CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid
Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Aplicación Web para Editar Licencias Digitales

Autor: Guillermo Gutiérrez Lorenzo
Directora: María del Carmen Suárez de Figueroa Baonza
Co-director: Víctor Rodríguez Doncel

MADRID, ENERO 2018

Índice

Resumen

Este trabajo presenta el desarrollo de una aplicación web para la edición de licencias digitales ODRL. Estas políticas (licencias), compuestas en RDF, permiten expresar permisos y prohibiciones sobre recursos digitales, explicitando elementos como a quien van dirigidos, acciones permitidas, restricciones temporales, etc.

La aplicación desarrollada para este trabajo está disponible online en: http://odrleditor.appspot.com/ .

El editor desarrollado consiste en un formulario que se va creado dinámicamente según los elementos que va añadiendo el usuario, abstrayéndole de toda la complejidad de las reglas para componer una política ODRL, y limitando lo menos posible las posibilidades que brindan. Al final, el usuario puede obtener una política en diferentes formatos, así como información sobre su validez.

Tras el estudio del funcionamiento de las políticas ODRL, se plantea el diseño de una interfaz lo más sencilla posible, y que ayude al usuario a comprender la estructura de la política que está generando.

Vemos como durante la implementación del sistema hemos lidiado con los problemas que suponen generar un formulario HTTP dinámicamente desde el *front-end*, y que tiene que poder leer el *back-end*. Durante el desarrollo se van escogiendo las tecnologías que han permitido la implementación de diferentes funcionalidades que ha ido requiriendo el sistema, entre las que se encuentran: Spring, jQuery, Bootstrap, Jackson, Apache HTTP Client, Apache Jena, y Google App Engine. De estas tecnologías mostramos como se han integrado en el sistema para obtener el resultado final.

Abstract

This work presents the development of a web application for the edition of ODRL digital licenses. These politics (licenses), composed in RDF, allow to express permissions and prohibitions on digital resources, specifying elements such as who they are addressed to, allowed actions, temporary restrictions, etc.

The application developed for this work is available online at: http://odrleditor.appspot.com/.

The developed editor consists of a form that is dynamically created according to the elements that the user is adding, abstracting him from the complexity of the rules to compose an ODRL politic, and narrowing as less as possible the possibilities that they offer. In the end the user can obtain a licence in different formats, as well as information about its validity.

After studying the operation of the ODRL politics, the design of an interface is proposed, being as simple as possible, and helping the user to understand the structure of the policy that is being generated.

During the implementation of the system we see how we have dealt with the problems that involve generating a HTTP form dynamically from the front-end, and that the back-end must be able to read. During the development are chosen the technologies that have allowed the implementation of different functionalities that have been required by the system, among which are: Spring, jQuery, Bootstrap, Jackson, Apache HTTP Client, Apache Jena, and Google App Engine. From these technologies we show how they have been integrated into the system to obtain the final result.

1. Introducción

En este primer capítulo introducimos el trabajo para comprender bien el contexto, el objetivo, y la problemática a la que nos enfrentamos a la hora de desarrollar la aplicación. Al final encontramos un pequeño desglose de la lista inicial de los objetivos del trabajo.

La aplicación está disponible en http://odrleditor.appspot.com/. Esto ha sido posible gracias a Google, que en el contexto de un trabajo académico nos ha cedido crédito en Cloud Platform para poder desplegarla, y a Víctor Rodríguez Doncel que se encargó de contactar con ellos.

1.1. Introducción al trabajo

Con el crecimiento que están experimentando los recursos digitales en los últimos años, surge la necesidad de regular su uso para asegurarse de que se ajuste a las condiciones expuestas por su autor. Estas condiciones normalmente están expresadas en una política de uso de los datos, pero crear estas políticas suele ser tedioso, puede ser complicado ajustarla a cada uno de los recursos, y se complica a la hora de añadir las condiciones de otros recursos que se han utilizado para generar uno diferente, que tendrá sus propias condiciones a su vez.

Con el fin de facilitar y automatizar el uso de estas políticas surge **ODRL** [2] (Open Digital Rights Language), que trata de unificarlas y flexibilizarlas, y expresadas de tal manera que puedan ser leídas y usadas por sistemas digitales.

ODRL se plantea como un lenguaje que permite plasmar permisos, prohibiciones, y obligaciones sobre recursos, expresado en **RDF**, y serializado como diferentes formatos de datos (entre ellos JSON-LD) [1].

El problema se genera porque plasmar estas políticas en formatos orientados a sistemas digitales requiere del conocimiento del lenguaje de estos formatos que no son, en una primera instancia, manejables y comprensibles por una persona.

```
{
    "@context": "http://www.w3.org/ns/odrl.jsonld",
    "@type": "Set",
    "uid": "http://example.com/policy:1010",
    "permission": [{
        "target": "http://example.com/asset:9898.movie",
        "action": "use"
    }]
}
```

Figura 1: Ejemplo de una política ODRL en JSON-LD

Con el fin de facilitar la creación de estas políticas por personas surge este trabajo, que trata de hacer una **aplicación** Web para la creación de **políticas ODRL** en JSON-LD. La idea de esta aplicación es proporcionar un formulario donde el propio usuario va añadiendo y rellenando campos según las necesidades que se le presentan para expresar sus condiciones sobre un recurso, y la aplicación te devuelve la política en formato JSON-LD (aunque como veremos más tarde, se le añaden más formatos) con los datos proporcionados.

1.2. Problemática

Generar políticas flexibles que puedan expresar cualquier condición de un autor sobre un recurso requiere de un lenguaje extenso, y que no tenga límites en cuanto al nivel de complejidad al que se pueda llegar anidando condiciones y recursos afectados. Por eso ODRL proporciona una extensa documentación sobre su modelo y vocabulario [3] [4].

Por lo tanto, la problemática de una aplicación Web para ayudar a los usuarios a generar estas políticas gira entorno a: por una parte, darles un formulario que sea fácil de entender a pesar de todas las posibilidades que tienen para generar la política, y por otra que la aplicación sea capaz de abarcar todas estas posibilidades de formularios distintos para generar políticas en JSON-LD.

Necesitamos una aplicación donde el usuario vaya sumando poco a poco todos los campos que necesita de manera guiada, indicándole a cada paso cada una de las posibilidades de campos que tiene para expresar sus necesidades y plasmar sus datos. Este problema se presenta por la flexibilidad que tienen las políticas ODRL, y lo que indicaría el fracaso de intentar resolverlo sería que el usuario optase por rellenar él mismo una política en JSON-LD u otro formato, en vez de usar la aplicación, lo cual se complica si no queremos renunciar a flexibilidad.

El sistema tiene que poder procesar estos formularios de tal forma que no necesite conocer todos los campos que tiene antes de recibirlo, y por lo tanto nos debemos encargar que la estructura de datos que se vaya a usar sea modular y flexible.

1.3. Objetivos

Para poder resolver los problemas planteados, gran parte del tiempo está dedicado a encontrar la combinación de tecnologías y herramientas que nos vayan a permitir cumplir los siguientes objetivos:

- Crear un formulario HTML que sea capaz de irse generando dinámicamente.
- Crear una interfaz sencilla e intuitiva que ayude al usuario en su tarea
- Construir un back-end que admita peticiones con diferentes formularios que cumplan un patrón.
- Generar bloques de texto en formato JSON-LD a partir de los datos proporcionados.
- Crear la Aplicación Web que permite generar políticas ODRL en JSON-LD, empezando por políticas sencillas e ir añadiendo prestaciones.
- Añadir la posibilidad de cargar políticas ODRL en JSON-LD para visualizarlas en la aplicación.

Como objetivo adicional se plantea el despliegue de la aplicación en una plataforma como App Engine de Google, para que la aplicación esté disponible para cualquiera que quiera usarla.

2. Estado del arte

En este capítulo estudiamos el contexto en el que se desarrolla el trabajo, que trabajos relacionados ha habido antes, y las tecnologías sobre las que se desarrolla la aplicación.

2.1. ODRL

W3C proporciona un **modelo de información** de ODRL que explica cómo funcionan las **políticas** (licencias) que se pueden crear con el lenguaje. Leer el modelo y entenderlo es lo primero para comprender cómo tienen que ser las funcionalidades.

Aquí intentamos explicar el **funcionamiento básico** de estas políticas. El modelo reúne sus principales componentes en el modelo de la figura 2, aunque las relaciones son (como se explica más adelante) más complejas:

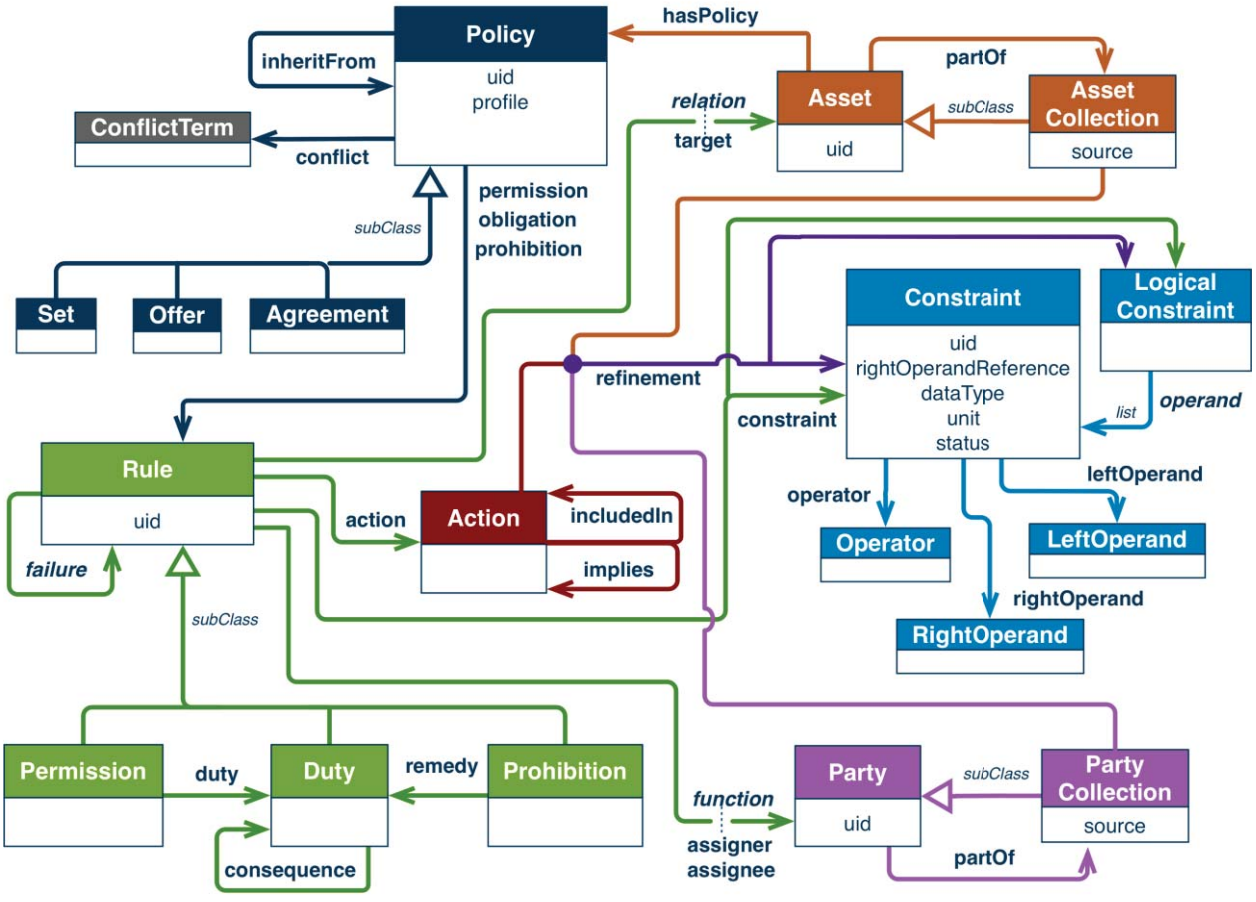

Figura 2: Modelo de información de ODRL. (Fuente: W3C)

De forma general, una **política** (*policy*) tiene **reglas** (*rules*, p.e. un permiso o una prohibición), y las reglas tienen una **acción** (*action*, p.e. imprimir o reproducir), un **objetivo** (*target*, p.e. un video o una publicación), y pueden contener **cesionarios** (*assignees*, que recibe los derechos) y **cedentes** (*assigners*, que concede los derechos). La política más básica se compone de una regla con una acción y un objetivo (figura 1).

