estuviera mal colocado, todos ellos pueden ser movidos o recolocados en cualquier momento hasta conseguir la disposición requerida. También a su vez, cualquier parte de la estructura que se ve dentro de la consulta puede ser borrada, ya sea en su totalidad o solo una parte. Si el elemento a borrar es una operación binaria, se borrará tanto la propia operación como todos los elementos que de ella dependan en la jerarquía.

Para borrarlo, tan sólo hay que arrastrar el elemento hasta el icono representado por una papelera y automáticamente el manejador asociado borrará toda estructura relacionada con dicho elemento.

4.3.5 Generación de condición SQL a partir del filtro

Como ya tenemos formulada nuestra consulta, el siguiente paso lógico sería resolverla. Para ello es necesario recopilar toda la información introducida por el usuario hasta el momento, clasificarla, asignarla a un tipo de filtro concreto y posteriormente llamar al servicio de Predictia encargado de generar el código SQL con dichos filtros como argumento.

Tal y como ya se ha comentado previamente, los tipos de filtros que maneja dicho servicio pueden ser de tres tipos. Por un lado, los de tipo AndFilter y OrFilter y por otro los de tipo ComplexFilter que a su vez pueden estar formados por cualquiera de estos tres tipos.

Es por ello, que nuestra misión consiste en convertir esta representación visual de la consulta a estos tipos de filtros, según convenga.

Gracias a la ayuda de un componente auxiliar con toda la lógica de conversión entre la vista y el modelo de negocio, este proceso es más fácil de explicar. Comencemos:

El primer paso sería conocer el alcance del filtro y sus componentes. El problema surge debido a que al tener estos filtros complejos una naturaleza recursiva, no se puede saber su estructura de forma inmediata y es necesario recorrerlo al completo. En este punto se pueden dar dos opciones:

Que el filtro tenga una estructura idéntica al filtro simple, es decir, que solo contenga una operación y diversas condiciones en su interior. En cuyo caso su resolución es directa, pues una vez detectado este hecho, se resuelve del mismo modo que si fuera un filtro simple.

Se recorre cada una de las condiciones de filtrado incluidas en la operación asignada, obteniendo la información asociada a cada variable mediante el enlace de datos o databinding establecido con la etiqueta.

Este proceso ha sido encapsulado en un método, al cual se le pasa como argumento la capa sobre la que se encuentra la operación, permitiendo que pueda ser usado en cualquier nivel de la jerarquía del filtro complejo.

```
1 public static Filter getSimpleFilter(QueryLayout root) {
2     Filter filter = Filters.emptyAndFilter();
3     Set<LayoutWithButton> LayoutWithButtons = root.getLayoutWithButtons();
4           if(!LayoutWithButtons.isEmpty()) {
5                 ArrayList<AbstractVariableFilter> array =
                  new ArrayList<AbstractVariableFilter>();
6                 for(LayoutWithButton l: LayoutWithButtons) {
7                       array.add((AbstractVariableFilter)l.getLabel().
                        getPropertyDataSource().getValue());
8                       filter = getTypeOfSimpleFilter(root, array);
9                 }
10    return filter;
11 }
```

```
1 public static Filter getTypeOfSimpleFilter(AbstractLayout root,
            ArrayList<AbstractVariableFilter> array) {
2     Filter filter = Filters.emptyAndFilter();
3     String andCaption = FilterOperation.AND.toString();
4     String caption = root.getCaption();
5     if(andCaption.equals(caption)) {
6           filter = new AndFilter(array);
7     } else {
8           filter = new OrFilter(array);
9     }
10    return filter;
11 }
```

La segunda opción se produce cuando el filtro está compuesto por diversas operaciones anidadas. Pero en el fondo no dista mucho de ser una lista de filtros simples unidos entre sí.

En definitiva, todo el proceso de resolución recae sobre el siguiente método:

```
1 public static Filter getFilter(QueryLayout root) {
2 Filter filter = Filters.emptyAndFilter();
3     Set<QueryLayout> layouts = root.getQueryLayouts();
4     if(layouts.isEmpty()) {
5           filter = getSimpleFilter(root);
6     } else {
7           Set<Filter> filters = new HashSet<Filter>();
8           Filter aux = Filters.emptyAndFilter();
9           for(QueryLayout l: layouts) {
10                if(!isEmpty((aux = getFilter(l))))
11                      filters.add(aux);
12          }
13          filter = getSimpleFilter(root);
14          if(!isEmpty(filter)) {
15                filters.add(filter);
16                filter = new ComplexFilter(filter.getFilterOperation(),
                        filters.toArray(new Filter[]{}));
17          } else {
18                filter = new ComplexFilter(aux.getFilterOperation(),
                         filters.toArray(new Filter[]{}));
19          }
20    }
21    return filter;
22 }
```

Como todo método recursivo tiene el caso base o directo, siendo éste la resolución del filtro al nivel más básico, es decir, un filtrado con una sola operación, y el caso recursivo, el cual va avanzado operación tras operación dentro de la jerarquía por niveles, resolviendo cada uno de ellos y uniéndolos entre sí, para finalmente devolver el total ya completado.

4.4 Traspaso de información entre widgets de filtrado

Tal y como ya se ha comentado, ambos widgets basan su funcionamiento en el mismo principio, que consiste en utilizar una operación como base y rellenarla con condiciones aplicadas a diversas variables.

Es por ello que desde el principio, la aplicación se ha implementado centrando su diseño en torno al desarrollo y resolución de filtros simples, intentando en la medida de lo posible que todos los widget creados sin importar su extensión, pudieran ser reutilizados posteriormente o que fueran fácilmente adaptables y extensibles.

Esta ardua labor dio como resultado la compatibilidad en el traspaso de información entre los widget de filtrado simple y complejo.

Poniéndonos en la piel del usuario final, no es de extrañar que a la hora de realizar un filtro, utilice primeramente el widget de filtrado simple, pero según componga el filtro, se dé cuenta que el alcance que permite dicho widget no es el suficientemente grande y tenga que cambiar al siguiente.

Gracias a este traspaso de información, dicho paso es automático y totalmente transparente al usuario, aunque por contra tiene una limitación por cómo se han definido lógicamente los filtros, y es el siguiente:

Un filtro complejo internamente está compuesto por estructuras de tipo ComplexFilter, que pueden albergar en su interior cualquier tipo de filtro, sin embargo, el filtro simple no dispone de tal capacidad y aquí está el problema en cuestión.

Si los datos que se reciben para la construcción de un filtro complejo provienen de un filtro simple, la construcción es inmediata, pero si sucede al revés, es decir, se pretende construir un filtro simple a partir de uno complejo, tenemos un serio problema.

La mejor solución que se pudo dar a este problema fue la siguiente: Se lee el filtro de entrada, si por casualidad todas las estructuras internas son completamente compatibles se transforma sin problema, si por el contrario no lo son, se avisa al usuario de una posible pérdida de información y se procede a la codificación del filtro, buscando internamente qué parte del filtro complejo es la más interesante de salvar.

Esta capacidad de recibir una instancia de tipo Filter y a partir de ella generar los elementos visuales necesarios para su interpretación, también abre las posibilidades en un futuro para que ambos widgets de filtrado puedan ser usados no solo para la creación de filtros, sino para la visualización y manejo de filtros generados externamente.

4.5 Pruebas realizadas

Al tratarse de un conjunto de widgets en donde la interfaz visual de los mismos juega un papel bastante importante, la pruebas se han centrado primordialmente en comprobar el funcionamiento correcto de todos los elementos en los que el usuario puede interaccionar, ya sea tanto introduciendo o modificando datos, como navegando por las diferentes pantallas y menús.

Prueba 1.

Descripción:

Generar el filtro de sexo = hombre y comunidad autónoma = Cantabria

Resultado esperado:

Un par de etiquetas que muestren el nombre de la columna junto con el valor de filtrado seleccionado.

Forma de verificar:

Acceder a la pestaña de filtro simple y seleccionar en el selector de operación el tipo AND y pulsar sobre el botón de condición de filtro simple, seleccionado la columna de sexo y escogiendo el valor de hombre. Repetir este paso seleccionando como columna la comunidad autónoma y el valor Cantabria.

Prueba 2.

Descripción:

Generar el filtro de sexo = hombre, edad entre 15 y 25 y la comunidad autónoma = Cantabria o País Vasco.

Resultado esperado:

Un rectángulo representando la operación base de tipo AND. Dentro de este rectángulo se encontrarán un par de etiquetas representando a las condiciones de filtrado de sexo y edad. Seguidamente aparecerá otro rectángulo representando la operación de tipo OR y en su interior se encontrará otro par de etiquetas mostrando las condiciones de filtrado de comunidad autónoma con los valores de Cantabria y País Vasco respectivamente.

Forma de verificar:

Acceder a la pestaña de filtro complejo y arrastrar en el selector de operación la etiqueta correspondiente al tipo AND hasta el espacio dispuesto para la formación de filtros.

Pulsar sobre el botón de condición de filtro simple, seleccionado la columna de sexo y escogiendo el valor de hombre. Volver a pulsar el botón y elegir la columna de edad e introducir los límites de 15 y 25 en sendos cuadros de texto. Pulsar el botón nuevamente dos veces más, seleccionando en ambos casos la columna de comunidad autónoma y como valor de la condición, elegir del desplegable, los elementos de Cantabria y País Vasco respectivamente.

A continuación arrastrar las etiquetas representando las condiciones de sexo y edad sobre el recuadro generado por la operación AND y seguidamente introducir una etiqueta de tipo OR en el interior de este mismo recuadro.

Por último arrastrar las etiquetas relativas a la comunidad autónoma en el interior del rectángulo generado por la operación OR.

Prueba 3.

Descripción:

Habiendo generado el filtro simple de la Prueba 1, traspasar la información introducida al widget de filtro complejo.

Resultado esperado:

En la capa de previsualización del filtro complejo aparece un rectángulo representando a la operación AND, y en su interior se encuentran un par de etiquetas representando a las condiciones de filtrado del sexo y comunidad autónoma.

Forma de verificar:

Teniendo el filtro simple ya construido, pulsar sobre la pestaña de filtros complejos.

