



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Departamento de Ciencia de la Computación

DESARROLLO DE PLUGINS DISTRIBUIDOS COMO SERVICIOS WEB REST

JUAN IGNACIO PUMARINO RODRÍGUEZ

Memoria para optar al título de
Ingeniero Civil de Industrias,
con Diploma en Ingeniería de Computación

Profesor Supervisor:

JAIME NAVÓN COHEN

Santiago de Chile, marzo de 2010

©MMX JUAN IGNACIO PUMARINO RODRÍGUEZ



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Departamento de Ciencia de la Computación

DESARROLLO DE PLUGINS DISTRIBUIDOS COMO SERVICIOS WEB REST

JUAN IGNACIO PUMARINO RODRÍGUEZ

Memoria presentada a la Comisión integrada por los profesores:

JAIME NAVÓN COHEN

CRISTIÁN TEJOS NÚÑEZ

ANDRÉS NEYEM

Para completar las exigencias del título de
Ingeniero Civil de Industrias,
con Diploma en Ingeniería de Computación

Santiago de Chile, marzo de 2010

A.M.D.G.

AGRADECIMIENTOS

Este trabajo no podría haber llegado a buen término de no ser por la importante colaboración de muchas personas. Quiero agradecer a algunas de ellas en particular.

Gracias a mi papás, Jaime y María Antonieta. Ustedes me enseñaron a ser un hombre de bien y a hacer de mi vocación un instrumento para mejorar el mundo y ser feliz. Sigo inténtadolo día a día. Gracias por el apoyo y la comprensión permanente en mis aventuras profesionales y académicas; en los aciertos y desaciertos.

Gracias a mis hermanos, Felipe, Carolina y Javiera; por mostrarme que en la familia hay un refugio permanente desde donde salir a enfrentar el mundo y que en ella las cosas sencillas de la vida son las que más valen. Gracias por enseñarme a aprender y discutir, a hacerme preguntas y buscar respuestas, e incluso a leer, escribir, sumar y nadar.

Gracias a mis amigos de la Escuela de Ingeniería, en especial a Pablo, Carlos, Cristóbal, Fernando, Francoise y Sebastián. Ustedes han sabido acompañarme en un camino lleno de oscilaciones, vaivenes y bytes. Forjé mi vocación junto a ustedes y desde la distancia hoy me ayudan no perder de vista el horizonte.

Gracias, Gabriela, por tu paciencia y la cuota precisa de impaciencia. Por hacerme descubrir la profunda fortaleza del amor que nos tenemos y la familia que estamos construyendo. Gracias en especial por ayudarme a encontrar a Dios tanto en las grandes decisiones que marcan la vida, como en los detalles que la van tejiendo.

Gracias a José Tomás. Desde antes de nacer ya me pedías terminar pronto este trabajo. Eres lo más maravilloso del mundo; contigo he vuelto a ser un niño para quien cada paso es un mundo nuevo. Con la ayuda de todas estas personas, te seguiré acompañando y ayudándote siempre a crecer.

Gracias a mi profesor Jaime Navón; durante este trabajo juntos la vida nos cambió por diferentes razones. Usted confió siempre en mis capacidades y en el buen destino al que finalmente hemos llegado. Gracias en especial por ver a la persona detrás del estudiante.

Le agradezco a Eduardo Santander, quien desde la Fundación Educacional Loyola me solicitó en primer lugar el desarrollo de la aplicación original, actuó como fiel cliente para refinar el producto y, finalmente, me dio su apoyo para realizar esta investigación.

Por último, gracias a Tim Berners-Lee por crear la web y a Roy Fielding por rescatar su elegante simplicidad.

ÍNDICE GENERAL

	PÁG.
DEDICATORIA.....	III
AGRADECIMIENTOS.....	IV
ÍNDICE GENERAL.....	VI
ÍNDICE DE FIGURAS.....	VIII
ÍNDICE DE TABLAS.....	IX
RESUMEN.....	X
ABSTRACT.....	XI
1.INTRODUCCIÓN.....	1
2.REVISIÓN BIBLIOGRÁFICA.....	3
2.1. Aplicaciones Extensibles.....	3
2.1.1. Desde la programación orientada a objetos.....	3
2.1.2. Plugins y extensiones.....	4
2.2. REST.....	6
2.2.1. La propuesta original.....	6
2.2.2. REST y la web.....	8
2.2.3. La Arquitectura Orientada a Recursos.....	10
2.2.4. REST y Rails.....	13
3.ANÁLISIS DE LA APLICACIÓN EXISTENTE.....	16
3.1. El problema.....	16
3.1.1. Proceso PAC.....	16
3.1.2. Proceso PAT.....	17
3.2. La solución.....	19
3.2.1. Diseño procedural.....	19