Para que las reglas den un significado a las acciones, las reglas están divididas en 3 tipos:

-**Permiso** (*Permission*): autorizan una acción sobre un objetivo.
-**Deber** (*Duty*): obliga a realizar una acción.
-**Prohibición** (*Prohibition*): inhabilita una acción sobre un objetivo.

Las reglas pueden estar relacionadas entre ellas. Por ejemplo, un permiso puede estar relacionado con un deber, de tal manera que no se puede hacer uso del permiso a menos de que se cumpla un deber.

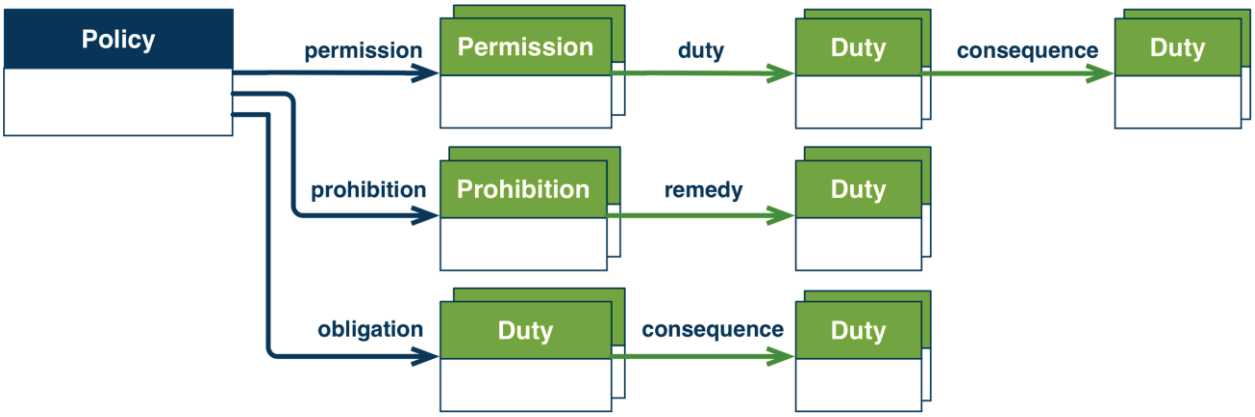

Figura 3: Modelización de las relaciones entre reglas en ODRL. Fuente: W3C.

La forma de **identificar** algunos de estos **recursos** (política, objetivos, cesionarios, cedentes…) para poderlos **referenciar** es un identificador único (**UID**, Unique IDentifier), que normalmente va a ser una **URI** (Uniform Resource Identifier [14]).

Los **objetivos** (target) son **activos** (*assets*). Estos activos pueden ser directamente el activo en cuestión, o una **referencia** a una **colección** de ellos, de tal manera que se asigna la misma política a diferentes entidades a la vez.

De una manera parecida los **cesionarios** (*asignees*) y los **cedentes** (*assigners*) son **grupos** (*parties*, *party*). Los grupos pueden ser también la entidad en cuestión, o una **referencia** a una **colección** de entidades.

Hasta ahora hemos visto como construir una política funcional pero bastante **limitada** en cuanto a la especificidad de las reglas, por ejemplo, si queremos que un permiso solo se pueda utilizar durante un mes, con lo visto hasta este punto no podríamos hacerlo. Lo que le da mucha de la **flexibilidad** a estas políticas son las **restricciones** (*constraints*).

La restricción es básicamente una operación que se compone de **2 operandos** (*LeftOperand* y *RightOperand*) y **1 operador** (*Operator*) que se tiene que cumplir para validarla. Una restricción podría ser por ejemplo que la fecha actual sea menor que el 1 de enero de 2018. Estas restricciones son aplicables a reglas, acciones, y colecciones de activos y grupos.

Las **restricciones** pueden ser también **operaciones lógicas** de otras restricciones. Por ejemplo, dos restricciones, una que implique que la fecha actual sea menor que el 1 de enero de 2018, y otra que implique que la fecha actual sea mayor que el 1 de diciembre de 2017, ambas unidas por el operador lógico "**y**" ("and").

```
{
    "@context": "http://www.w3.org/ns/odrl.jsonld",
    "@type": "Offer",
    "uid": "http://example.com/policy:88",
    "profile": "http://example.com/odrl:profile:10",
    "permission": [{
        "target": "http://example.com/book/1999",
        "assigner": "http://example.com/org/paisley-park",
        "action": [{
           "rdf:value": { "@id": "odrl:reproduce" },
           "refinement": {
               "xone": [ { "@id": "http://example.com/p:88/C1" },
                         { "@id": "http://example.com/p:88/C2" } ]
            }
        }]
    }]
}

{
   "@context": "http://www.w3.org/ns/odrl.jsonld",
   "@type": "Constraint",
   "uid": "http://example.com/p:88/C1",
   "leftOperand": "media",
   "operator": "eq",
   "rightOperand": "online"
}

{
   "@context": "http://www.w3.org/ns/odrl.jsonld",
   "@type": "Constraint",
   "uid": "http://example.com/p:88/C2",
   "leftOperand": "media",
   "operator": "eq",
   "rightOperand": "print"
}
```

Figura 4: Ejemplo de una política con una acción refinada con una restricción lógica "xone" de dos operandos, y las dos restricciones ODRL.

Por último, las políticas adquieren diferente nombre en función de qué tipo de política sea. Una política puede ser de 3 tipos diferentes:

-**Conjunto** (*Set*): se compone de cualquier conjunto de reglas.

-**Oferta** (*Offer*): se compone de reglas que son ofertas hechas por el cedente.

-**Acuerdo** (*Agreement*): se compone de reglas que implican el acuerdo entre cedente y cesionario.

El modelo de ODRL se compone de muchas más especificaciones y atributos de los elementos, sobre todo en cuanto a la forma y relaciones que deben tomar entre ellos, pero los elementos principales y los esenciales para entender el trabajo son los presentados. Además, a lo largo de la memoria aparecerán más especificaciones que serán explicadas para comprender mejor las dificultades que presentan en el desarrollo.

2.2. Trabajos con ODRL

ODRL es relativamente nuevo, y por lo tanto no se pueden encontrar muchos trabajos que tengan una relación con la aplicación a desarrollar. En la página del grupo de W3C podemos encontrar algunas implementaciones [11], pero son de versiones anteriores de ODRL (actualmente en la versión 2.2) y no se han actualizado, o directamente ya no están disponibles.

El trabajo más interesante con W3C es el validador de políticas ODRL de Víctor Rodríguez Doncel (Co-director del trabajo) [12], del cual está disponible el código en GitHub [13]. Esta aplicación es capaz de validar políticas ODRL tanto en JSON-LD como en Turtle, y de indicar qué elementos fallan en caso de que sea erróneo. También proporciona una API REST con la que podemos hacer validaciones desde otras aplicaciones. Quizás la parte que más nos pueda interesar de este trabajo sea la posibilidad de validar y obtener información de las políticas en nuestra aplicación con la API REST, y sobre todo poder ir comprobando durante el desarrollo que las políticas generadas son correctas.

2.3. Trabajos con licencias digitales

Otro tipo de licencias digitales son las licencias XACML (eXtensible Access Control Markup Language). Estas licencias están expresadas en XML, pero no entran dentro de la web-semántica. Su función también es controlar el uso de recursos, aunque está enfocado directamente a un entorno de servidores de control de acceso. Tienen una composición parecida a ODRL, donde hay reglas definidas como permisos y prohibiciones, restricciones, acciones, entidades etc [19].

WSO2, un proveedor de tecnología *open-source,* ofrece un servidor de control de recursos que utiliza políticas XACML, y dentro del panel de control del servidor encontramos un editor de licencias XACML. Este editor tiene diferentes vistas para diferentes niveles de flexibilidad y complejidad. Al ser un servidor con base de datos y que controla recursos, sus políticas se construyen más elementalmente, por ejemplo: puedes crear un usuario, puedes hacer

referencia a ese usuario desde una regla, la regla se guarda en la base de datos, desde el panel de creación de políticas puedes buscar y hacer referencia a esa regla, etc. Su uso está enfocado a los administradores del sistema [20].

Sus formularios comparten muchas similitudes en forma y vocabulario con el de nuestra aplicación final, como podemos ver en la figura 5.

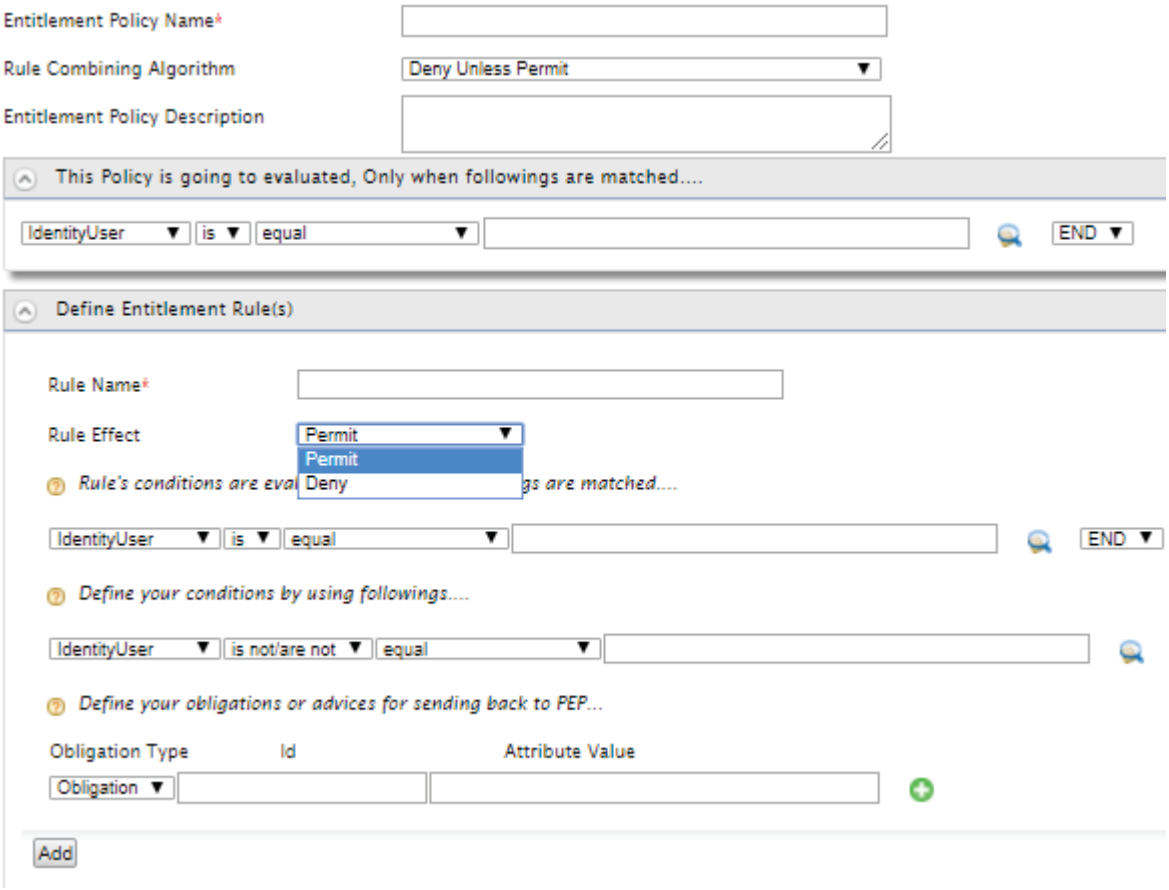

Figura 5: Formulario para la edición de una política XACML en un servidor de identidad.

2.4. Tecnologías base

2.4.1. RDF

Resource Description Framework es una familia de especificaciones del W3C que componen un modelo de datos para el intercambio de datos en la web [15], más particularmente en la web semántica. Su versión más actual es el RDF 1.1. ODRL está representado en RDF.