Prueba 4.

Descripción:

Habiendo previamente generado el filtro complejo de la prueba 2, traspasar la información introducida al widget de filtro simple.

Resultado esperado:

En la lista desplegable de operaciones se auto selecciona la operación AND y la parte inferior a dicho desplegable, se generan dos etiquetas correspondientes a las condiciones de filtrado acerca del sexo y la edad.

Forma de verificar:

Teniendo el filtro complejo ya construido, pulsar sobre la pestaña de filtros simples.

Una posible fuente de errores que suele necesitar un conjunto de pruebas asociadas, es la entrada de datos por parte del usuario. En nuestro caso, estas pruebas no han sido necesarias puesto que se han definido un conjunto de validadores apoyados en el framework de validación de vaadin, el cual, nos permite conseguir un control sobre los campos de texto

Para hacer uso de estos validadores basta con enlazar al campo, una variable de tipo básico (string, entero, etc) y él por sí mismo impondrá las restricciones clásicas a dicho tipo, como puede ser la imposibilidad de escribir un texto en un campo numérico, etc.

Además estas validaciones pueden ser fácilmente ampliables con nuestras propias reglas o incluso aplicables a nuevos tipos, siempre que sean derivaciones de los básicos.

En nuestro caso todos los campos de texto tienen asociada una validación por detrás.

Las más simples corresponderían al control de rango numérico realizado entre dos campos y las más complejas por ejemplo, al control de fechas con un formato en concreto y que además cumplan ciertas restricciones (el día de ingreso de un paciente no puede ser anterior a su fecha de nacimiento, la fecha de alta debe ser el mismo día o superior que la de ingreso, etc).

Debido a que realizar todas las pruebas sobre estos campos de texto de forma exhaustiva supondría una gran cantidad de tiempo, se han limitado a solo probar los casos en los valores límite. De este modo clasificando las pruebas por tipo de datos tenemos:

Tipo texto:

- Se permite la introducción de cualquier carácter sin restricciones

Tipo numérico: (entero o decimal)

- Solo se admite como valores correctos aquellos relativos a la representación numérica, caracteres del 0-9 en el caso de ser un entero, incluida la "," si son decimales.
- Si el campo numérico a su vez depende de otro para mostrar un rango, comprobar que los intervalos estén bien formados, límite inferior igual o menor que el superior y viceversa.

Tipo fecha:

- En caso de no utilizarse el módulo de calendario dispuesto para tal fin y ser introducidas las fechas manualmente, dichas fechas tendrán que cumplir con el formato siguiente DD/MM/YYYY, en caso contrario se tomará como inválidas.
- Si el campo depende de otro para mostrar un rango se tendrá que comprobar que el rango esté bien formado por orden cronológico.

Tipo discreto: (conjunto de valores)

- Los valores seleccionados no podrán repetirse, siendo estos únicos en la selección realizada.

5 Gestión del metadato

5.1 Gestión del metadato a nivel de Dominio

Cuando usamos el término Dominio, nos estamos refiriendo a una base de datos con los datos que el cliente ha proporcionado. Así mismo, el término metadato hace referencia a la definición del propio dato en sí.

Podríamos definir a este gestor como un gestor de seguridad que dependiendo del rol del usuario de entrada, limita el acceso a ciertos elementos u oculta parte de la información.

En cuanto a términos de base de datos se refiere, la funcionalidad de este widget se asemeja a la acción de crear distintas vistas sobre las tablas/columnas de una base de datos dependiendo del rol de cada usuario.

Debido a su planteamiento, este gestor está enfocado al uso de forma interna a cargo de un administrador, que en la medida de lo posible deberá conocer la estructura de la BBDD y los distintos roles o usuarios que contendrá.

5.1.1 Implementación de asistente

Para facilitar el uso, este gestor basa su funcionamiento en un asistente, permitiendo de este modo que el administrador sea guiado por todas las opciones y configuraciones posibles de una manera secuencial. Asegurándonos de esta forma, que todos los pasos han sido ejecutados, evitando que por despiste, el administrador, no incluya ciertos parámetros o se salte pasos que en su juicio sean irrelevantes.

La funcionalidad base de este asistente se ha conseguido gracias a la inclusión de un add-on de Vaadin. Dicho añadido permite indicar el número de pasos totales que tendrá el asistente, así como los menús que serán mostrados en cada paso y las acciones a tomar en los controles de avance y retroceso.

Un ejemplo de la creación de un paso del asistente:

```
1 private WizardStep createColumnSelector(final DathumMessageSource
            messageSource) {
2     final DataDomainColumnComponent ddv =
      new DataDomainVariableStateDetailsComponent(domain,
      messageSource, repositoryHelper);
4     return new WizardStep() {
6           @Override
6           public boolean onBack() {
7                 return true;
8           }
```

```
 9            @Override
10            public boolean onAdvance() {
11                  DataDomain data = ddv.getUpdatedElement();
12                  domain.setStatesSubset(data.getStatesSubset());
13                  domain.setFilter(data.getFilter());
14                  return true;
15            }
16            @Override
17            public Component getContent() {
18                  return ddv;
19            }
20            @Override
21            public String getCaption() {
22                  String message =
                    messageSource.getMessage("wizard.3.step.caption",
                    "Third Step");
23                  return message;
24            }
25      };
26 }
```

El código superior pertenece al último de los pasos de este gestor, en el cual se tratan las columnas asociadas a una tabla perteneciente al dominio especificado con antelación.

Hay que resaltar el método "getContent" pues éste será el encargado de devolver el componente que manejará la interfaz mostrada por este paso del asistente. Es por ello que dicho componente deberá estar asociado a una vista para poder visualizarse.

5.1.2 Widget de CRUD

Debido a que dentro de muchos de los pasos del asistente se realiza operaciones de tratamiento de datos tales como: crear, leer, actualizar y borrar sobre los registros asociados, para evitar la repetición y simplificación del código perteneciente a estas operaciones, se ha utilizado un tipo de estructura muy empleada en computación para la gestión de la capa de persistencia denominada como "CRUD", siendo este término un acrónimo del inglés ***C**reate*, ***R**ead*, ***U**pdate y **D**elete*.

Estos cruds tienen normalmente una apariencia en forma de tabla, en donde cada fila representa un registro, pudiendo realizar sobre cada uno de ellos, las operaciones antes citadas.

Por suerte, la empresa Predictia ya disponía de una implementación genérica de esta estructura de tipo crud, explicándose a continuación su funcionamiento:

El punto central de estos cruds, es la forma en la que obtienen los datos de la fuente de información y los procesan. Estas operaciones no se realizan escribiendo la consulta directamente y mandándola al motor de BBDD, sino que gracias al módulo de Spring, Spring Data, todo este proceso de acceso y explotación de datos nos viene dado mediante un servicio que internamente hace uso de repositorios.

En estos repositorios están definidas las operaciones básicas a realizar, tales como obtener todos los registros o solo el especificado y guardar o eliminar uno en concreto, que en nuestro caso es justo lo que necesitamos para los cruds.

Poniendo un ejemplo en concreto de la clase que implementa el crud de dominios, sería el siguiente código:

```
1  public class DataDomainCrudComponent extends
           AbstractCRUDComponent<DataDomain>{
2      private final transient RepositoryHelper repositoryHelper;
3      private final transient DathumMessageSource messageSource;
4      public DataDomainCrudComponent(DathumMessageSource messageSource,
           RepositoryHelper repositoryHelper) {
5          super(messageSource);
6          this.messageSource=messageSource;
7          this.repositoryHelper=repositoryHelper;
8          createLayout();
9      }
10     @Override
11     protected void create(DataDomain arg0) {
12         repositoryHelper.save(arg0, DataDomain.class);
13     }
14     @Override
15     protected void update(DataDomain arg0) {
16         repositoryHelper.save(arg0, DataDomain.class);
17     }
18     @Override
19     protected void delete(DataDomain arg0) {
20         repositoryHelper.delete(arg0, DataDomain.class);
21     }
22     @Override
23     protected Iterable<DataDomain> elements() {
24         return repositoryHelper.findAll(DataDomain.class);
25     }
26     @Override
27     protected List<PropertyDescription<DataDomain, ?>> properties() {
28         List<PropertyDescription<DataDomain, ?>> props =
           new ArrayList<PropertyDescription<DataDomain, ?>>();
29         props.add(
           new BeanStringPropertyDescription<DataDomain>(getMessageSource().
           getMessage("button.data.domain.create.caption", "Name"), "name")
           );
30         return props;
31     }
32 }
```

En esta clase, el servicio del tipo “DathumMessageSource” nos proporciona los textos que mostrará la aplicación, siempre dependiendo de la cultura del sistema establecida.

También mediante el método “properties” podemos definir cuántas propiedades del registro serán mostradas en la tabla.

El resultado visual del crud de dominios es el mostrado en la ilustración 5.1.

En la parte superior antes de la tabla propiamente en sí, se encuentran las opciones de crear un registro nuevo o filtrar los disponibles por una de las propiedades visibles. Dentro de la tabla, en la 1ª columna se listan todos los registros concernientes a los dominios de datos y en la 2ª columna se encuentran las operaciones disponibles para el registro asociado.

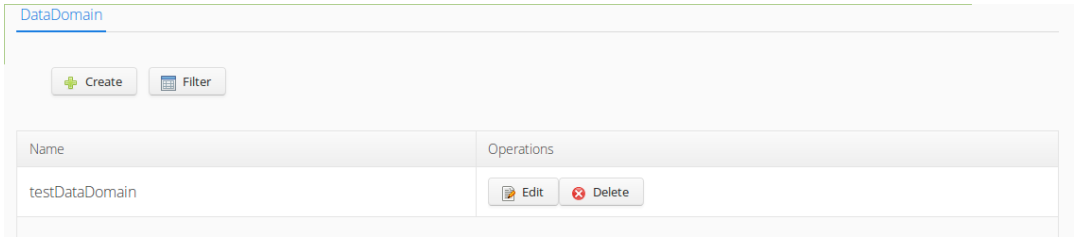

Ilustración 5.1 Interfaz visual de un crud listando los dominios disponibles

5.1.3 Funcionamiento del gestor

Una vez mencionados los principios en los que se basa este gestor, expliquemos los pasos a seguir con un ejemplo práctico.