3.2.2. Diseño estructural.....	21
3.3. Limitaciones.....	23
4. REDISEÑO DE LA SOLUCIÓN.....	25
4.1. Correcciones al modelo.....	25
4.2. Simplificación del modelo.....	27
4.3. Separación entre aplicación central y plugins.....	28
4.3.1. Identificación estructural de la aplicación central.....	28
4.3.2. Identificación funcional de los plugins.....	28
4.4. Construcción de los plugins como servicios web REST siguiendo la ROA.....	30
4.4.1. URIs y enlaces.....	31
4.4.2. Métodos HTTP y representaciones.....	34
4.4.3. Códigos de respuesta.....	39
4.5. Uso de plugins desde aplicación anfitriona.....	40
5. CONCLUSIONES.....	42
5.1. Logros.....	42
5.1.1. Factibilidad de la solución.....	42
5.1.2. REST.....	42
5.1.3. Rails y ActiveResource.....	43
5.1.4. Aplicaciones de escritorio.....	44
5.2. Áreas por mejorar.....	44
5.2.1. Esquema de usuarios.....	44
5.2.2. Seguridad de comunicación.....	45
5.2.3. Tolerancia a fallas y eficiencia de comunicación.....	45
BIBLIOGRAFÍA.....	46

ÍNDICE DE FIGURAS

Figura 1: Diagrama de secuencia de proceso PAC manual.....	17
Figura 2: Diagrama de secuencia de proceso PAT manual.....	18
Figura 3: Diagrama de secuencia del proceso PAC en la aplicación original.....	20
Figura 4: Diagrama de secuencia de proceso PAT en la aplicación original.....	21
Figura 5: Modelo de clases de la aplicación original.....	22
Figura 6: Modelo de clases corregido.....	26
Figura 7: Modelo de clases corregido y simplificado.....	27
Figura 8: Modelo de clases de la aplicación central.....	28
Figura 9: Navegación entre recursos comunes entre plugins PAC y PAT.....	33
Figura 10: Navegación entre recursos específicos del plugin PAC.....	33
Figura 11: Navegación entre recursos específicos del plugin PAT.....	34

ÍNDICE DE TABLAS

Tabla 1: Comparación de categorías de servicios web.....	9
Tabla 2: Distribución de funcionalidad de la aplicación entre sus componentes.....	30
Tabla 3: Recursos, URIs y enlaces comunes entre plugins PAC y PAT.....	31
Tabla 4: Recursos, URIs y enlaces específicos del plugin PAC.....	32
Tabla 5: Recursos, URIs y enlaces específicos del plugin PAT.....	32
Tabla 6: Recursos, métodos HTTP y representaciones del plugin PAC.....	37
Tabla 7: Recursos, métodos HTTP y representaciones del plugin PAT.....	39

RESUMEN

El mundo actual es de condiciones cambiantes y escenarios poco predecibles. En este contexto, es de primera importancia que las aplicaciones sean desarrolladas con la adaptabilidad como una característica principal.

Uno de los enfoques que permiten hacer aplicaciones extensibles consiste en abrir éstas por medio de plugins: unidades de software que amplían o adaptan la funcionalidad de la aplicación anfitriona en tiempo de ejecución. Sin embargo, la distribución de plugins conserva el modelo tradicional de las aplicaciones de escritorio –la instalación local– con la consiguiente pérdida de control por parte de sus distribuidores, lo que ocasiona, entre otros, problemas de actualización de versiones y de sincronización. Esto puede impactar negativamente en las posibilidades comerciales del desarrollo de aplicaciones extensibles y también limita las potencialidades que su presencia en la web podría generar.

Ante esta realidad, el presente trabajo pretende indagar acerca de la viabilidad de desarrollar una arquitectura de plugins distribuidos como servicios web, bajo las condiciones establecidas por el estilo arquitectónico Representational State Transfer (REST).

ABSTRACT

The current world is characterized by its evolving conditions and its non predictable scenarios. In this context, it is highly important that software applications are developed considering its adaptability as a very important feature.

One of the approaches that permit the ellaboration of extensible applications consists in having them opened by means of the utilization of plugins: software units that expand or adapt the functionality of the host application at run-time. However, the distribution of plugins continue to keep the traditional model of desktop aplications –local deployment–, with its subsequent loss of control for their distributors; this fact creates, amongst others, problems with keeping versions up to date and also with its synchronization. These problems may negatively impact the commercial possibilities of the development of extensible applications and may also limit the potential that its presence in the web should generate.