Este modelo se basa en expresiones que llaman “*triples*”, que están compuestas de un sujeto, un predicado, y un objeto. Con esta expresión podemos relacionar un sujeto y un objeto utilizando un predicado. Por ejemplo “Pedro escucha Mozart”, en RDF, “Pedro” sería el sujeto, “escucha” sería el predicado, y “Mozart” el objeto.

RDF está penado para que sea legible y utilizado por máquinas, y por lo tanto los formatos que adopta no están pensados en un primer momento para que sea comprensible por personas. Algunos de los formatos en los que se serializa son Turtle, N-Triples, N-Quads, N3, RDF/XML y JSON-LD.

2.4.2. JSON/JSON-LD

JSON (JavaScript Object Notation) es un formato de datos en texto, derivado de JavaScript. Consiste en pares clave-valor, y *arrays* de valores. JSON-LD (JavaScript Object Notation for Linked Data) es un formato para representar datos enlazados en JSON, y está regulado por el W3C. Su versión más actual aprobada por el RDF Working Group es la 1.0. [16]

La diferencia entre JSON y JSON-LD es que JSON-LD aporta algunas claves específicas (en los pares clave-valor) que le dan un significado particular al par clave-valor, y que habilita su uso en la web-semántica.

2.4.3. HTML

HyperText Markup Language es el lenguaje de marcado que se utiliza para generar las páginas web. Su estándar está regulado por el W3C [5], y su versión más actual es el HTML5.

Su función principal es etiquetar las diferentes partes de la página web, de tal manera que puedan ser identificadas y asignarles diferentes funciones y aspectos.

HTML nos proporciona la base sobre la que construir la vista de la aplicación, y nos aporta las funciones más básicas y necesarias como formularios, campos de texto donde introducir datos, y generar peticiones.

Las posibilidades de personalización de HTML son limitadas, por eso es necesario el uso de archivos CSS para asegurarse de que podemos darle exactamente la forma que queremos.

HTML por naturaleza es un lenguaje estático, por eso para que la vista mostrada pueda cambiar de forma, añadiendo y modificando trozos de HTML, sin generar una petición al servidor a cada vez, tenemos que usar JavaScript.

2.4.4. JavaScript / jQuery

JavaScript es un lenguaje de programación que puede ser ejecutado en tiempo real en los navegadores. Sus funciones son limitadas, así que es necesario usar jQuery, que es la librería de JavaScript más usada en navegadores, y además es *open-source*.
jQuery permite hacer acciones más avanzadas como encontrar etiquetas en el texto HTML, asignar funciones a eventos, modificar el estilo de la página, y añadir trozos de HTML a la página actual [6].

jQuery es la tecnología que nos va a permitir formar formularios dinámicamente, añadiendo trozos de HTML modulares a distintas partes del formulario, y a la vez identificándose con diferentes nombres para que el servidor pueda reconstruir la jerarquía original del formulario.

Además, jQuery contiene AJAX (Asynchronous JavaScript And XML), que es una técnica y un subconjunto de funciones que permiten hacer peticiones desde el cliente al servidor sin recargar la página. Esto nos sirve para obtener el resultado del procesado de la información del formulario (que hace el servidor) sin perder la información del formulario por tener que recargar la página, y mostrarlo directamente en la misma página.

2.4.5. CSS / Bootstrap

Las Cascading StyleSheets son un lenguaje de diseño gráfico que se usa para definir el estilo de lenguajes de marcado, principalmente el HTML. Su estándar está regulado también por el W3C, y su versión más actual es el CSS3.

Crear estas hojas de estilo desde cero es tedioso y muchas veces tienen partes comunes a todas las páginas web que siguen una forma más o menos estandarizada. Por esta razón tiene sentido usar Bootstrap, que es un *framework open-source* que proporciona principalmente estilos CSS (aunque también hace uso de JavaScript y otros scripts), que dan una base para el formato de muchos elementos de las paginas, así como soporte para otros eventos como formatear la página para que sea responsiva.

El uso de CSS en la aplicación será limitado, ya que en este caso no se trata de una página con muchas presentaciones y servicios distintos, y es más una simple herramienta, pero sigue siendo un elemento clave para que el usuario pueda comprender mejor de manera visual la estructura de la política que está generando.

2.4.6. Spring

Spring es un *framework* de aplicaciones basadas en *servlets*, *open-source*, que funciona con Java EE [9]. Aunque tiene otros usos, el principal es la de hacer de *back-end* de aplicaciones Web.

Spring tiene diferentes módulos para añadir diferentes características a una aplicación (seguridad, manejo de bases de datos, caches, etc.). Uno de los más interesantes es el Spring-core, que proporciona un servidor sobre el que desplegar tu aplicación, así como algunos otros módulos como Spring-web y Spring-MVC, que aportan las herramientas básicas para desarrollar la aplicación.

El funcionamiento básico de Spring es la asociación de rutas relativas (p.e. */índice/capítulos) a funciones de java, que suelen terminar con el envío de una vista en HTML al cliente, con la información que ha solicitado. Sobre este funcionamiento básico se van sumando otras funcionalidades más complejas como el paso de parámetros, el mapeo de la entrada recibida, diferentes métodos de input, etc.

Aparte del soporte para generar la aplicación web, la funcionalidad que más nos interesa es la capacidad de mapear la entrada de un formulario desde el cliente como clases POJO (Plain Old Java Object [8]) de java, a partir del nombre que se le ha dado a cada uno de los campos del formulario.

Por otra parte, Spring también habilita la posibilidad de generar un cliente REST que pueda atender a este tipo de peticiones. Esto es interesante porque habilita la posibilidad de que otros servicios hagan uso de la misma interfaz para usarla en otros servicios, y de atender a peticiones con AJAX, para no tener que recargar la página.

2.4.7. Jackson

Jackson es un "*parser*" entre clases de Java y JSON desarrollado por FasterXML [10]. Este *parser* convierte objetos java POJO a formato JSON, mostrando todos sus atributos y valores. Esto nos permite reutilizar las clases que utilizamos para introducir los datos del formulario con la licencia para generar automáticamente la política ODRL en JSON.

Para controlar el JSON formado por Jackson, este utiliza una serie de anotaciones que se pueden añadir a las clases Java, sin interferir con el mapeo que tiene que hacer Spring, de tal manera que ambas tecnologías son compatibles.

3. Desarrollo

En este capítulo vemos las cuatro partes principales del desarrollo, la creación del diseño funcional, donde a partir de las necesidades de la aplicación se crea la estructura y funciones que debe tener la interfaz, la selección de la arquitectura y tecnologías para la aplicación, donde comparamos diferentes opciones para escoger la que mejor se adapta al proyecto, la implementación de la aplicación, y el despliegue de la aplicación en Google Cloud.

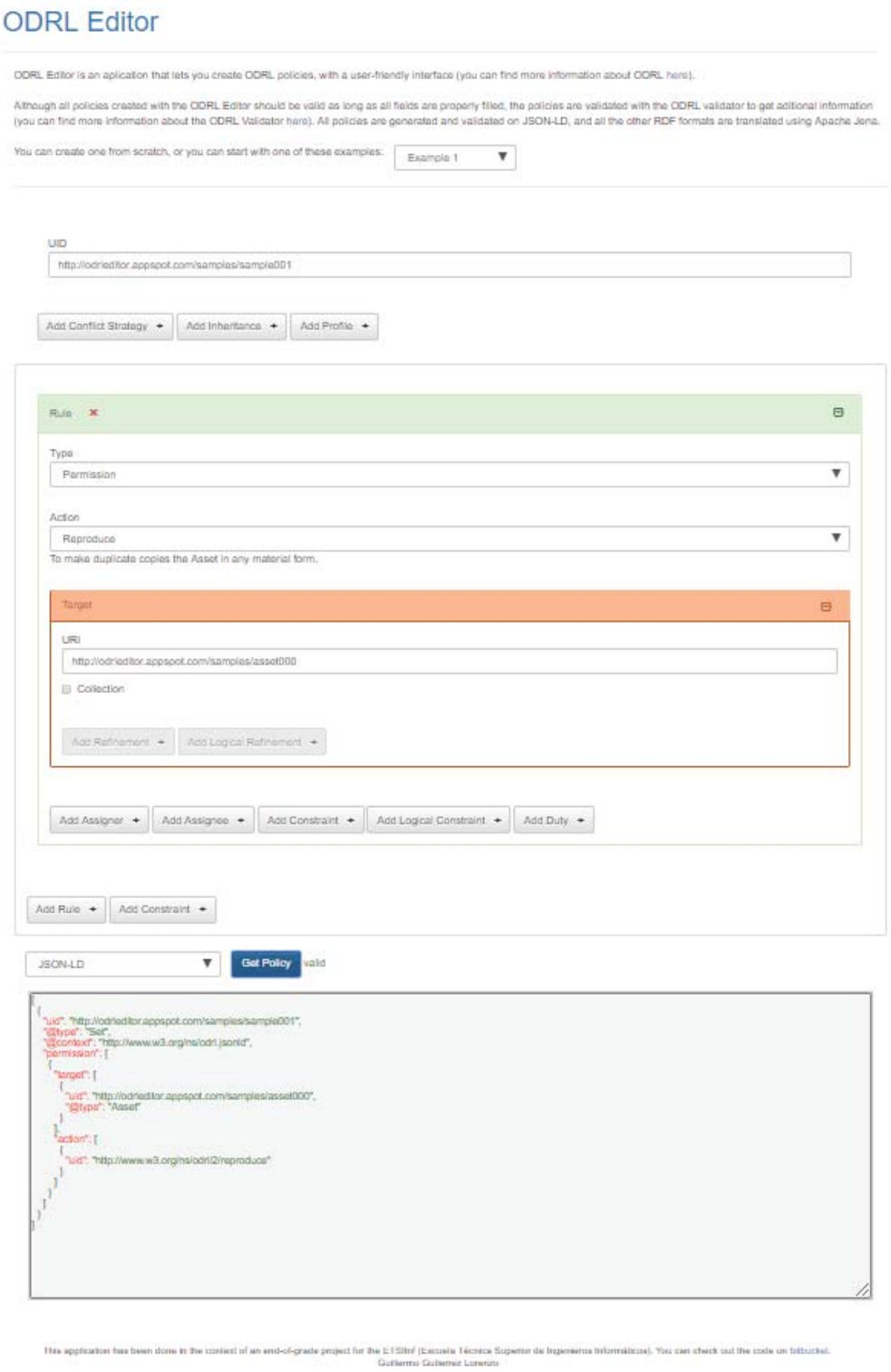

Figura 6: Vista final de la aplicación finalizada.

3.1. Problemática y creación del diseño funcional

Después de un primer estudio de ODRL, se pudo proceder a hacer una aproximación del diseño funcional de la aplicación. Hay **4 elementos principales**: cabecera, herramientas, política, y validación. El orden de estos 4 elementos en la aplicación sería este porque en un caso de uso idílico este sería el orden que sigues para obtener la política.

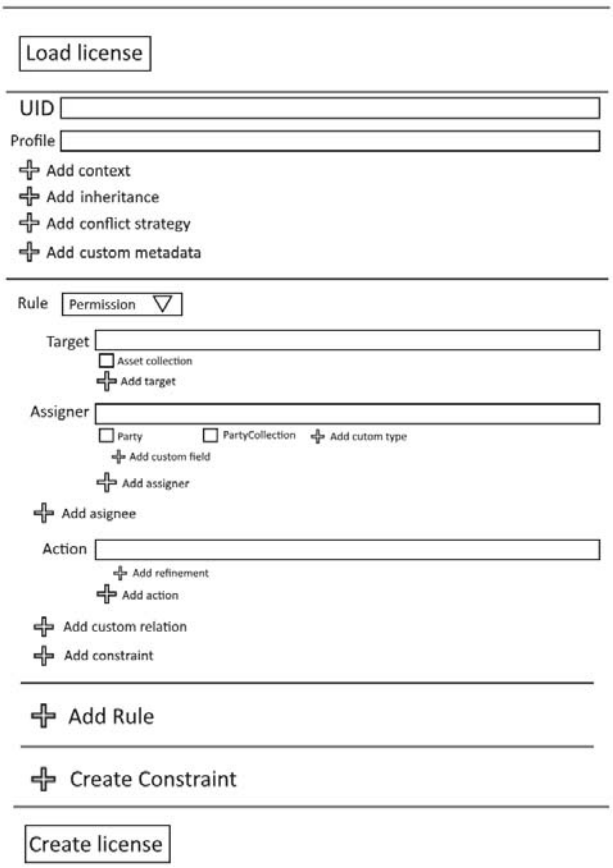

Figura 7: Diseño funcional de la aplicación.