El primer menú del que dispone este gestor nada más entrar, es un crud mostrándolos todos los dominios definidos hasta el momento.

Ya sea creando un nuevo dominio o usando uno previamente hecho, los pasos a seguir son siempre los mismos:

- El primero hace referencia a las opciones de configuración de la conexión de la BBDD a la que queremos conectarnos.

- El segundo, una vez comprobada y establecida la conexión con la BBDD, nos permite editar la estructura o visibilidad de las distintas partes
  .
- El tercero, se encarga de imponer filtros sobre las consultas que realicen los usuarios.

Si pulsamos el botón de editar, entraríamos al primer paso de los tres definidos dentro del asistente, mostrándonos todos los datos referentes al usuario, la conexión, etc.

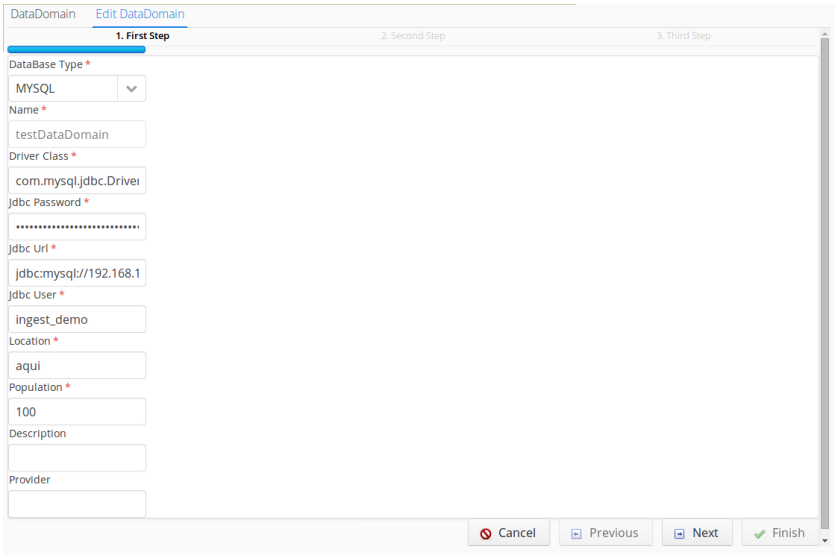

Ilustración 5.2 Paso primero del asistente de configuración de dominios.

El siguiente paso en el asistente (ilustración 5.3), consiste en la visualización y edición de todas y cada una de las tablas de las que está formada la base de datos. Aquí podemos elegir y editar cualquier tabla, o por el contrario borrar aquellas que no nos interese que dicho usuario pueda ver, es decir, seguirán estando definidas en la base de datos pero el usuario no tendrá visibilidad sobre ellas

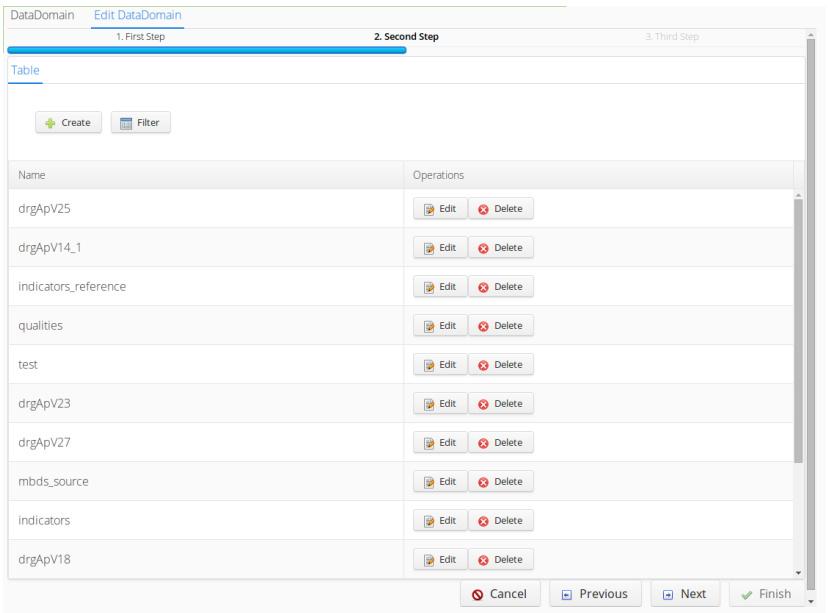

Ilustración 5.3 Paso segundo del asistente de dominios

Si decidimos editar una tabla cualquiera, se nos desplegará una nueva pestaña en cuyo interior aparecerá un nuevo crud pero está vez referente a las columnas que contiene la tabla seleccionada. Del mismo modo que en el caso anterior, las operaciones posibles son las mismas.

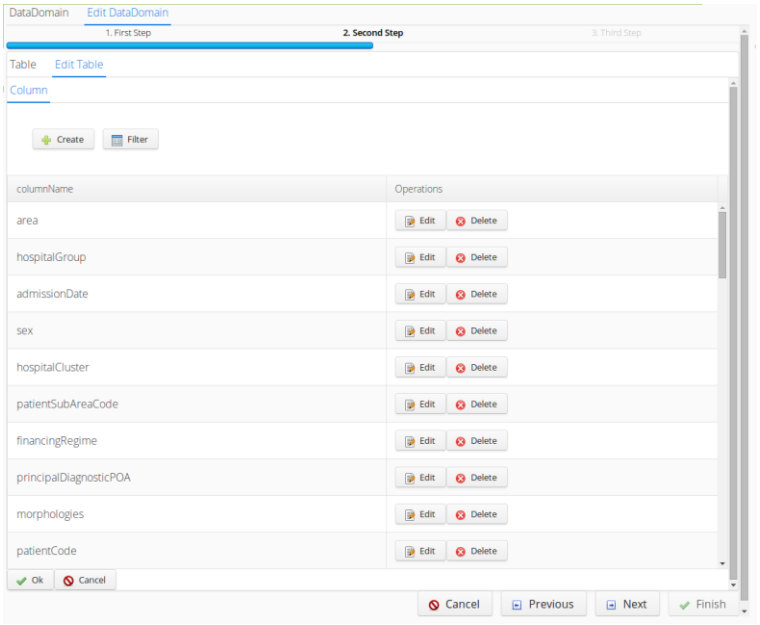

Ilustración 5.4 Paso tercero del asistente de dominios

Una vez ya nos encontremos dentro de la columna a editar, además de ver el nombre y tipo de datos que contiene la columna, podremos seleccionar de una lista desplegable, una de las variables ya establecidas a la que corresponde dicha columna.

Esto permite la reutilización del metadato definido, puesto que muchas de las columnas de diferentes dominios, corresponden con la misma variable virtual, permitiendo con ello reutilizar su definición.

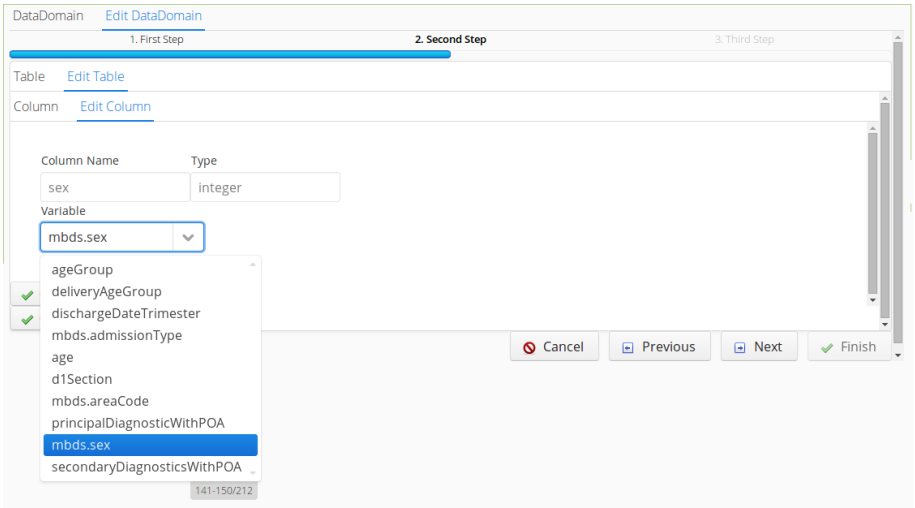

Ilustración 5.5 Correspondencia entre una columna de la BBDD y la variable definida previamente

Por último y para finalizar, quedaría el paso final del asistente, el cual consiste en definir un filtro sobre el que todas las consultas tendrán que ejecutarse, o los diferentes estados que se mostrarían ante una consulta positiva.

Internamente se ha hecho uso de los widgets de filtrado previamente generados.

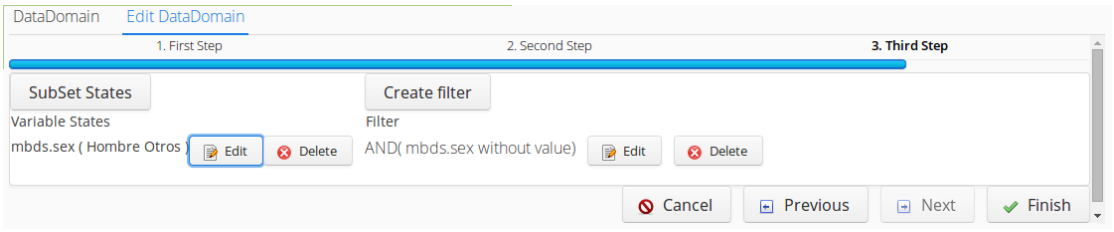

Ilustración 5.6 Último paso del asistente de configuración de dominios

5.2 Gestión del metadato a nivel de variable

Por defecto los tipos de las variables nada más volcados de la BBDD son tipos básicos, es decir, números o textos, y por tanto, no disponemos de mayor conocimiento sobre el que tratar dichas variables en sucesivas operaciones.

Por ello es necesario disponer de alguna forma, que permita cambiar este tipado por defecto, por uno que se asemeje lo máximo posible al dato a tratar, pues habrá casos en los que estos dos tipos básicos se queden cortos y sean imprecisos.