In view of these facts, the present work intends to investigate the viability of developing an architecture of plugins distributed as web services under the conditions determined by the Representational State Transfer architectural style.

1. INTRODUCCIÓN

El volumen de soluciones informáticas provenientes tanto desde el mundo del software libre como del sector comercial, bajo el formato de aplicaciones web, se ha incrementado dramáticamente durante los últimos años. El desarrollo de aplicaciones accesibles desde cualquier parte del mundo en todo momento, sin embargo, no ha limitado la demanda de aplicaciones hechas a la medida. Este hecho parece demostrar que las soluciones prefabricadas en realidad no pueden ajustarse a la especificidad de los requerimientos, o lo hacen a costos demasiado elevados.

Una de las aproximaciones para atacar el problema de aplicaciones excesivamente rígidas, no adaptables a una realidad particular, consiste en abrir éstas a la inclusión de plugins o extensiones: pequeñas unidades de software que pueden ser desarrolladas por terceros y permiten ampliar o adaptar la funcionalidad de la aplicación anfitriona.

Ruby on Rails es uno de los frameworks para desarrollo de aplicaciones web, basado en la arquitectura Modelo-Vista-Controlador, con mayor crecimiento en el último tiempo. Desde sus inicios en 2004, Rails ha hecho una transición desde un primera etapa orientada a poner sobre la mesa su filosofía de desarrollo (basada, fundamentalmente, en los conceptos *Don't Repeat Yourself* y *Convention over Configuration*) hacia un segunda fase en que la plataforma, ya consolidada y con muchas otras siguiendo sus pasos, ha conseguido un lugar privilegiado en el mercado de aplicaciones web. Desde el lanzamiento de su versión 2.0, la comunidad Rails ha dirigido sus esfuerzos a proponer y motivar la utilización del estilo arquitectónico para la web *Representational State Transfer* (REST), propuesto por Roy Fielding en 2000.

Las arquitecturas REST consisten, fundamentalmente, en la identificación de los recursos que componen un sistema, concentrándose en proponer una ontología rica en sustantivos antes que en los verbos que operan sobre ellos. Aún más, REST, asociado directamente con la web, propone utilizar sólo los cuatro métodos esenciales del protocolo HTTP: GET, POST, PUT y DELETE para identificar las operaciones sobre los recursos del sistema.

La comunidad Rails ha hecho su propia interpretación del estilo REST desde el punto de vista del desarrollo de aplicaciones web que utilizan bases de datos relacionales. Por medio de este esfuerzo, ha demostrado el impacto que puede tener para el desarrollo a nivel de arquitectura de las aplicaciones, especialmente en la identificación de los modelos y la organización de los controladores de las aplicaciones basadas en MVC.

El presente trabajo tiene por objetivo indagar acerca de la incidencia que tiene la aplicación de REST en la eficacia y eficiencia del desarrollo de aplicaciones extensibles a través de plugins aprovechando las facilidades de Rails.

En la práctica, se trabajará sobre la base de una aplicación previamente desarrollada por el autor para la gestión de donaciones de una fundación educacional, actualmente en pleno uso productivo. Esta aplicación actúa intermediando y automatizando el proceso de comunicación de los mandatos de donaciones a los sistemas de Pago Automático de Cuentas (PAC) del Banco de Chile y Pago Automático con Tarjeta (PAT) de Transbank. La intención es otorgar a la aplicación la posibilidad de gestionar otros procesos de recaudación por medio de plugins, desacoplándola de esta manera de los sistemas ya mencionados y abriendo así la posibilidad de su uso en otras instituciones con necesidades similares.

2. REVISIÓN BIBLIOGRÁFICA

2.1. Aplicaciones Extensibles

2.1.1. Desde la programación orientada a objetos

Entre los conceptos que caracterizan al paradigma de programación orientada a objetos (Booch, 2007; Schach, 2004), podemos mencionar tres que son esenciales: el encapsulamiento, el polimorfismo y la herencia.

El encapsulamiento, en particular, propone la noción de enmascarar los contenidos internos de un objeto, y dar acceso a modificarlo sólo a través de una interfaz. Una de las grandes ventajas que se aprecian en el encapsulamiento es la posibilidad de modificar el comportamiento interno de los objetos de una clase sin pasar a llevar la forma en que otros actores involucrados hacen uso de ellos. En otras palabras, se forma una especie de contrato respecto al comportamiento externo de los objetos de una clase, pero la manera de llevarlo a cabo en su interior queda a su discreción.