Dentro de la política habría 3 zonas principales: datos relativos a la política (UID, contexto, estrategia de conflicto…), reglas y restricciones, y botonera para añadir elementos a la política.

La parte más compleja es la zona reglas y restricciones. La idea para las reglas era marcar la división de los elementos, y marcar la **estructura jerárquica** de los elementos, que en el diseño funcional se marcaba con una "tabulación" de los elementos. Los diferentes elementos debían estar agrupados (todos los cesionarios juntos, todos los objetivos

juntos…). Los botones que añaden elementos debían estar siempre en la zona del elemento jerárquico que les corresponde, y siempre debajo del elemento que añaden (el botón de añadir objetivo siempre debajo del último objetivo añadido).

El problema era que cuando ya había varios elementos en la política, buscar un botón para añadir un elemento específico entre otros muchos elementos era difícil y poco intuitivo. Se pasó entonces a reunir los botones siempre en la misma zona dentro de un elemento, y por lo tanto ya no tenía sentido reunir los elementos del mismo tipo juntos, puesto que cuando añades un elemento quieres verlo automáticamente, por lo tanto, tenía más sentido que el elemento saliese siempre detrás de todo el resto de elementos, junto a los botones.

Otro elemento que estaba bastante claro desde el diseño funcional, era que la aplicación no debía dejarte añadir elementos no permitidos (por ejemplo, un permiso dentro de una prohibición), y por lo tanto el formulario de la aplicación debía cambiar de forma según las opciones que has elegido. Por ejemplo, un permiso debe dejar añadir deberes, pero si cambiamos el tipo de la regla a una obligación, ya no podemos añadir deberes, sino consecuencias.

3.2. Arquitectura y tecnología para crear la aplicación

La aplicación no requiere de base de datos y, como veremos más adelante, la aplicación tampoco tiene estado, así que en caso de que haya varios servidores de carga, no es necesario que siempre te responda el mismo servidor.

Lo normal en un servidor que atiende a peticiones HTML normales es que conozca los datos que le pueden llegar. Por ejemplo, en un formulario de inscripción a una página sabe que le tiene que llegar nombre, apellidos, email y contraseña. Cuando un formulario cambia de forma, normalmente es de forma controlada y bastante limitada, y lo que se suele hacer es ocultar y mostrar campos de entrada HTML, aunque realmente siempre están presentes.

La aplicación crea formularios con distintas formas, y por lo tanto casi siempre es diferente. Por ejemplo, dentro de una regla puede haber una o ninguna restricción. Tampoco queremos limitar el número de elementos que se le pueden añadir a la política, y eso significa que virtualmente pueden ser infinitas (seguramente estaría limitado por el tamaño de la petición).

La primera opción que se planteó fue que a cada campo nuevo que se quisiese añadir el servidor atendiese a una petición nueva, de tal manera que el servidor conociese todos los campos de entrada, y además pudiese responder con el tipo de campo que se le debe añadir a la interfaz. Esta solución requiere de más programación de la lógica del sistema,

pero facilita el *front-end*. Surgen sin embargo dos problemas con este sistema: por una parte, el sistema tendría que atender a muchas más peticiones pequeñas, durante el proceso, más una grande al final, y por otra parte, el servidor tendría que ser un servidor con estado, puesto que debe recordar cuáles son los campos que ha ido añadiendo el usuario durante la sesión.

Este tipo de tecnología se planteó desarrollar con asp.net, un *framework* de Windows. Este *framework* tiene la ventaja de que la vista se genera a partir de un editor de vistas mucho más intuitivo.

La segunda opción (y la que se usó finalmente) era que toda la **generación** del **formulario** se hiciese en el ***front-end***, y que se hiciese de tal manera que los campos generados tuviesen una nomenclatura que pudiese ser procesada por el *back-end* de tal forma que no dependiera de conocer todos y cada uno de los campos que le van a llegar antes procesar la petición. Esta opción tiene la ventaja de que solo requiere de una petición al servidor, y que la respuesta de la interfaz cuando estás cambiando la forma del formulario es instantánea. Las desventajas son, por una parte, que, sin la tecnología adecuada, este sistema requeriría construir un *back-end* que fuese capaz de leer una nomenclatura muy particular, *parseando* todos los campos que han sido introducidos. Por otra parte, este modelo requiere más programación de *front-end* para que la construcción del formulario se haga toda del lado del cliente.

SpringMVC tiene un sistema para montar la información recibida en un formulario HTML normal como clases java, ayudándose de la nomenclatura de los campos. Con esta tecnología, mientras podamos crear un *front-end* sólido, el *back-end* sería muy flexible y reducido.

Para explicar el funcionamiento, ponemos el siguiente ejemplo:

En un formulario para empleados tienen que rellenar 2 campos, uno para su nombre y otro para su número de identificación. Esto se puede plasmar en una clase con un atributo de "id" de tipo *long* y otro atributo "nombre" de tipo *string*, tal que así:

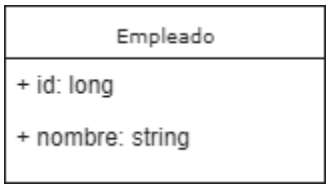

Figura 8: Modelo de la clase Empleado.

Al controlador que va a atender a la petición se le debe indicar que la entrada va a ser una clase con el *tag* “@ModelAttribute”, que va a ser de tipo empleado, y que el formulario va a tener un atributo “modelAttribute” con valor “empleado”:

```
public String submit(
      @ModelAttribute("empleado") Empleado empleado)
```

Figura 9: Método que atendería a un formulario con las propiedades de Empleado.

En la vista, tenemos que presentar un formulario que tenga un atributo “modelAttribute” con valor “empleado”, y mientras dentro de este formulario las entradas tengan un atributo “path” o un atributo “nombre” con los mismos nombres que los atributos de la clase Empleado, la entrada del controlador contendrá los datos del formulario. El formulario quedaría entonces como:

```
<form:form method="POST" action="/spring-mvc-java/addEmployee"
  modelAttribute="empleado">
    <form:label path="nombre">Name</form:label>
    <form:input path="nombre" />

    <form:label path="id">Id</form:label>
    <form:input path="id" />

    <input type="submit" value="Submit" />
</form:form>
```

Figura 10: Formulario HTML con las propiedades de Empleado.

Esta manera de comunicar al controlador como debe rellenar las clases desde el formulario va extendiéndose utilizando una nomenclatura muy parecida a la de instancias de objetos. Si por ejemplo dentro de la clase empleado hay una lista “apellidos”, en el formulario podemos rellenar esa lista indicando los campos como “apellidos[0]”, “apellidos[1]”, “apellidos[2]” etc. Y si por ejemplo dentro de la clase “Empleado” hay un otro empleado “jefe” también podemos acceder a los atributos del empleado “jefe” indicando los campos del formulario como “jefe.id” y “jefe.nombre”.

Adoptando esta solución, obtendríamos una arquitectura parecida a la de la figura 11, en la que estaríamos usando un patrón **modelo-vista-controlador** tanto en *front-end* como en *back-end*.

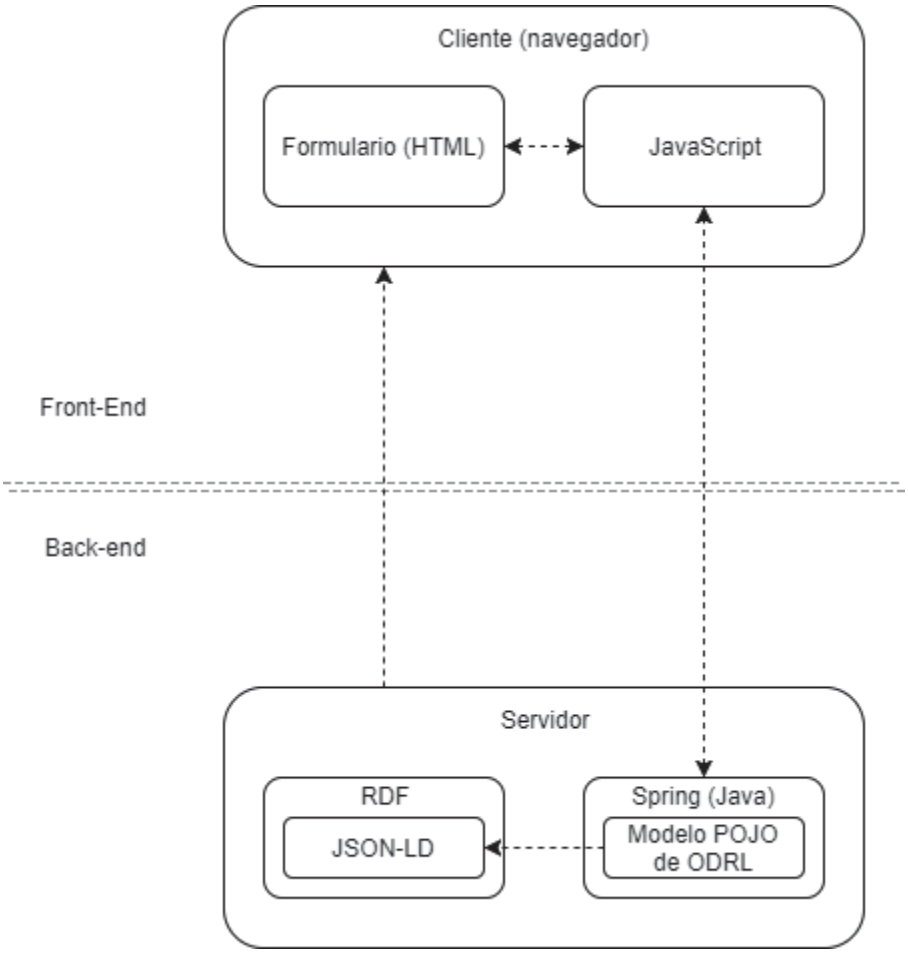

Figura 11: Esquema de la arquitectura de la aplicación.

3.3. Implementación

3.3.1. Estructura base de las funciones

La implementación del sistema se hizo gradualmente, y de manera paralela para *el front-end* y el *back-end*. La manera más intuitiva de abordar el problema era ir implementando la mayor parte posible de ODRL partiendo de la política más sencilla posible, e ir implementando poco a poco la capacidad de añadir más elementos a la política.

A lo largo de la implementación han ido surgiendo problemas a los que ha habido que buscarles una solución, de los cuales se presentan los más importantes.

Lo primero fue conseguir que desde el *front-end* se pudiese construir campos nuevos en un formulario. **jQuery** proporciona una función "*append*" con la que podemos insertar trozos

nuevos de HTML en el documento HTML que ya está siendo mostrado en la interfaz. La inserción siempre se hace dentro de un elemento HTML, y por lo tanto basta con indicarle en qué elemento lo quieres insertar.

Para seleccionar un elemento HTML desde jQuery existen 2 maneras principales:

-**Selectores**: es una función que selecciona elementos HTML comparando principalmente alguno de sus atributos o tipo con una cadena de caracteres. Por ejemplo el selector $("[value='item']") estaría buscando los elementos HTML con un atributo "value" con valor "item".

-Métodos de navegación jerárquica ("***traversing***"): son métodos que navegan por el documento HTML basándose en la estructura jerárquica de árbol. Por ejemplo, la función "children('div')" aplicada a un elemento buscaría todos los hijos directos de ese elemento que sean de tipo "div".

Conociendo estas condiciones, está claro que, para poder navegar por el documento HTML, y poderlo modificar adecuadamente, hay que mantener una **estructura concreta** y uniforme.