La implementación de este gestor, permite precisamente eso, modificar a nuestro criterio el metadato asociado a cada una de las variables mediante una serie de cruds y asistentes guiados.

Cada una de las variables sobre las que trabaja este módulo, en términos de base de datos, se podrían definir como una abstracción de una columna de una tabla.

5.2.1 Variable de tipo coordenada

Al igual que sucedía con el anterior gestor, el primer menú que vemos es un crud, aunque ahora en vez de dominios, se nos muestran todas las variables definidas previamente.

Una vez decidida la variable a tratar, entraremos en el asistente de configuración, cuyo primer paso es la selección del tipo de dato al que hará referencia la variable. Este punto es primordial, puesto que dependiendo de la opción elegida, se mostrará en cada caso, un asistente específico para dicho tipo con las opciones pertinentes.

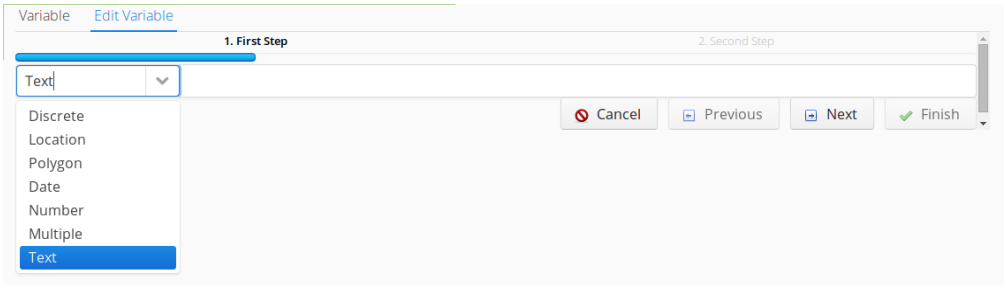

Ilustración 5.7 Listado con todos los tipos de definición de variables

En nuestro caso, vamos a tratar una variable con un componente geográfico de tipo coordenada, ya que es uno de los procesos más elaborados (junto con la asignación de polígonos) y por tanto dará una visión en perspectiva más acertada del trabajo realizado.

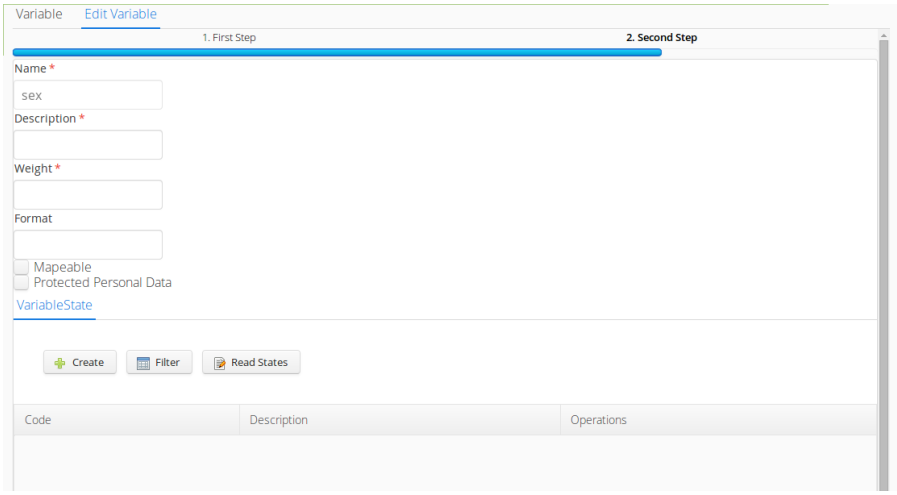

Ilustración 5.8 Menú de introducción de datos básicos de una variable

El siguiente paso, consiste en rellenar los datos que definirán a la variable en sí junto con los diferentes puntos geográficos de los que va a disponer esta variable. Esto se puede realizar de dos formas:

1- Creando punto por punto cada uno de ellos.
2- Leyéndolos mediante un fichero.

Siendo la opción 2, la más recomendable si se dispone de un archivo, pues generará los puntos de forma automática evitando posibles errores de transcripción humana al volcar los datos.

La lectura de ficheros se realiza mediante un asistente en tres pasos.

1. Elección del fichero a tratar.
   Para la carga del fichero se ha usado el widget predefinido de Vaadin dispuesto para esta acción.

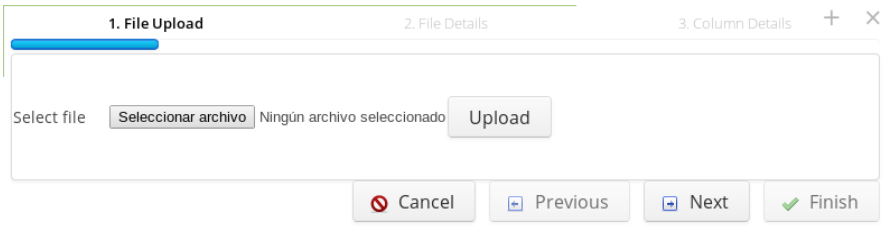

Ilustración 5.9 Primer paso del asistente de lectura de puntos geográficos

2. Selección del tipo de codificación en que está escrito el fichero, si este dispone de cabeceras y del tipo de delimitador por el que están separados los campos de cada dato.

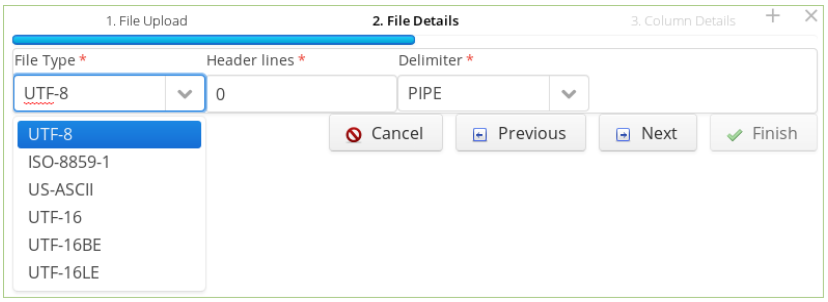

Ilustración 5.10 Segundo paso del asistente, mostrando las opciones disponibles

3. Por último, se nos muestra la primera línea leída del fichero para así tener una muestra y saber asignar a cada campo de la variable, el valor correcto del fichero.

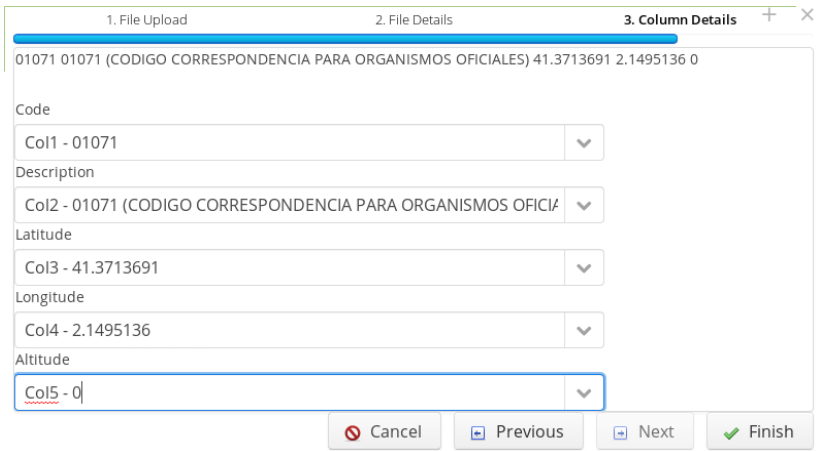

Ilustración 5.11 Asignación de cada columna procesada a una propiedad de la variable

Una vez leído el fichero, se nos mostrará de nuevo un crud con toda la información procesada relacionada con las coordenadas leídas, siendo posible editar o borrar aquella que no corresponda con nuestras necesidades.

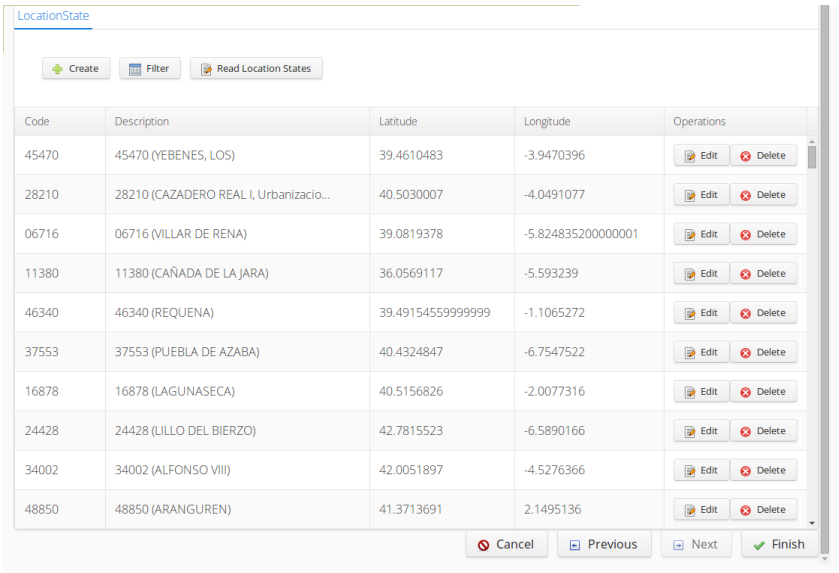

Ilustración 5.12 Crud con los datos geográficos procesados en el fichero

Ilustración 5.13 Menú de edición de un punto cualquiera de una variable. geográfica

5.2.2 Otros tipos de variables

Para los demás tipos de variables, el proceso es análogo, estableciendo eso sí, una serie de opciones acorde al tipo, es decir, si tratamos fechas tendremos que elegir la fecha de inicio, de fin, la duración, la escala, etc. Si por casualidad el tipo de dato es poligonal, el proceso de carga también permite hacerse mediante un fichero aunque tiene ciertas condiciones que deben cumplirse debido a la propia definición del formato del fichero que contiene los datos espaciales.