La herencia permite la especialización de clases de objetos que difieren en ciertos atributos, pero que tienen un conjunto común de funcionalidades. Junto al polimorfismo, la herencia permite de que otros agentes del sistema utilicen determinados objetos independientemente del nivel de especialización de sus clases, sólo importando que respondan al contrato establecido por las clases que están más arriba en la jerarquía.

Por medio de estos tres conceptos, el esquema de la programación orientada a objetos propone desde su base establecer ciertos puntos controlados de interacción entre los objetos de diferentes clases. Estas características pueden utilizarse beneficiosamente para extender la funcionalidad de una aplicación aislando las áreas de preocupación entre ellas. Sin embargo, este es un esquema adecuado para el diseño estático de un sistema, pero no resuelve, en general, el problema de hacer extensible una aplicación en tiempo de ejecución. En parte, este diagnóstico acerca de los problemas de la herencia respecto a la extensibilidad de las aplicaciones ya fue detectado por Gamma et al. (1995).

La programación orientada a objetos efectivamente permitió que los diseños de las aplicaciones computacionales fueran menos acoplados. Esto no se tradujo en la utopía de la creación de “repositorios” de clases intercambiables que los desarrolladores utilizarían a voluntad, estimulando la reutilización de código entre aplicaciones. Sin embargo, en la estructura interna de las aplicaciones la programación orientada a objetos efectivamente permitió una tendencia hacia la reutilización y así evitar la duplicación de funcionalidad. Se estimuló la generación de interfaces contractuales que permitieran reducir el esfuerzo en la modificación de partes individuales de las aplicaciones, pues el radio de acción de un determinado trozo de código estaba entonces mucho más acotado.

Podemos atribuir en gran medida el desarrollo de la Ingeniería de Software moderna a las posibilidades que entregó el paradigma de orientación a objetos, en oposición a la cultura *hacker* y al *spaghetti code*. Hoy diferentes grupos de personas pueden especializarse en áreas específicas de una aplicación e interactuar con otras a partir de los “contratos” que regulan estas interacciones (Hunt y Thomas, 1999).

2.1.2. Plugins y extensiones

La tendencia de encapsular áreas del software derivó, con el tiempo, en que ciertos fabricantes descubrieran un importante potencial en el permitir que determinadas responsabilidades de sus aplicaciones fueran cumplidas por componentes intercambiables en tiempo de ejecución. Podemos identificar en el reproductor de medios Winamp un caso paradigmático de esta tendencia (Fowler, 2002; Mac-Vicar y Navón, 2005).

El desarrollo de estos componentes podría darse por actores especializados en ciertas áreas, como el caso de los códecs de audio, reduciéndose así el tamaño de la aplicación base; también el dejar una puerta abierta en una aplicación implica la posibilidad de que terceros descubran potencialidades no previstas por el desarrollador original o que, por limitaciones técnicas o económicas, no pudo desarrollar. La sección de descarga de *plugins* de Winamp es aún un notable ejemplo del desarrollo de un producto y sus complementos guiado y producido por una comunidad.

Otro modelo, similar en motivaciones, pero diferente en potencialidades, corresponde a las *extensiones*. Hoy en día, la aplicación más popular que encarna este modelo es el navegador web Mozilla Firefox. Las extensiones –a diferencia de los *plugins*– tienen por objetivo complementar la aplicación de un modo más externo que interno, llevando a cabo tareas que la aplicación anfitriona no necesariamente tenía consideradas. Las extensiones suelen aprovechar una infraestructura base (*look and feel*, interacción con el sistema de archivos, notificaciones, preferencias, etc.), transformando a las aplicaciones en verdaderas plataformas de desarrollo. Hoy podemos decir que Firefox se ha constituido en un verdadero soporte para desarrollar aplicaciones multiplataforma.

Tanto las extensiones como los *plugins*, sin embargo, comparten un mismo modelo de distribución: ubicar un código en una parte específica del equipo local desde donde la aplicación anfitriona los recuperará para ser ejecutados. El modelo, sin duda alguna, funciona bien, pero su distribución impone ciertas restricciones que, a la luz de las tendencias dominantes en la web, pueden adquirir relevancia. Un primer problema que podemos identificar consiste en la sincronización de versiones de las extensiones, aún más cuando esto implica controlar las versiones de la aplicación anfitriona.