Para insertar nuevos trozos de HTML desde JavaScript podemos utilizar simplemente *strings*, pero para trozos grandes de HTML se vuelve rápidamente incómodo y poco usable. Una manera mejor es escribir estos trozos de HTML en el documento HTML de la página, pero fuera del cuerpo, dentro de elementos tipo *script* con un identificador. Con jQuery podemos entonces buscar estos trozos de código HTML con selectores, y después insertarlos dentro de los elementos que nos interesan.

Una de las funcionalidades de JavaScript es que se pueden crear funciones dentro de otras funciones, y esas funciones se pueden asignar a variables y pasar por parámetros. De esta manera es como jQuery asigna funciones a elementos HTML (botones en la interfaz, por ejemplo). Pongamos como ejemplo el siguiente código:

```
$("[name='prueba']").click(function(e) {
   console.log("funciona");
})
```

Figura 12: Función JavaScript que asigna una función a un evento de un elemento.

Para los elementos con un atributo "name" con valor "prueba", en cuanto ocurre el evento "click" se va a ejecutar una función que escribe en la consola un mensaje.

Gran parte de la política que se crea son **listas**: listas de reglas, listas de cesionarios, listas de cedentes etc. Para poder ir generando un índice que vaya marcando la posición en la lista que tienen que ocupar los campos del formulario, tenemos que crear variables que estén en el mismo contexto que la función que se ejecuta para añadir un nuevo elemento en el formulario, pero fuera de la función para que mantenga el valor durante la sesión de la aplicación:

```
var i =0;
$("[name='prueba']").click(function(e) {
   i++;
})
```

Figura 13: Función JavaScript con una variable fuera de la función, pero en el mismo contexto.

En este trozo de código cada vez que hagamos *click* en el elemento con atributo “name” y valor “prueba”, el valor de la variable “i” aumentará. Si **encapsulamos** este trozo de código dentro de una función, y que esta función pueda cambiar el elemento HTML al que se le está asignando la función, podemos crear diferentes contextos de estos “sumadores” con cada uno su contador de *clicks*, y asignarlos a diferentes elementos HTML:

```
function newCounter(newName){
   var i =0;
   $("[name='"+newName+"']").click(function(e) {
      i++;
   })
}
```

Figura 14: Encapsulación de la función.

Esta manera de asignar funciones es la base de muchos de los métodos que se crean para esta aplicación, puesto que cada elemento (dos reglas diferentes, por ejemplo) de la política debe tener su propio contexto, y tiene sus propios botones, que, aunque llamen a la misma función, tienen que estar en contextos diferentes.

3.3.2. Estructura HTML

Los elementos del documento HTML están organizados de manera muy ligada a cómo están organizados los contextos en el código JavaScript. Diferenciamos 2 tipos de elementos distintos: los **campos** y los **paneles**. Los paneles son un contexto nuevo, tienen una cabecera, botones, y otros elementos dentro. Los campos solo son campos del formulario.

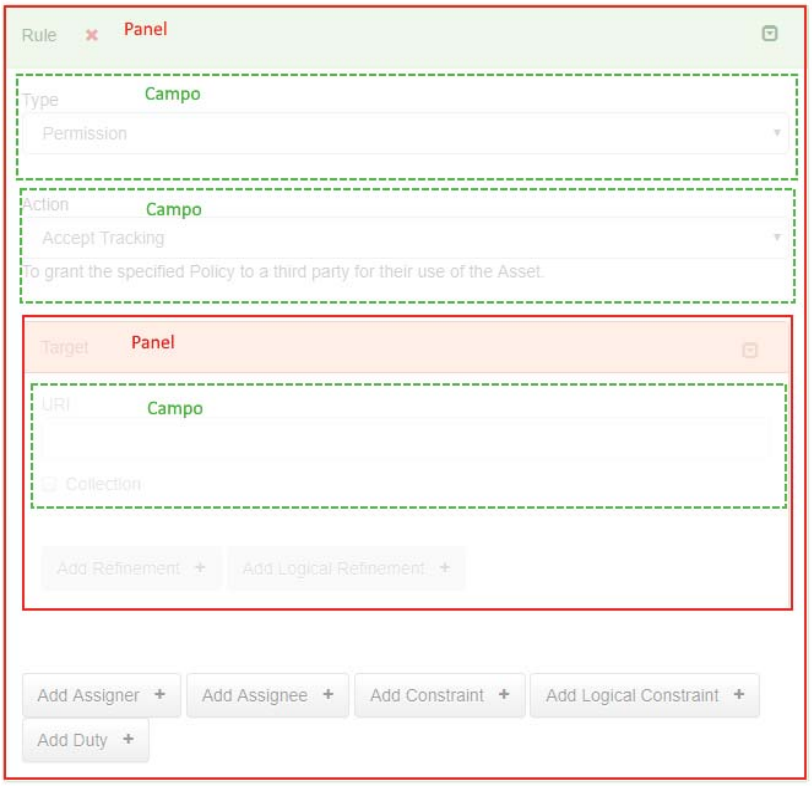

Figura 15: Esquema de los diferentes elementos de la interfaz

Los paneles definen un contexto nuevo, y por lo tanto tienen que llevar una información asociada en el documento a la que recurrir cuando hagamos nuevos cambios en el panel.

Para entender mejor como está construida la estructura del documento HTML, en la figura 16 está representado (de manera inexacta) la estructura del documento HTML sobre la interfaz final.

Figura 16: Esquema del documento HTML sobre la interfaz.

En la figura 16 vemos dos paneles distintos: uno en rojo y uno en verde. Se corresponden a una regla (en rojo) que, entre otras cosas, contiene un objetivo (en verde), aunque la naturaleza dentro de la política no nos interesa en este caso. Todos los **paneles** tienen la

misma estructura, como podemos ver en este caso con el panel marcado en rojo y el panel marcado en verde.

Tienen un contenedor de tipo div marcado como “panel-body”, este contenedor sirve para que la hoja de estilo (en este caso la de Bootstrap) los marque como elementos dentro del cuerpo del panel padre que lo contiene (no es relevante para los paneles raíz).

Después tienen otro contenedor de tipo div, marcado como “panel”. Este contenedor define un nuevo panel, este contenedor no lo tiene el otro tipo de elementos (los campos). Este tipo de contenedor sirve para que la Bootstrap encuentre dentro los elementos que crean el panel, pero también encontramos dentro información sobre el contexto, más concretamente el prefijo que van a tener todos los campos que estén dentro, marcado como “name”, y el numero de la lista (*array*) que ocupa marcado como “number”, si hace parte de una lista. Esta información es para poder navegar por los elementos, y para poder construir la información de los elementos hijos, pero nunca llega al servidor como tal.

Finalmente, dentro de este último contenedor, hay 3 contenedores más. Uno marcado como “panel header”, que marca a Bootstrap la parte que es la cabecera del panel, y que contiene el título del panel, el botón para colapsar el panel, y el botón de eliminar (si se puede eliminar). Los otros dos son el contenedor de los campos y paneles hijos (marcado como “fields”), y el contenedor de los botones del panel (marcado como “buttons”).

La estructura de los otros elementos, los campos, también empieza con un contenedor marcado como “panel-body”, pero luego el contenido es diferente según el tipo de campo, pero tampoco es tan relevante puesto que otros elementos (que no sean él mismo) no tienen que navegar por dentro, no como en los paneles.

Con esta estructura, podemos navegar por el documento HTML desde JavaScript y encontrar información actualizada sobre los elementos. Pongamos un caso de uso de esta estructura. El usuario quiere añadir un nuevo cesionario a la regla, y pulsa el botón de añadir un nuevo cesionario. Se ejecuta la función de añadir un nuevo cesionario que tiene asignada el botón de añadir un nuevo cesionario, y por lo tanto conocemos el elemento HTML que ha ejecutado esa función. Para nombrar correctamente el nuevo cesionario, sabemos que desde el contenedor “buttons” que lo contiene, el padre de este, que está marcado como “panel”, tiene un atributo “name” con el prefijo que va llevar el nuevo cesionario. A este prefijo tendrá que añadirle “assignees[X]” donde X es el numero de la lista que ocupa (esta información la tiene en el contexto JavaScript que corresponde al botón de añadir cesionario). Finalmente, sabe que el contenedor hermano “fields” es donde tiene que insertar el nuevo trozo de HTML que define el nuevo cesionario.

3.3.3. Eliminación de elementos

Una de las características del editor de políticas debe ser, lógicamente, que puedas borrar elementos que hayas añadido previamente. Para elementos que no pertenecen a una lista, es un procedimiento bastante sencillo, basta con eliminar el contenedor de la lista que lo contiene y volver a añadir el botón que deja añadir el elemento (si es necesario). El problema es eliminar un elemento de una **lista de elementos**. Esto no sería un problema relevante si limitásemos el borrado al último elemento añadido a la lista, pero esto es poco usable, tienes que poder eliminar cualquiera. El problema cuando dejas eliminar cualquier elemento de la lista es el mismo problema que cuando intentas eliminar un elemento de un array, tienes que **desplazar el índice** de todos los elementos que tenga por delante una posición hacia atrás.

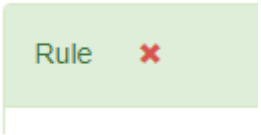

Figura 17: Ilustración del botón para eliminar elementos.

En el caso de la aplicación es más complejo, puesto que los campos que se encuentran en el formulario HTML tienen un nombrado que depende de este índice para que el servidor pueda mapearlos correctamente. Cuando eliminamos un elemento en la aplicación tenemos que cambiar el nombre de todos los elementos que iban por delante en la lista, y para cada uno de ellos el nombre de todos los elementos hijos que tienen. Pongamos un el caso de una regla que tiene 4 cesionarios y queremos eliminar el 2º de la lista, es decir el que tiene como nombre “rules[0].assignees[1]”, los elementos “rules[0].assignees[2]” y “rules[0].assignees[3]”, así como todos los hijos de estos 2 elementos, tienen que ser renombrados.

La función que elimina un elemento se ejecuta en el mismo contexto que el botón que añade los elementos, y por lo tanto tenemos una variable que nos indica el número total de elementos en la lista (en el ejemplo propuesto serían 4). Como hemos visto previamente, uno de los elementos que mantenemos en el documento HTML es el índice de la lista que ocupan los elementos que están en una lista, así que navegando por el documento podemos obtener el índice del elemento que queremos eliminar (en el ejemplo propuesto sería “1”). Otro de los elementos que podemos recuperar del panel antecesor del elemento que estamos borrando es el prefijo que lleva (en el ejemplo propuesto sería “rules[0]”), y que por lo tanto llevan todos los elementos de la lista que estamos manipulando.

Con estos 3 elementos (el **índice**, el **número total**, y el **prefijo**) podemos encontrar todos los elementos que están en la lista por delante del elemento que queremos eliminar, pero nos faltan los elementos hijo de esos elementos de la lista. Conocemos el nombre del elemento que queremos renombrar, así que tenemos que encontrar todos los elementos que empiecen igual, en el ejemplo anterior serían los elementos que empiezan por "rules[0].assignees[2]" y "rules[0].assignees[3]" seguido por un punto y cualquier cadena de caracteres. Esto se puede hacer con un selector que proporciona jQuery que compara el principio del atributo con la cadena de caracteres elegida. Este selector devuelve una lista de elementos (los que tenemos que renombrar), y para cada uno de ellos podemos ejecutar una función que cambia el prefijo del atributo nombre dejando el resto intacto. Finalmente basta con actualizar la variable que mantiene el número de elementos total en el contexto.

3.3.4. Colapso de campos

Cuando el editor ya admitía varios elementos anidados, surgió un problema que era más difícil de ver durante el diseño funcional: al perder de vista las cabeceras se hacía más **difícil entender** qué botones correspondían a que panel.

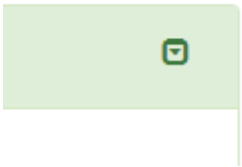

Figura 18: Ilustración del botón para colapsar elementos.

Como solución a este problema se introdujeron los botones para colapsar los paneles, esto permite ocultar paneles y así reducir el tamaño del panel padre. Por ejemplo, si estás trabajando en una regla y en un momento introduces una restricción, cuando terminas la restricción puedes colapsarla y seguir trabajando en la regla de manera más sencilla.