El fichero a subir deberá estar comprimido y contendrá en su interior como mínimo tres archivos con extensión .shp .shx .dbf

Cabe resaltar el caso de tratar tipos discretos, pues a la hora de definir los posibles estados, se nos permite leerlos de los ya volcados en el dominio de datos.

Aunque estos estados no tengan todos los campos correctos, permiten hacernos una idea de la cantidad total de los que se dispone, evitando así posibles omisiones. Además solo suele ser necesario retocar ciertos aspectos como puede ser la descripción, pero otros más importantes como el identificador si son válidos.

6 Visualización de los datos procesados

Cuando se realiza un análisis de datos, tan importante es el propio procesamiento de los datos, como su posterior visualización. De la representación que mejor defina y muestre los datos, dependerá en gran medida la facilidad para la comprensión de los resultados por parte del usuario.
Como todos los tipos tratados por esta aplicación no pueden ser representados de la misma forma y el abanico de posibilidades es bastante extenso, será necesario desarrollar diferentes módulos de visualización, como pueden ser diferentes tipos de gráficas, series temporales, o mapas. En el proyecto de fin de carrera se ha desarrollado un módulo de visualización capaz de representar aquellos tipos que tenían una componente geográfica asociada, habiendo permitido esto mismo su visualización mediante un mapa.

Dentro de la categoría de datos geográficos que maneja esta aplicación, podemos hacer una división en dos tipos:
Aquellos cuya representación se puede realizar mediante una coordenada geográfica o punto en el mapa y por otro lado, aquellos quienes necesiten representarse mediante un polígono o conjunto de polígonos cerrados debido a sus características.
Los mapas usados, al ser una representación bidimensional del terreno no tienen la capacidad de mostrar de forma clara más de dos variables simultáneamente mediante un marcador y es por ello que el máximo de dimensiones a mostrar se ha limitado a dos.
También hay que aclarar que no cualquier pareja de variables puede ser usada en el análisis al mismo tiempo, sino que solo podrá existir una con un componente geográfico, acompañada si se desea de otra variable, pero esta vez de tipo discreto.
Se ha optado por esta decisión por el siguiente motivo. La variable geográfica nos indica la posición en el mapa del punto o área a mostrar, mientras que la variable discreta, la cual posee datos con los diferentes estados, será la que muestre la información relativa a este punto en el mapa.

Respecto al tratamiento sobre todo lo relacionado con los mapas, es decir, su generación, visualización, manejo y personalización, actualmente existen numerosas librerías que realizan todos estos procesos por nosotros. El gran problema aquí reside en que dichas librerías aunque basan su funcionamiento en tecnologías web y son compatibles con Vaadin, no pueden usarse directamente y es necesario adaptarlas antes.
Pero gracias a la activa comunidad de Vaadin, la librería Leaflet ha sido portada y añadida al repositorio de plugins. Con tan sólo añadir este plugin como dependencia en el proyecto base, conseguimos obtener todas las funcionalidades que provee dicha API.
Leaflet es una librería Javascript de código abierto para el desarrollo de aplicaciones web tanto móvil como de escritorio que requieran del manejo de mapas. Permite la visualización de mapas mediante una capa compuesta por multitud de baldosas o tiles y utilizar tantas capas de personalización como sean necesarias para adaptarlo a nuestras necesidades.

En nuestro caso se ha usado esta librería dentro del widget para conseguir las siguientes características:

- Poder cargar como base un mapa del mundo y situarnos por defecto en las coordenadas pertenecientes a España.
- Disponer de varios niveles de zoom y poder movernos libremente por el mapa.
- Agregar sobre el mapa base una capa de personalización con marcadores pertenecientes a los resultados que arrojan nuestras consultas.

6.1 Desarrollo del widget de generación de mapas

Este nuevo widget engloba todas las funciones relativas al mapa. Pero tal y como ha sucedido en widgets anteriores, antes de poder mostrar el resultado, relativo a la visualización del mapa con las diferentes variables dispuestas sobre él, es necesario previamente introducir una serie de parámetros de configuración relativos a la consulta que queremos procesar.
Para que el usuario sepa qué parte está configurando en cada momento, el menú de este widget ha sido dividido en diversas secciones, cada una con un título descriptivo de la acción a realizar. Estas secciones han sido apiladas consecutivamente de forma descendiente, en donde el lugar que ocupan depende del orden en que deben ser rellenadas, tal y como se muestra en la ilustración 6.1

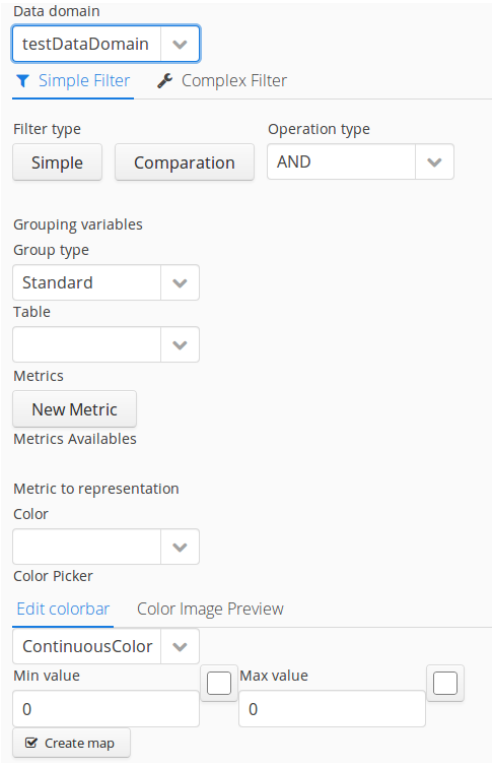

Ilustración 6.1 Menú final del widget de representación de mapas

Por un lado, relativos a la consulta tenemos: la base de datos sobre la que actuar, el filtro de datos, las variables de agrupación y las métricas, y por otro, relativos al aspecto visual que definirá el mapa, la escala de colores.

En la primera sección, debemos elegir de entre las BBDD previamente configuradas, en cual queremos que se ejecute nuestra consulta.

El siguiente apartado corresponde con el filtrado de los resultados a mostrar. Como se puede apreciar se ha hecho uso del widget de filtrado generado con anterioridad. No es necesario tener que rellenarlo si no se necesita, por tanto este paso es opcional.

La siguiente sección hace referencia al agrupamiento de resultados en torno a una o varias variables. En términos de BBDD equivale a la cláusula "*group by*" de la sentencia. Aquí será en donde se seleccionarán las variables que serán representadas en el mapa.

Cada vez que se seleccione una variable, ésta se añadirá a una lista situada debajo de ambos selectores emparentados, pudiendo eliminarse siempre que se pulse sobre el token creado.

El apartado con el título de métricas sirve para definir la parte de la consulta SQL que queremos representar, es decir, sirve para definir las métricas que serán procesadas para cada caso de la agrupación seleccionada. Así, por ejemplo, una métrica podría ser la media de edad: si agrupamos por sexo, la consulta nos devolverá la media de edad para cada sexo.

Definir una métrica puede ser, en principio tan complicado como definir un filtro complejo, por lo que el widget de definición de métricas tiene multitud de opciones. Se accede al widget pulsando sobre el botón de añadir métrica mostrado en la ilustración 6.1 y se sigue un proceso en forma de asistente. No entraremos en más detalles puesto que no es necesario para entender el funcionamiento del widget de visualización de mapas.

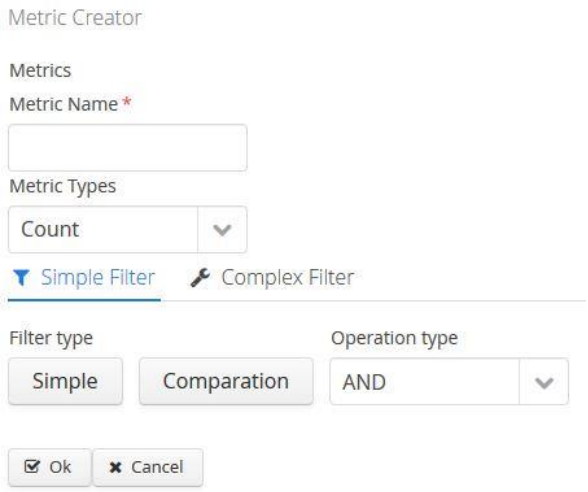

Ilustración 6.2 Widget para el manejo de métricas

En la ventana modal que se despliega mostrada en la ilustración 4.4, tendremos que poner un nombre identificativo de la métrica a crear y seleccionar del desplegable una de las opciones básicas (conteo, tasa o media).

En la 2ª parte, se encuentra una capa que contendrá las métricas creadas por el botón citado. De forma análoga a la representación hecha con las condiciones de filtrado en el widget de los filtros, aquí también se utiliza la misma estructura: una etiqueta cuyo texto describe el objeto contenido y dos botones para realizar las operaciones de editar y borrar. La diferencia es que el enlace de datos en vez de corresponder con la clase de tipo Filter ahora es de tipo Metric.

Por último la 3ª parte sirve para enlazar la métrica a su representación en el mapa.

Si disponemos de una sola métrica en el campo de métricas disponibles, solo será visible la asignación de ésta métrica al color del marcador, sin embargo si disponemos de más de una, se activará la asignación de la segunda que corresponde al tamaño del marcador. Sobre este punto existe una limitación, pues tal y como ya se ha comentado, la generación de consultas sobre los mapas puede soportar un máximo de dos dimensiones simultáneas, pero en el caso de utilizar sólo una y siendo ésta de tipo coordenada, ambas métricas serán usadas, pero en caso contrario la 2ª será ignorada.

6.1.1 Escalas de color

Para finalizar, el último apartado del menú, se corresponde con el widget definido para el tratamiento de las escalas de colores.

Este widget está formado por dos pestañas, una para la construcción de la escala y otra para la previsualización de la misma.

Dentro de la pestaña de construcción de la escala, ésta a su vez dispone de dos opciones diferentes:

Escala de color de tipo continua, que sirve para representar un intervalo de valores continuo mediante un degradado entre dos colores.