El colapso de paneles es una función que introduce Bootstrap para los paneles de forma opcional. Para ello tienes que crear un botón con un **atributo "data-target"** que debe contener el nombre del atributo y el valor del atributo que tiene que buscar para encontrar los elementos que colapsa el botón. El atributo que nosotros introducimos es uno que se llama "pc".

Como tiene que ser único para cada panel, lo más lógico es que el valor sea también el mismo que el del nombre del panel. Este atributo lo lleva el contenedor de los campos y el contenedor de los botones, para que desaparezcan cuando se colapsa el panel. Como el

valor es el mismo que el del nombre, cuando borramos un elemento que estaba en una lista, hay que actualizar el valor de estos atributos, al igual que con el nombre del panel.

3.3.5. Eliminación de elementos desde contextos diferentes

Hay elementos que por cambiar sus características admiten elementos hijos o no. Por ejemplo, solo cuando un objetivo está marcado como una colección admite refinamientos. Además, también existe la posibilidad de que, por cambiar las características del padre, los elementos hijo admitan a su vez elementos hijos o no.

Es el caso de las reglas. Un permiso (“Permission”) admite dentro deberes (“Duties”), y estos deberes admiten dentro consecuencias (“Consequences”). Sin embargo, una prohibición (“Prohibition”) admite dentro remedios (“Remedy”), pero los remedios no admiten otras reglas dentro. Esto implica que, si en el usuario ha creado un permiso que tiene un deber y dentro una consecuencia, y decide cambiar el tipo de la regla de un permiso a una prohibición, los deberes deben cambiar su tipo a remedio, y las consecuencias deben desaparecer.

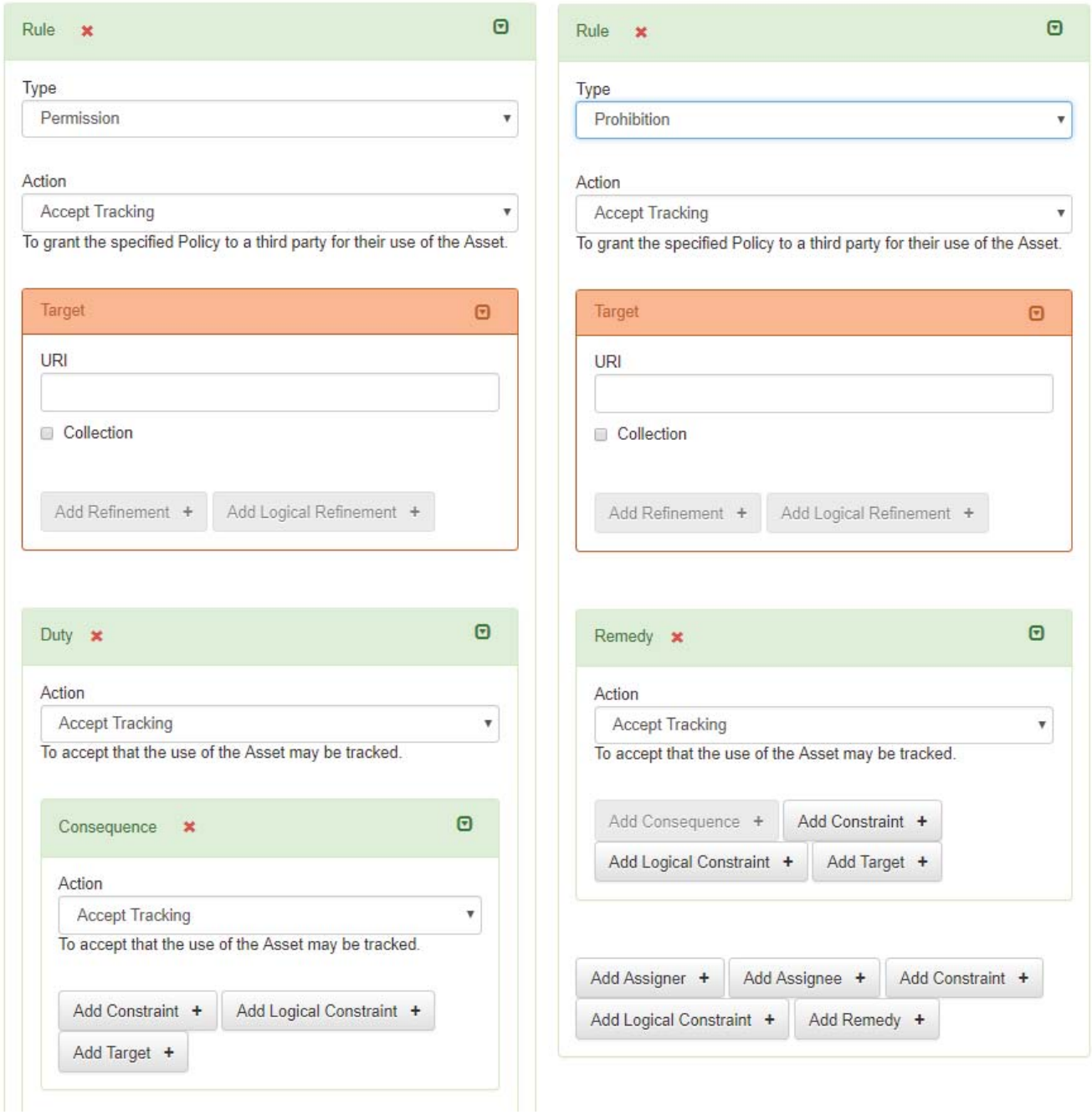

Figura 19: Cambio de un permiso (izquierda) a una prohibición (derecha).

Al ser el padre al que se le está cambiando las características, esto implica también que es el contexto del padre el que se tiene que encargar de eliminar los elementos. Por lo tanto,

ya no es el contexto desde el que se había añadido el elemento el que también se ocupa de eliminarlo, si no el contexto del elemento padre.

Desde el elemento padre podemos encontrar en el documento HTML los elementos hijo que queremos borrar fácilmente, el problema es que no podemos acceder a las variables que mantiene el número total de los elementos en las listas puesto que están contextos diferentes.

Para poder **acceder** a estas variables el contexto del padre tiene que mantener las **referencias** en su contexto. JavaScript no puede pasar enteros como referencia, así que lo encapsulamos dentro de una variable que hace de objeto contenedor, de tal manera que si accedemos al entero a través de esta variable sí que estamos accediendo por referencia.

Para que el contexto del elemento padre tenga todas estas variables, y facilitar el mantenimiento de estas variables, creamos una **función de suscripción** en el contexto del elemento padre que guarda la variable, y así es el que crea la variable del número total de elementos en la lista el que se encarga de comunicarse con el padre y pasarle la referencia. Esta función se le puede pasar a los elementos hijo por parámetro cuando es creado. Ahora cuando las características del padre cambien simplemente tiene que recorrer la lista de variables suscritas y resetearlas si es necesario.

3.3.6. Población de listas desplegables

Las políticas ODRL están pensadas para que el usuario pueda rellenarlas como más le interese con los valores que más le convengan. Sin embargo, para algunos campos lo normal, y lo que está recomendado, es que se rellenen con **URIs estándar**, como es el caso de las acciones, los operadores, los operandos izquierdos etc. Estos valores estándar también están definidos por el modelo ODRL, y para facilitarle al usuario su uso, lo lógico es que estos campos sean listas desplegables con todos los campos definidos por el W3C. De esta manera el usuario solo ve el valor (por ejemplo “imprimir” o “copiar”) y solo en la política final aparece la URI que lo representa. Además de esta manera podríamos darle al usuario una definición del valor elegido al escogerlo, como información adicional.

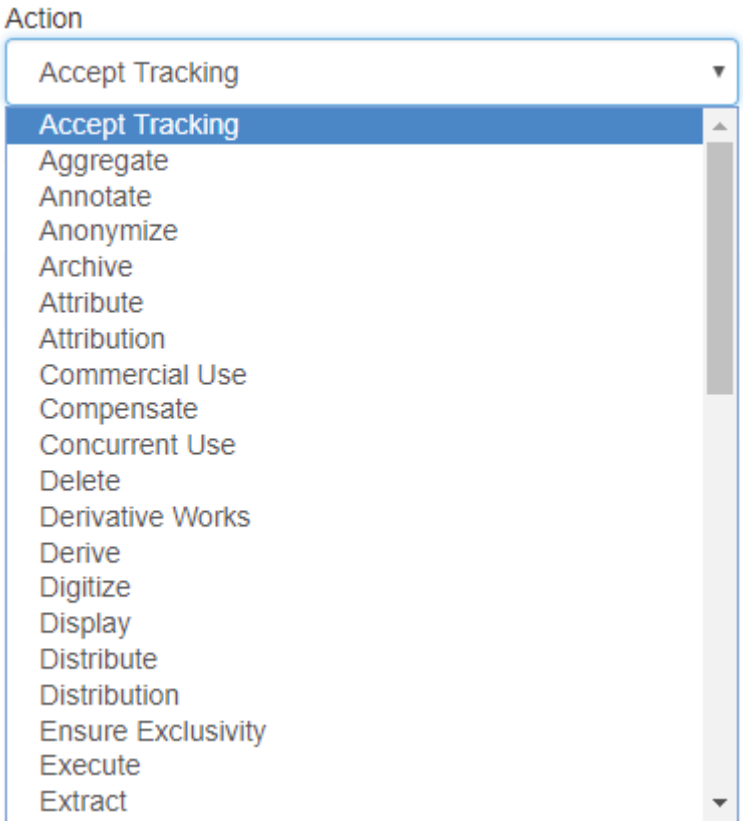

Figura 20: Lista desplegable de una acción.

El problema es que introducir todos estos valores a mano implicaría ir uno a uno por todos los valores disponibles, y añadir en el documento HTML su nombre, su URI, y su definición, lo cual es tedioso y poco mantenible.

W3C proporciona en https://www.w3.org/ns/odrl/2/ODRL22.json un JSON con todo el **vocabulario** asociado a las políticas ODRL, entre el que se incluyen los valores que necesitamos. Desde el *back-end* podemos recuperar este JSON directamente de la dirección que proporciona W3C. Una vez tenemos el archivo JSON podemos ayudarnos de Jackson para mapearlo en objetos Java, pero para ello antes debemos crear las clases Java que lo representan. Crear las clases que lo representan a mano es muy complejo, así que nos ayudamos de un **generador** (en este caso hemos utilizado el disponible en http://www.jsonschema2pojo.org/) que crea las clases a partir del JSON.

Una vez hemos podido mapear el JSON como objetos java podemos recorrer el JSON como listas de valores, y buscar los valores que nos interesan buscando que su tipo

coincida. Si es el tipo que nos interesa, podemos guardar el objeto aparte, conteniendo el valor, la URI, y su definición.

Ahora que tenemos los valores en el *back-end* tenemos que meterlos en la **vista** que se le manda al usuario cuando carga la página. Para ello nos ayudamos de **Thymeleaf**, una tecnología que se usa mucho con Spring, y que crea partes del documento HTML que se le manda al usuario cuando pide una vista.

El uso de Thymeleaf se hace a través de notaciones especiales en el documento HTML. Thymeleaf busca atributos que tengan el prefijo “th:” dentro del documento HTML y trata su contenido de una manera distinta, más parecida a código dirigido a programación con objetos. A Thymeleaf podemos pasarle objetos java desde el *back-end*, y utilizar estos objetos en las notaciones especiales Thymeleaf dentro del documento HTML. De esta manera podemos pasarle una lista de objetos a Thymeleaf, y que él los introduzca en el documento HTML para que cree una lista de opciones dentro de una lista desplegable, como nos interesa en este caso.

```
<option  th:each="action : ${actions}" th:value="${action.uri}"
th:text="${action.name}"
th:attr="description=${action.description}"></option>
```

Figura 21: HTML de una opción con anotaciones Thymeleaf.

En el elemento HTML de la figura 21 le estamos indicando a Thymeleaf que haga un “for each” con la lista “actions” que le llega desde *back-end*, y que para cada elemento de la lista cree un nuevo elemento HTML “option”, en el que el atributo “value” tenga el valor de la URI de la acción, el texto mostrado sea el nombre de la acción, y que le añada un atributo “descripción” con la descripción de la acción.