En su interior dispone de dos campos numéricos de tipo decimal para introducir el rango de valores entre los que se creará la escala y dos selectores de color asociado uno a cada valor límite en la escala.

Estos selectores de color no han tenido que ser implementados por nuestra cuenta, ya que gracias al uso de un add-on de Vaadin, se han incluido como si fueran un componente predefinido más.

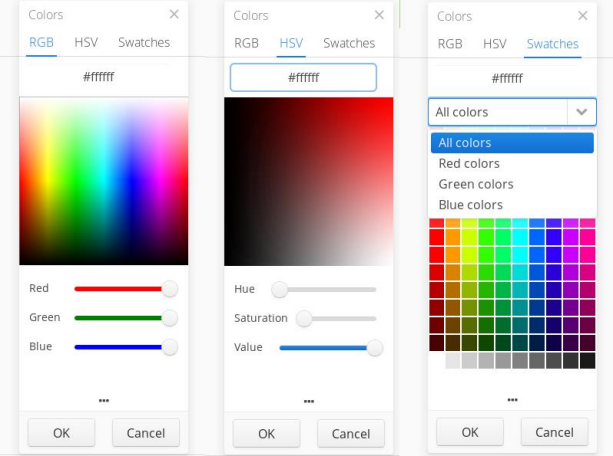

Ilustración 6.3 Selectores de color

Por su parte, la opción de la escala de color de tipo discreta, sirve para crear tantos intervalos de color como el usuario necesite.
Esta pestaña en su interior tiene dispuestos dos botones, uno para añadir un nuevo color a la lista de colores y otro para permitir definir un nuevo límite numérico de una de las secciones de las que estará compuesta la escala.
Cada vez que se genera un nuevo elemento, éste puede ser editado o borrado en cualquier momento.

Ilustración 6.4 Menú de creación de una escala de color discreta

Para que el proceso de formación de la escala dé como resultado una escala correcta será necesario que exista un color más, que campos numéricos existan. Esto es debido a que por cada rango definido en la escala se establecerá un color, y a su vez todos aquellos valores que se encuentren fuera de los límites de la escala también serán representados por colores, uno por cada extremo, tal y como se muestra en la ilustración 6.5.

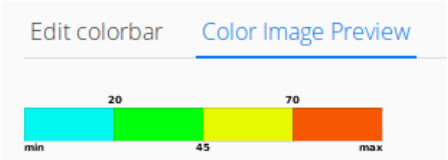

Ilustración 6.5 Ejemplo de una escala de color de tipo discreta

La pestaña de previsualización de la escala de color, necesita en primer lugar que una de las dos opciones dispuestas en la anterior pestaña hayan sido rellenadas para funcionar, sino, tan solo se mostrará una escala vacía sin valor alguno y con el color blanco por defecto.
A la hora de generar la escala, el método dispuesto para tal fin leerá el modo de escala seleccionado, los colores elegidos juntos con los valores numéricos y devolverá una imagen en formato .png con la representación.

Dicha representación no es necesaria para la generación del mapa con el resultado de la consulta, pero si sirve para que el usuario tenga una idea de cómo quedarán coloreados los valores sobre el mapa.

Por último, nos queda por explicar el proceso de generación del mapa una vez tenemos todos los parámetros introducidos.

El primer paso del widget una vez invocado es generar la consulta SQL que será ejecutada sobre la BBDD asociada, para obtener los datos. Este proceso de la generación de la consulta ha sido posible gracias a un servicio creado por la empresa Predictia, permitiéndonos de este modo obviar los detalles de la implementación interna. Este servicio recibe como entradas los objetos de modelo que ya hemos preparado en widgets anteriores como: Filter, generado por el widget de filtros, GroupParameter, generado por el combo de variable de agrupación y Metric, generado por el widget de métricas. Este servicio nos devuelve directamente el resultado de la consulta sobre la base de datos, asociando cada valor numérico con el objeto de métrica suministrado. Poniendo un ejemplo:

Como parámetro de agrupación tenemos el sexo y como métrica, el conteo, obteniendo un resultado de:

Sexo = 1, conteo = 1246

Sexo = 2, conteo = 386

6.1.2 Tipos de mapas según su representación

Una vez obtenida la información con los resultados, el siguiente paso es analizar las variables introducidas para comprobar qué tipo de representación va a tener que ser generada, puesto que dependiendo del número de dimensiones a tratar, unido al tipo de variable a procesar, junto el número de métricas seleccionadas obtendremos como resultado una de las siguientes opciones.

- Solo una dimensión asociada a una variable con componente geográfico.
  - Si la variable es de tipo coordenada, el resultado será una serie de puntos distribuidos sobre el mapa
    - Si solo se dispone de una métrica, dicha métrica será la base para definir el color de cada marcador dependiendo de la escala de colores.
    - Si por el contrario se han definido dos, una para el color y otra para la dimensión. Cada marcador generado variará en color y tamaño de forma relativa al resultado de ambas métricas.

  - Si es de tipo polígono, el resultado será una serie de formas geométricas representando distintas áreas sobre el mapa.

- Dos dimensiones asociadas a una variable geográfica y una discreta.
  - Si la variable geográfica es de tipo coordenada, se generará un gráfico sectorial representando en porcentaje los diferentes estados de la variable discreta por cada punto del mapa.
  - Si es de tipo polígono, se generará de igual forma que en el anterior caso un gráfico sectorial, pero está vez cada uno de ellos estará situado en el centro del área que marca dicho polígono.

Independientemente de la elección de cualquiera de estas opciones, sobre todas ellas se ha agrupado la información mediante el uso de un cluster. Esta funcionalidad se ha conseguido gracias a la inclusión de un plugin especialmente diseñado para actuar sobre la librería Leaflet en Vaadin.

El funcionamiento de estos clusters es el siguiente: dependiendo del nivel de zoom sobre el mapa, si los valores expuestos sobre él, es decir, el número de puntos es demasiado alto para realizar una visualización cómoda, se irán agrupando en pequeños conjuntos. La formación de estos conjuntos dependerá de la distancia a la que se encuentren unos puntos de otros respectivamente.

Estos conjuntos han sido representados mediante un círculo junto con un número en su interior. Este número indica la cantidad de puntos que están contenidos.
Para finalizar, pasemos a explicar más en detalle la implementación realizada capaz de generar el mapa con los resultados.
Debido a que para poder hacer una representación visual de todos los datos recibidos, ha sido necesario utilizar estructuras en algunos casos algo complejas para facilitar la tarea al desarrollador. El código generado en los casos más complicados ha sido extenso, lo que provoca que la lectura de este código pueda resultar monótona y aburrida al lector. Es por ello, que se ha optado por presentar sólo los casos más sencillos, haciéndose una idea de los siguientes más complejos.

Comencemos por la generación de marcadores sobre el mapa con la selección de una sola dimensión y variable de tipo coordenada:
Para este caso en concreto, el método asociado es capaz de tratar ambas opciones en cuanto al número de métricas seleccionadas se refiere.

```
1 private void addLocationMarkersToMap(Column column,
2  AssingMetricToRepresentation representation, Colorbar optional,
3 GroupResult groupResult) {
4     LocationVariable locationVariable =
      (LocationVariable)column.getVariable();
5     Map<String, org.springframework.data.geo.Point> pointsMap =
      createPointsMap(locationVariable);
6     HashMap<Number, StreamResource> imageMap =
      new HashMap<Number, StreamResource>();
7     int max = Integer.MIN_VALUE;
8     if(representation.getComboMetric2()!=null) {
9           int aux = 0;
10          for(GroupRow row : groupResult) {
11                OutputGroupField<Object> ogf =
                  row.getOutputGroupField(representation.getMetric2())
12                if(ogf.getValue() instanceof Number){
13                      aux = ((Number)ogf.getValue()).intValue();
14                      if(max < aux){
15                            max = aux;
16                      }
17                }
18          }
19    }
20    for(GroupRow row : groupResult) {
21          ColumnGroupField<Object> vgf =
            row.getColumnGroupField(column);
22          if(vgf.isTotalValue() || vgf.getValue() == null){
23                continue;
24          }
25          if(pointsMap.containsKey(vgf.getValue())) {
26                org.springframework.data.geo.Point p =
                  pointsMap.get(vgf.getValue());
27                if(Double.valueOf(p.getX()).isNaN()) {
28                      continue;
29                }
30                Number value = 0;
31                StringBuilder popup = new StringBuilder();
32                popup.append(locationVariable.
                  findLocation(vgf.getValue().toString())
                  .get().getDescription());
33                int tam = 20;
34                String aux= "";
35                if(representation.getComboMetric2()!=null) {
```

```
36                          value  =
                            (Number)row.getOutputGroupField(
                            representation.getMetric2()).getValue();
37                          aux =(" "+representation.getMetric2().getName()+
                            " "+decimalFormatter.format(value.doubleValue()));
38                          tam = (int) (value.doubleValue()/max*20+20);
39                    }
40                    LMarker marker =
                      new LMarker(new Point(p.getY(), p.getX()));
41                    marker.setIconSize(new Point(tam, tam));
42                    StreamResource resource;
43                    if(!imageMap.containsKey(value)) {
44                          SvgOvalImageSource img2;
45                          if(representation.getMetric1()!=null) {
46                                value =
                                  (Number) row.getOutputGroupField(
                                  representation.getMetric1()).getValue();
47                                popup.append(" "+
                                  representation.getMetric1().getName()+""+
                                  decimalFormatter.format(value.doubleValue()))
48                                if(optional.isPresent()) {
49                                      img2 =
                                        new   SvgOvalImageSource(   tam,   tam,
                                        optional.get().getColor(
                                        value.floatValue()));
50                                } else {
51                                      img2 =
                                        new     SvgOvalImageSource(tam,     tam,
                                        Color.GRAY);
52                                }
53                          } else {
54                                img2 =
                                  new SvgOvalImageSource(tam, tam, Color.GRAY);
55                          }
56                          resource =
                            new                                   StreamResource(
                            img2,"image"+System.currentTimeMillis()+".svg");
                            imageMap.put(value, resource);
57                    } else {
58                          resource = imageMap.get(value);
59                    }
60                    marker.setIcon(resource);
61                    popup.append(aux);
62                    marker.setPopup(popup.toString());
63                    lMarkerClusterGroup.addComponent(marker);
64              }
65        }
66  }
```

Como parámetros de entrada recibe: la columna con la variable a procesar, la lista de métricas, la escala de color, y el resultado del agrupamiento.