3.3.7. Generación de la política con Jackson

Cuando el cliente consigue mandar una petición de política con el formulario correctamente, en *el back-end* del servidor obtenemos directamente una **política mapeada** en objetos java según como hayamos recreado el modelo ODRL en clases JAVA. En este caso la representación del modelo ODRL es muy parecido a lo que nos encontramos en la figura 22 y se ha ido construyendo según se han ido añadiendo características a la aplicación. A partir de estos objetos necesitamos crear una política ODRL en JSON-LD.

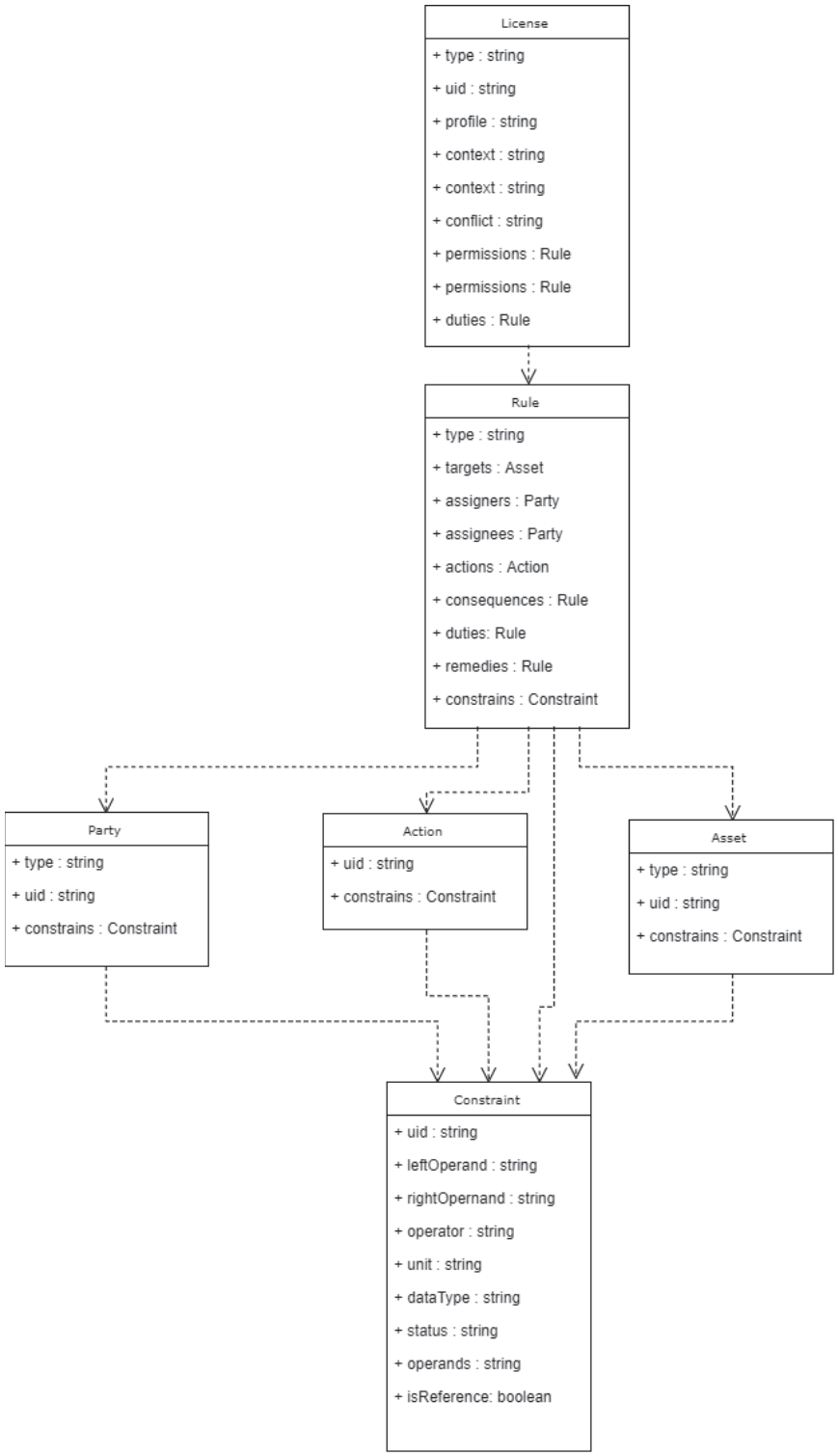

Figura 22: Modelo UML simplificado de la modelización de ODRL en el proyecto.

Lo primero es refinar algunos de los objetos mapeados simplemente con la ejecución de una función del objeto. Es por ejemplo el caso de las reglas, llegan todas en una lista “rules”, con los permisos, obligaciones, y prohibiciones mezcladas. Esto es porque resulta mucho más fácil unificarlo en una sola lista en el *front-end*, y ahora en el *back-end* simplemente tiene que repasar la lista “rules” sacando las reglas y repartiéndolas en 3 listas diferentes según su tipo.

Jackson por defecto crea el JSON asignando a los pares clave-valor el mismo nombre que la variable que está representando, y el valor asignado sea cual sea (incluyendo “null”). Para poder controlar el JSON de salida Jackson proporciona 2 herramientas: las **etiquetas** y los **serializadores**.

Las **etiquetas** son anotaciones Java con el prefijo “@”, y con ellas podemos hacer gran parte del formateo final del JSON **fácilmente** como: indicarle a la clase que ignore los campos nulos, indicarle que variables deben ser ignoradas, que variables tienen que tener un nombre distinto etc. Sin embargo, con las etiquetas encontramos limitaciones cuando la salida está condicionada. Es el caso por ejemplo de los objetivos (“targets”): el campo “uid” debe pasar a llamarse “source” en caso de que el objetivo sea una colección refinada, y este tipo de condicionamiento no lo podemos representar con etiquetas.

Los **serializadores** son clases que implementan métodos que permiten controlar la serialización del objeto java en formato JSON de manera mucho más **flexible** y compleja. Cada serializador que creas se aplica a una clase en concreto de las que quieres serializar, y solo sirve para esa. En la clase a la que quieres aplicar el serializador tienes que indicar que serializador va a utilizar (si no es el de por defecto). Dentro del objeto del serializador obtienes el objeto instanciado que quieres serializar, y por lo tanto puedes controlar la salida en función de sus campos. Para indicarle como tiene que ser la salida al serializador se utilizan métodos que indican cuando quieres añadir un nuevo campo, que nombre y valor va a llevar, cuando inicias un array, cuando terminas el array etc.

Los “asset”, “party”, y “constraint”, tienen serializadores aparte, el resto con el serializador por defecto y las etiquetas pueden reconstruirse adecuadamente en JSON. Con estos recursos podemos serializar la política y las restricciones adicionales, y las obtenemos en un *string* listo para ser mandado de vuelta al cliente.

3.3.8. Petición de la política con AJAX

Cuando un usuario manda un formulario, lo normal es que el servidor le mande una vista nueva con la información mandada procesada. En este caso sería mostrarle de nuevo el formulario con la nueva política al final, pero esto supondría perder toda la información introducida en el formulario, puesto que estamos recargando la página.

Para evitar recargar la página, podemos enviar la información desde JavaScript, en vez de que se encargue el navegador directamente. Para ello utilizamos **AJAX**, un subconjunto de funciones que permite enviar y recibir información con jQuery. A AJAX solo hay que indicarle que método REST tiene que usar, a que dirección, los datos, la función a ejecutar cuando reciba los datos correctamente, y la función a ejecutar en caso de error.

Desde el lado del servidor necesitamos entonces un **controlador** que atienda a este tipo de peticiones. Spring habilita esta opción cuando le añadimos una anotación especial a la clase del controlador. La respuesta que manda el servidor ya no es un documento HTML, si no un mensaje en JSON que contiene (entre otras cosas como veremos más tarde) la política en JSON. Este mensaje lo podemos crear con una clase marcada con unas notaciones especiales de Jackson.

En la función AJAX del cliente, cuando se reciben los datos correctamente, el método que se ejecuta recibe como parámetro el JSON mandado desde el *back-end*, y desde ahí podemos recoger la política como un *string*, y añadirla a un espacio al final de la página.

Como función añadida, cuando el formato de la política pedida es JSON, para mostrar la política se parsea antes etiquetando los elementos con etiquetas HTML para mostrar la política formateada y con colores. Estos parsers son muy comunes y se pueden encontrar de varias fuentes, en este casi hemos utilizado el ofrecido por un usuario en Stack Overflow [18].

3.3.9. Validación de las políticas

Durante todo el desarrollo para comprobar que las políticas generadas eran correctas se utilizaba el validador de políticas hecho por Víctor Rodríguez Doncel, y disponible en http://odrlapi.appspot.com/. Esta aplicación tiene disponible además una **API REST** con la que podemos utilizar el validador. Aunque no es posible hacer una política invalida con nuestra aplicación mientras estén todos los campos adecuadamente rellenados, validarla con el validador indicaría los casos en los que falta algún campo, y además siempre da información extra sobre buenas prácticas, aunque sea válida.

Como cliente para comunicarse con la API del validador usamos el **cliente HTTP** de Apache. Al cliente solamente le tenemos que indicar la URL, las cabeceras indicando el formato del contenido, y el contenido (es decir la política en JSON-LD). El cliente obtiene la respuesta de forma **síncrona**, lo que implica que añade un pequeño retraso a la respuesta. La respuesta está en formato JSON también, y por lo tanto podemos recuperarla en un mapa de pares clave-valor con Jackson. De ahí obtenemos un booleano que indica si la

política es válida, y un texto explicando si es válida o invalida, e información sobre buenas prácticas o porque es invalida.

Estos dos datos los encapsulamos junto con la política en la respuesta en JSON que enviamos al cliente AJAX del usuario. En el cliente obtenemos los valores igual que con la política, mostramos el texto que explica si la política es válida encima del recuadro de la política, y con el booleano sabemos si el texto tiene que estar en rojo o en verde.

3.3.10. Traducción de las políticas

RDF se puede expresar en varios formatos diferentes, así que resulta conveniente que el editor deje obtener la política en otros formatos además de JSON-LD.

Al tener ya la política en JSON-LD, lo más fácil es traducir este formato a otros formatos. Hay aplicaciones online que permiten hacer esta traducción, así que la primera opción fue buscar una API REST en alguna de estas aplicaciones y utilizar un cliente HTTP igual que el del validador para obtener otros formatos.

De esta manera la implementación fue rápida, pero surgían dos inconvenientes principalmente. Por una parte, se aumentaba el tiempo de respuesta del editor de políticas, puesto que necesitaba la respuesta de otro servidor, además de depender de otro servicio externo. Por otra parte, y más importante, las peticiones a la aplicación utilizada para traducir estaban limitadas diariamente por cliente, y a partir de unas 20 peticiones el servidor dejaba de dar servicio.

Esta opción quedaba entonces descartada, y se pasó a buscar una opción que estuviese implementada de forma nativa. Tras pasar por otras opciones sin éxito, dio resultado **Apache Jena**, un *framework open-source* de Apache para web semántica y RDF.

Para traducir desde JSON-LD lo que hacemos es cargar leer y cargar la política en RDF en un modelo, lo cual nos permite después exportarlo a los formatos permitidos que son: Turtle, N-Triples, RDF/XML, N3 (Notation 3), y RDF/JSON (este, a diferencia de JSON-LD, no admite claves con significado y necesita explicitar todas las propiedades a cada elemento).

3.4. Despliegue

Los componentes Spring elegidos permiten bastante flexibilidad a la hora de desplegar. Según como está configurado se puede exportar en un ejecutable Java, o en un paquete WAR que se puede añadir a un servidor de aplicaciones Java EE tipo ***servlets*** (Tomcat, Glassfish, etc.). Un *servlet* es una aplicación Java que normalmente atiende a peticiones

web dentro de un servidor. Puede haber *servlets* diferentes atendiendo a diferentes peticiones dentro de un mismo servidor [17].

En este caso lo queremos desplegar específicamente en **Google Cloud**, y podemos implementar su solución de despliegue directamente, que es un caso de despliegue con un paquete WAR. Para configurar el *back-end* simplemente hay que indicar que la aplicación se configure cuando lo arranque el contenedor de *servlets*, sobrescribiendo un método de Spring. En el proyecto se traduce en dos dependencias de Maven nuevas, las herramientas de Google Cloud, y la API de App Engine. Además, hay que añadir un XML de configuración que indica algunos parámetros de la aplicación en Google Cloud, como el nombre o el número de versión, y otro archivo para indicarle que tipo de entorno requiere el *servlet* en particular.