El primer paso es obtener todos los puntos que se van a visualizar en el mapa y en el caso de haberse seleccionado una métrica para la dimensión del marcador, conseguir el tamaño máximo que tendrá el marcador. A continuación recorremos la lista con todos los puntos y para cada uno de ellos creamos un marcador dentro del mapa generado por Leaflet. Ahora creamos una imagen con formato svg cuyo contenido sea un círculo con el tamaño y color especificado por los datos relativos a este punto y se la asociamos como icono al marcador. Posteriormente generamos un tooltip con la información de la consulta para este marcador y se la asociamos también. Por ultimo añadimos el marcador al cluster de marcadores.

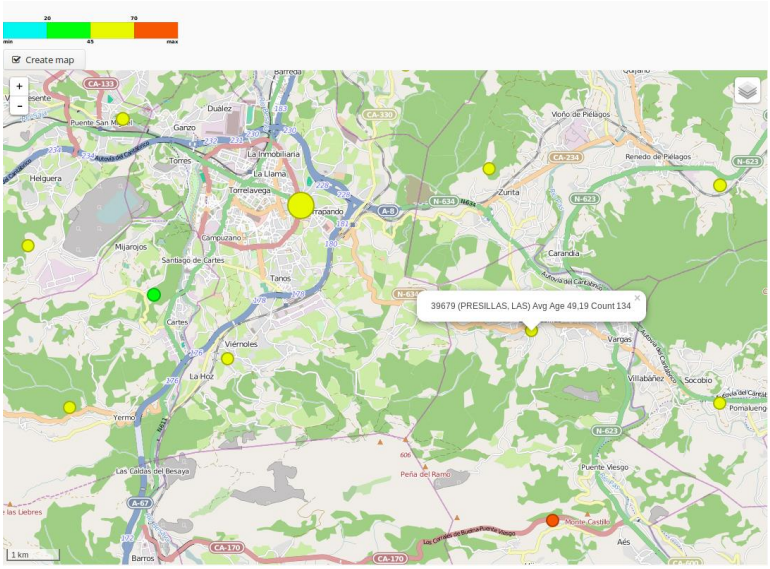

Ilustración 6.6 Representación de una variable de tipo coordenada con dos métricas

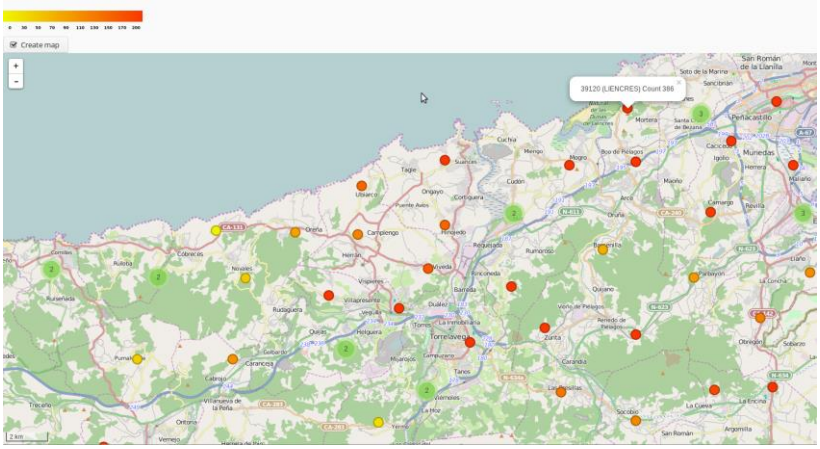

Ilustración 6.7 Representación de una variable de tipo coordenada con una sola métrica

Si por el contrario la opción fuera la contraria, es decir, la variable asociada a la agrupación fuera poligonal, todos los pasos son los mismo, salvo por el hecho que en vez de generar un marcado por cada coordenada, recorreríamos todos los puntos pertenecientes al polígono, generando una forma geométrica equivalente, la cual sería el marcador propiamente en sí.

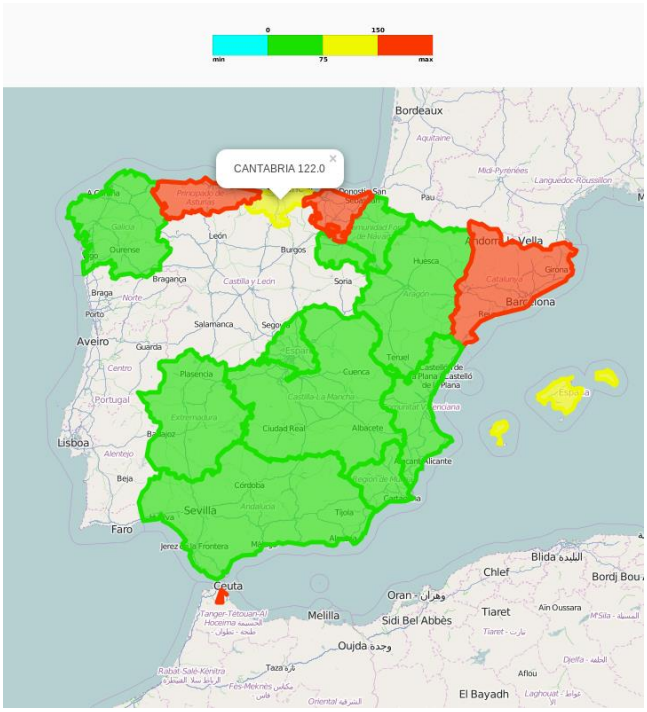

Ilustración 6.8 Representación de una variable poligonal

Para el tratamiento de las siguientes dos opciones que quedan respecto a la generación de marcadores sobre el mapa, debido al uso de una segunda variable de agrupamiento de tipo discreta. Ahora el icono que representa al marcador en vez de ser un simple círculo, será sustituido por una imagen representando un gráfico sectorial con el resultado.

Para la generación de los gráficos sectorial dispuestos sobre el mapa, se ha utilizado la librería jFreeChart. Dicha librería está escrita en Java y permite la generación de multitud de gráficos diferentes con una alta personalización.

Sobre su uso, aunque resulta en un primer momento imponente la cantidad de opciones que ofrece, en nuestro caso, no nos hizo falta disponer de un gran conocimiento de la herramienta para poder generar los gráficos sectoriales, puesto que se usaron las opciones más básicas.

Bastó con elegir el tipo de gráfico a crear, unos colores para cada estado de la variable a procesar y un tamaño y tipo de imagen de salida para obtener una configuración más que aceptable.

```
1 private StreamResource getCharImage(DefaultPieDataset pieDataSet,
  ColorGenerator colorGen){
2     Color transparentColor = new Color(1f,1f,1f,0);
3     List<String> list = pieDataSet.getKeys();
4     PiePlot piePlot = new PiePlot(pieDataSet);
5     Map<String, Color> colors = colorGen.getMapColor();
6     for(String s : list) {
7           piePlot.setSectionPaint(s, colors.get(s));
8     }
9     piePlot.setBackgroundPaint(transparentColor);
10    piePlot.setBackgroundImageAlpha(0.0f);
11    piePlot.setLabelGenerator(null);
12    piePlot.setSectionOutlinesVisible(false);
13    piePlot.setOutlineVisible(false);
14    piePlot.setInteriorGap(0.0D);
15    piePlot.setShadowPaint(transparentColor);
16    JFreeChart chart = new JFreeChart("",piePlot);
17    chart.setBackgroundPaint(transparentColor);
18    chart.setBackgroundImageAlpha(0.0f);
19    chart.removeLegend();
20    BufferedImageStreamSource img =
      new BufferedImageStreamSource(chart.createBufferedImage(100,100));
21    return new StreamResource(img,
22    "image"+System.currentTimeMillis()+".png");
23 }
```

Además para poder controlar en cualquier momento, la visualización de las distintas variables dispuestas sobre el gráfico de sectores, se ha creado un pequeño widget bastante simple, el cual enlazado al mapa, nos permite gestionar cada uno de los diferentes estados que contiene la variable discreta de forma separada.

En resumidas cuentas, este widget se encarga de generar tantos botones como estados de la variable se vayan a mostrar. Identificando cada uno de ellos mediante un icono con el color asociado a dicho estado, junto con el nombre del estado en sí. De este modo si pulsáramos en cualquiera de ellos, el estado asociado a él se eliminaría de la representación que actualmente se muestra en el mapa, sucediendo exactamente al contrario si se volviera a pulsar.