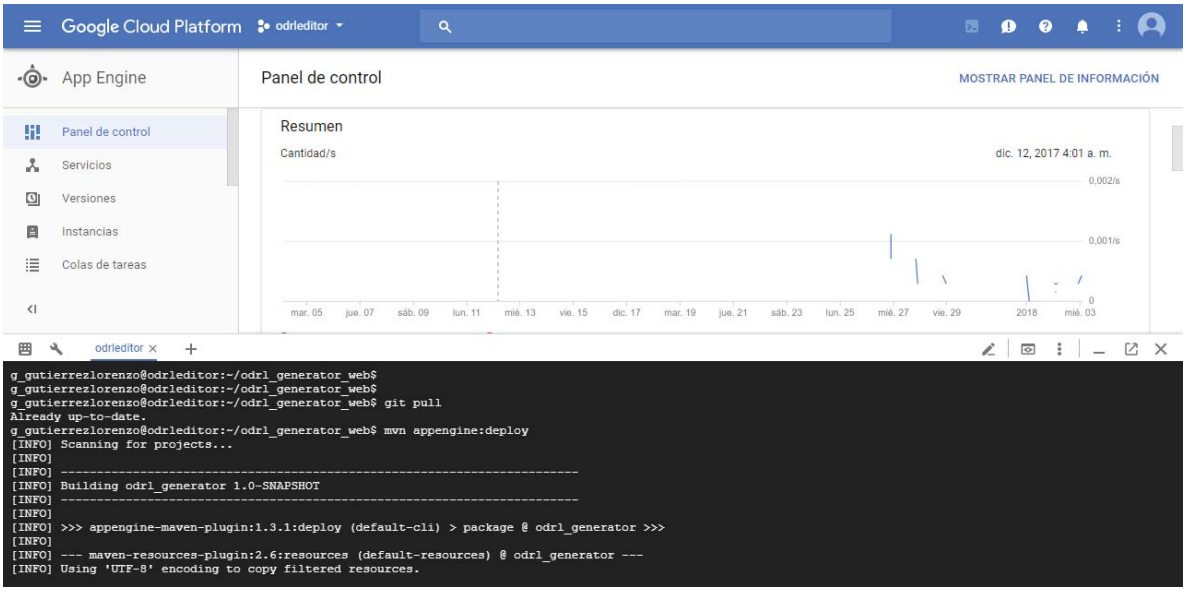

Figura 23: Consola y terminal SSH de Google Cloud.

La configuración del proyecto de Google Cloud es bastante sencilla, y en ella eliges algunos datos importantes como el id del proyecto (que define la URL de la aplicación) y la región donde se debe situar el servidor. Cuando ya está configurado puedes acceder a un ***shell*** que conecta con el servidor, el cual ya viene con algunas herramientas necesarias como las de despliegue en App Engine, Maven, o Git. Cuando ya está configurado el servidor, puedes descargarte el proyecto con Git, y desplegar con las herramientas de App Engine.

4. Resultados y conclusiones

En este capítulo hacemos una recapitulación del trabajo realizado, comprendiendo los límites de la aplicación, el proceso de decisión que llevó a no implementar la carga de políticas, y la conclusión del trabajo.

4.1. Conclusión

Se ha creado una aplicación web para la **edición** de políticas ODRL que cubre una gran parte de los elementos que pueden componer una política ODRL. La aplicación permite componer políticas relativamente **complejas**, y con una seguridad de que la política obtenida es teóricamente **valida**.

La aplicación permite a un usuario que no necesita automatizar la generación de políticas crear una de manera rápida, y en diferentes formatos. Como mínimo, puede generar una política base bastante consistente, a la cual se le pueden añadir “manualmente” elementos después de generarla, si falta alguno.

En cuanto al resultado final del trabajo, se echa en falta el objetivo de la carga de políticas, que resultó ser un objetivo mucho más costoso de lo que parecía inicialmente. En contrapartida, se han añadido algunas funcionalidades que no se consideraban (o se consideraban adicionalmente) que han aportado mucho a la aplicación. Estas funcionalidades son: el **despliegue** de la aplicación en Google Cloud, la **traducción** de las políticas a otros formatos, y la **validación** de las políticas con la API del Validador ODRL.

Gran parte del trabajo poco visible tras el desarrollo del editor es ha sido la familiarización con las políticas ODRL y su funcionamiento, y el gran desafío de la implementación ha sido la **generación dinámica** de un formulario HTML desde el *front-end*. Al final para el funcionamiento del sistema se han utilizado muchas **tecnologías** y herramientas diferentes que ha habido que estudiar para poder aprovecharlas adecuadamente, pero que se han adaptado muy bien a las necesidades.

Personalmente me ha resultado muy interesante poder colaborar en un proyecto *open-source* que trata de facilitar la creación de unas políticas que tratan de proteger el contenido de un autor.

4.2. Límites de las políticas generadas

Las **acciones** no se pueden **refinar** con el editor. Cuando se refina una acción, el valor de la acción debe estar encapsulado dentro de una etiqueta “rdf:value”. El problema es que al encapsular el valor el validador no reconoce la URI del valor, y por lo tanto no queda claro si la acción está siendo reconocida correctamente.

Las restricciones lógicas no incluyen el “**andSequence**”, que es un operador “and” en el que hay que respetar el orden de la lista. Esto es debido a que la notación de una lista ordenada en JSON-LD requiere que la lista tenga una clave “@list” delante, y en Turtle está anotada entre paréntesis. Estas notaciones parece que no están reconocidas por el validador, y por lo tanto tampoco queda claro si la política es correcta, puesto que tampoco existe un ejemplo con este operador en el modelo.

El atributo de herencia “**inheritFrom**” no se puede añadir a la política. El problema con la herencia es que requiere leer la política padre para poder completar la política. Algunos valores, como el del tipo de la política, están ocultos en la aplicación puesto que se pueden deducir de los campos que ha elegido el usuario. Si estuviese permitido el campo “inheritFrom” tendríamos que poder leer las políticas heredadas para poder componer esos valores correctamente.

No se pueden añadir **metadatos**, como por ejemplo el par clave valor: ["vcard:fn": "Sony Books LCC"]. Este tipo de metadatos requieren conocer el **vocabulario** de su **contexto** (en el ejemplo seria “vcard”). Una solución para poder añadir estos metadatos sería dejar que el usuario pueda introducir campos clave-valor arbitrarios, pero esto supone dos inconvenientes. Por una parte, es poco usable y no le estamos facilitando nada al usuario, hay muy poca diferencia entre añadirlo en el formulario de esta forma, e introducirlo directamente en la política en JSON-LD, solo le estaríamos introduciendo complejidad a la aplicación. Por otra parte, no podríamos admitir que las políticas creadas son correctas si no se hiciese algún tipo de validación sobre estos metadatos.

Como no se pueden añadir metadatos, tampoco se pueden añadir **contextos**, puesto que principalmente los contextos se utilizan para poder introducir los metadatos en la política.

4.3. Carga de políticas

El **objetivo no cumplido** es la carga de políticas en la aplicación. Durante el desarrollo se priorizó tener el editor lo más terminado posible, y por la dificultad que implicaba la carga de políticas se decidió que no tenía sentido intentar implementarlo.

Por una parte, la carga de políticas requiere que la aplicación pueda leer y modelizar políticas. Esto implica tener un parseador que pueda ir convirtiendo desde algún formato de RDF a los objetos que modelizan las políticas en nuestra aplicación, o a partir de un modelo RDF (como el que utilizamos para la traducción de políticas) poder crear los objetos que modelizan las políticas. Cualesquiera de las dos opciones suponen una dificultad bastante grande puesto que las políticas obtienen muchas formas distintas, y atienden a muchos casos particulares para cada tipo de elemento de la política.

Por otra parte, está la recreación del formulario de la política desde el servidor. El formulario se crea desde cero en el *front-end*, y el *back-end* no está implicado en ningún momento. El usuario va creando el formulario poco a poco según va ejecutando las funciones de JavaScript, lo que significa que no hay una traducción directa entre la política (o el modelo de la política) y como formar el HTML. Intentar implementar un sistema como este supondría que en el *back-end* tendríamos que tener una manera de traducir cada uno de los elementos de la política en elementos HTML, y además ser capaces de preparar el **estado** de los contextos en JavaScript (el valor de todas las variables necesarias) para que el cliente pueda modificar la política desde el estado en el que lo ha dejado el *back-end*.

Cuando se decidió que era muy difícil implementar la carga de políticas fue cuando se añadieron los ejemplos que se pueden cargar directamente desde la aplicación. Estos ejemplos se crean simulando los eventos que normalmente saltan por la interacción del usuario con la interfaz con una secuencia, y luego se rellenan los campos creados sabiendo cuales son los campos que existen y como están nombrados.

5. Trabajos futuros

Entre las mejoras que ser interesantes para el editor de políticas ODRL, sin duda la que más urge es la integración de un sistema de **carga** de políticas, para que el usuario pueda modificar políticas ya existentes en algún formato RDF.

En cuanto a la implementación del modelo, quizás lo que más pueda aportar a los usuarios sea la implementación y adopción de otros **contextos**, para que puedan utilizar vocabulario de otros contextos (como vcard o dcterms). La integración de la **herencia** de políticas sería igualmente interesante si el sistema fuese capaz de adoptar las condiciones que implican heredar elementos de otras políticas.

Aprovechando que la idea de estas políticas es la automatización de su uso, quizás una funcionalidad interesante para el editor sería que pudiese **automatizar** parte de la generación de políticas a partir de **colecciones** de recursos. Un ejemplo de esto sería que el usuario pudiese cargar una lista de cesionarios, y que el editor fuese capaz de añadir todos a la política directamente.

6. Bibliografía

[1] "Comunidad de W3C sobre ODRL (W3C ODRL Community Group)" [En línea]: https://www.w3.org/community/odrl/

[2] "Página de Wikipedia de ODRL" [En línea]: https://en.wikipedia.org/wiki/ODRL

[3] "Modelo ODRL" [En línea]: https://www.w3.org/TR/odrl-model/

[4] "Vocabulario y Expresiones ODRL" [En línea]: https://www.w3.org/TR/vocab-odrl/

[5] "HTML en W3C" [En línea]: https://www.w3.org/html/

[6] "API de jQuery" [En línea]: https://api.jquery.com/

[7] "Introducción a Bootstrap" [En línea]: https://getbootstrap.com/docs/4.0/getting-started/

[8] "Explicación de POJO según Spring" [En línea]: https://spring.io/understanding/POJO

[9] "Introducción a Spring Framework" [En línea]: https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/overview.html

[10] "GitHub de Jackson Project" [En línea]: https://github.com/FasterXML/jackson

[11] "Implementaciones de ODRL en W3C" [En línea]: https://www.w3.org/community/odrl/implementations/

[12] Víctor Rodríguez Doncel, "Validador ODRL" [En línea]: http://odrlapi.appspot.com/

[13] Víctor Rodríguez Doncel, "Repositorio del Validador ODRL" [En línea]: https://github.com/oeg-upm/licensius/tree/master/odrlapi

[14] "Página de URI en Wikipedia" [En línea]: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

[15] "RDF en la wiki del W3C" [En línea]: https://www.w3.org/RDF/

[16] “Página web de JSON-LD” [En línea]: https://json-ld.org/

[17] “Página de Wikipedia sobre los Servlets Java” [En línea]: https://en.wikipedia.org/wiki/Java_servlet

[18] “Parseador de JSON en Stack Overflow” [En línea]: https://stackoverflow.com/questions/4810841/how-can-i-pretty-print-json-using-javascript

[19] “Página de Wikipedia sobre XACML” [En línea]: https://en.wikipedia.org/wiki/XACML

[20] “Documentación para crear una politica XACML de WSO2” [En línea]: https://docs.wso2.com/display/IS540/Creating+a+XACML+Policy

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Sun Jan 14 21:44:51 CET 2018
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)