En el caso de intentar representar dos dimensiones, una concerniente a una variable de tipo coordenada y otra de tipo discreta, el método asociado es el siguiente:

```
1 private void addPieChartLocationMarkersToMap(Column geoColumn,
  Column pieColumn, GroupResult groupResult, VisualizationContext context)
  throws Exception {
2     LMarker marker;
3     Map<ColumnGroupField<Object>,    Map<String,    Number>>    groupedMap    =
      groupVariablesByGeoPosition(groupResult, context, geoColumn, pieColumn);
4     ColorGenerator colorGen = new ColorGenerator(groupedMap);
5     if(checkIfOtherValueIsPresent(groupedMap)) {
```

```
6           colorGen.getColor("Others");
7     }
8     if(buttonList==null        ||        buttonList.getButtonList().keySet()
      .size()!=colorGen.size()) {
9           buttonList = new ColorButtonList(colorGen);
10    } else {
11          buttonList.updateColors(colorGen);
12    }
13    LocationVariable locationVariable =
      (LocationVariable)geoColumn.getVariable();
14    Map<String, org.springframework.data.geo.Point> pointsMap =
      createPointsMap(locationVariable);
15    for (ColumnGroupField<Object> vgf :groupedMap.keySet()) {
16          if(pointsMap.containsKey(vgf.getValue())){
17                org.springframework.data.geo.Point p =
                  pointsMap.get(vgf.getValue());
18                if(Double.valueOf(p.getX()).isNaN()) {
19                      continue;
20                }
21                int tam = 100;
22                marker = new LMarker(new Point(p.getY(), p.getX()));
23                marker.setIconSize(new Point(tam, tam));
24                Map<String,Number> drawPieMap = groupedMap.get(vgf);
25                DefaultPieDataset pieDataSet =
                  createPieDataset(drawPieMap, buttonList);
26                BufferedImageStreamSource         img         =         new
                  BufferedImageStreamSource(getCharImage(pieDataSet,colorGen)
27                StreamResource resource = new StreamResource(img,
                  "image"+System.currentTimeMillis()+".png");
28                marker.setIcon(resource);
29                StringBuilder popup = new StringBuilder();
30                popup.append(locationVariable.findLocation(vgf.getValue()
                  .toString()).get().getDescription()+" ");
31                double total = 0;
32                for(String s :drawPieMap.keySet()) {
33                      total += drawPieMap.get(s).doubleValue();
34                }
35                for(final String s :drawPieMap.keySet()) {
36                      popup.append(" "+s+""+
                        percentFormatter.format(drawPieMap.get(s).
                        doubleValue()/total));
38                }
39                marker.setPopup(popup.toString());
40                lMarkerClusterGroup.addComponent(marker);
41          }
42    }
43 }
```

Explicando los pasos a seguir más concretamente serían: Primero se organizan los datos recibidos dentro de una estructura que permite su manejo de forma más simple. A continuación se genera el widget explicado anteriormente con la lista de botones asociados a cada valor posible de la variable de tipo discreta.

Posteriormente se recorre la lista con las coordenadas creándose un marcador por cada una de ellas. Al mismo tiempo se realiza una llamada a jFreeChart con los datos asociados a dicha coordenada, devolviéndonos un gráfico de sectores ya completo.

Dicho gráfico se asocia como icono del marcador y se genera un tooltip descriptivo con la información procesada. Y por último se agrega dicho marcador al cluster de marcadores.

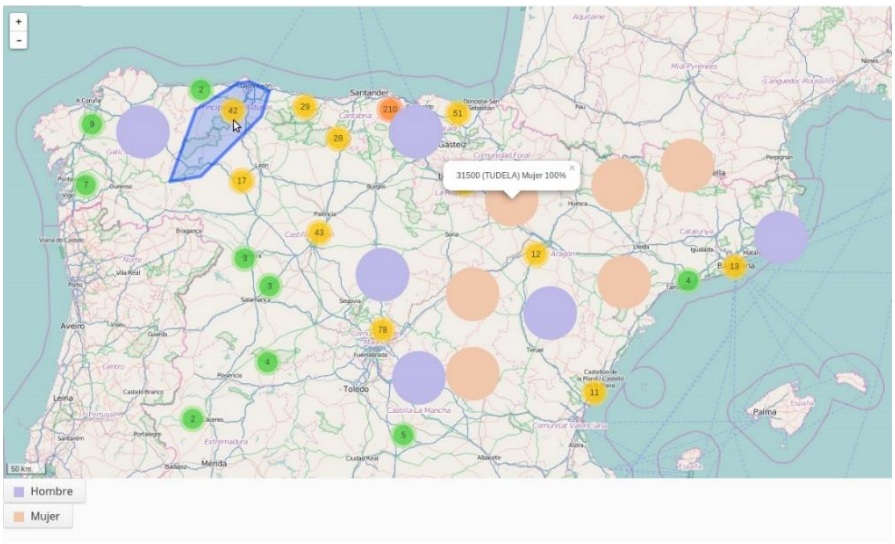

Ilustración 6.9 Representación de dos variables, siendo una de tipo coordenada

Si por el contrario la variable con componente geográfico fuera de tipo poligonal, el proceso es análogo al caso anterior, salvo porque en vez de ir punto a punto, vamos recorriendo polígono a polígono, generando para cada figura su contorno de forma invisible y calculando el punto medio interior de este contorno, para posicionar el gráfico sobre dicho punto.

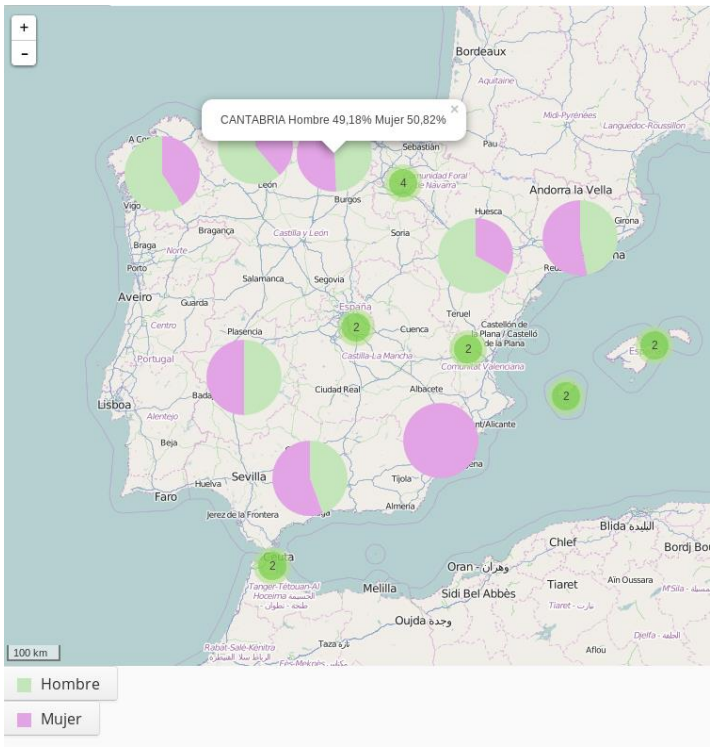

Ilustración 6.10 Representación de dos variables, siendo una de tipo poligonal

7 Conclusiones y trabajos futuros

Antes de comenzar con el proyecto de fin de carrera, pensaba que el desarrollo de aplicaciones orientadas completamente al ámbito web, era complicado y complejo al ser necesario el manejo de diversos lenguajes o tecnologías de forma conjunta y conseguir que todas ellas funcionaran en armonía.

Además, desde hace unos años he podido comprobar cómo evolucionaba la web y las expectativas que actualmente tienen los usuarios en cuanto a la interfaz visual se refiere. Esperan que sea funcional, bonita, pero sobre todo simple.

Una vez completado el proyecto y manejado Vaadin con soltura, puedo afirmar que mi antigua creencia sobre la complejidad de este tipo de desarrollos estaba equivocada.

Gracias a este framework incluso personas sin gran conocimiento previo sobre desarrollo web, pueden llegar a conseguir unos buenos resultados con tan solo disponer de una base de conocimiento en el lenguaje Java.

El hecho de disponer de todas las herramientas y características propias de un lenguaje orientado a objetos, resulta altamente gratificante y eficaz a la hora de desarrollar, pudiendo resolver los problemas encontrados de un modo familiar, tal y como se haría en una aplicación de escritorio.

Por todo ello, considero que el devenir futuro del desarrollo web, pasa por utilizar este tipo de soluciones que se encarguen de simplificar la tarea de desarrollador en gran medida.

También he de destacar el gran salto que supone para el alumno enfrentarse por primera vez a una aplicación real y no manejar un simple prototipo de "juguete".

Tener que integrarse con una arquitectura ya dada, adaptarse al código que previamente ha sido generado, ser capaz de manejar de forma eficaz los diversos módulos de los que está compuesta la aplicación o simplemente trabajar con un equipo de forma simultánea puede suponer todo un reto.

Este momento es en el que te das cuenta de la importancia que tiene aplicar todo el conocimiento aprendido sobre el desarrollo software durante la carrera.

7.1 Trabajos Futuros

Como posibles mejoras o ampliaciones futuras de la aplicación, hay tres vías principalmente:

La primera es relativa al widget de creación de métricas, puesto que aunque el desarrollo actual de este módulo está terminado, debido a la falta de tiempo, sus funcionalidades tuvieron que ser acortadas.

Actualmente las métricas que dicho widget puede manejar, alcanzan una complejidad media-alta, pero por desgracia los casos más complejos y largos no tienen cabida en este widget, y es ahí donde se encontraría el punto de ampliación.

La otra ampliación posible sería desarrollar un nuevo conjunto de widgets cuyo foco sea la representación del resto de datos que es capaz de manejar la aplicación mediante diversos tipos de gráficas, tales como gráficos de barras, líneas temporales, etc que haciendo uso de una base ya establecida, pudieran ser configuradas a gusto del usuario.

La última vía, sería generar un gestor de filtros y métricas frecuentes para que se puedan definir a nivel de aplicación y los usuarios puedan elegirlas mediante un desplegable.

8 Referencias

Duarte, A. (2013). *Vaadin 7 UI Design by Example.* Packt Publishing.

Fränkel, N. (2013). *Learning Vaadin 7.* Packt Publishing.

Grönroos, M. (2014). *Book of Vaadin:Vaadin 7 Edition - 3rd Revision.* Vaadin Ltd.

Holaň, J. (2013). *Vaadin 7 Cookbook.* Packt Publishing.

JFreeChart. (2 de 04 de 2015). Obtenido de jfree: http://www.jfree.org/jfreechart/index.html

Leaflet. (17 de 03 de 2015). Obtenido de Leaflet: http://leafletjs.com/

Neo4j. (12 de 02 de 2015). Obtenido de Neo4j: http://neo4j.com/developer/graph-db-vs-rdbms/

Neo4J. (12 de 03 de 2015). Obtenido de Neo4J: http://docs.neo4j.org/chunked/stable/images/RDBMSvsGraph.svg.png

Pöntelin, T. (14 de 03 de 2015). *WizardForVaadin*. Obtenido de github: https://github.com/tehapo/WizardsForVaadin

Spring-framework. (10 de 03 de 2015). Obtenido de Spring.framework: http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/overview.html#overview-modules

Vaadin. (11 de 03 de 2015). Obtenido de Vaadin: https://vaadin.com/gwt

Wikipedia. (6 de 02 de 2015). Obtenido de Wikipedia: http://es.wikipedia.org/wiki/Spring_Framework