Otra área importante tiene que ver con la predominancia de aplicaciones web que suplantán a las tradicionales aplicaciones de escritorio. Lo que en su momento se inició con el correo electrónico hoy ya se puede ver con suites de oficina y aplicaciones de reproducción de medios digitales. La distribución tradicional de *plugins* en este contexto simplemente no tiene sentido.

Finalmente, podemos llamar la atención sobre la necesidad de control que ciertas empresas pueden desear tener sobre determinados algoritmos, particularmente en el desarrollo de aplicaciones del área científica (investigación de operaciones, inteligencia artificial y *business intelligence*, por mencionar algunos) donde la confidencialidad puede ser un elemento competitivo importante.

2.2. REST

Durante los últimos tres años la sigla REST (*Representation State Transfer*) ha pasado a ser una más dentro del vocabulario tecnológico dominante en el desarrollo de aplicaciones y servicios web. Muchos servicios web comerciales y otros tantos esfuerzos de código libre describen sus soluciones como RESTful (Amazon, Yahoo, Flickr, CouchDB, etc.) En esta sección se pretende abordar la definición de REST desde el punto de vista original utilizado por su creador, identificando sus características esenciales y la real relación que guarda con la web. Se estudiará brevemente el impacto que tienen las restricciones REST para el desarrollo de servicios web y el diseño estructural de las aplicaciones. Finalmente, se describirá la Arquitectura Orientada a Recursos (ROA), una aplicación específica del estilo REST.

2.2.1. La propuesta original

Roy Fielding (2000), tras trabajar por años en la definición de estándares básicos de la web; tales como el protocolo HTTP, el lenguaje HTML y las URIs, propone a REST, en su tesis doctoral, como un “estilo de arquitectura de software para sistemas de hipermedios distribuidos”. La intención del autor es dar cuenta, por medio de un estilo arquitectónico abstracto y separado de su implementación concreta, de las razones que hacen de la web un modelo exitoso para el desarrollo de aplicaciones. En este sentido, REST, si bien se inspira en ella, está intencionalmente separado de la definición de la web y, de hecho, se ha utilizado para describir otros sistemas distribuidos.

Fielding, por otra parte, reconoce que la web tiene un diseño arquitectónico subyacente intencional; mas, en la práctica, puede ser utilizada sin tomar este diseño en cuenta, abusando de sus características. Siguiendo la analogía propuesta por Richardson y Ruby (2007): si bien hay lenguajes de programación con un diseño explícitamente orientado a objetos (por ejemplo, Java), el desarrollador perfectamente puede hacer un diseño procedural de su aplicación sobre el lenguaje, obviando los beneficios inherentes a utilizarlo según su diseño.

Tras una introducción a la terminología básica, Fielding propone en el capítulo 2 de su tesis una serie de definiciones que le permiten evaluar el diseño de arquitecturas de software: define siete propiedades de interés para realizar esta evaluación: desempeño, escalabilidad, simplicidad, modificabilidad, visibilidad, portabilidad y fiabilidad. Con estos criterios en mente, en el capítulo 3 se dedica a describir un amplio conjunto de estilos arquitectónicos basados en redes. En el cuarto capítulo describe los requerimientos específicos de la WWW.

Finalmente, en el quinto capítulo, Fielding finalmente propone REST como un estilo arquitectónico que se deriva de los estudiados anteriormente y que pretende atender los requerimientos de la web. Así, este estilo se compone de cinco de los presentados en el capítulo 3:

- Funciona sobre una arquitectura cliente – servidor
- Los servidores no almacenan el estado de los clientes
- La comunicación es *cacheable* (puede ser repetida cuando no hay cambios)
- Los clientes y los servidores se comunican utilizando una interfaz uniforme
- Los servidores se pueden montar en capas, de manera transparente para los clientes

Estas condiciones tienen por objetivo aumentar la interoperabilidad, el desempeño, la escalabilidad y la fiabilidad de los sistemas.

Adicionalmente, Fielding hace especial hincapié en la definición de la interfaz uniforme que debe ser provista por un sistema RESTful. Esencialmente, define cuatro condiciones para ellas:

- Los recursos deben ser identificados de manera explícita en las interacciones
- Los recursos deben ser manipulados por medio de representaciones
- Los mensajes para la manipulación de recursos deben ser autodescriptivos
- El estado de los clientes sólo cambia por medio de interacciones de hipermedios, es decir, siguiendo enlaces a otros recursos presentes en sus representaciones

2.2.2. REST y la web

Como se mencionó en la sección anterior, REST nace de una descripción del funcionamiento de la web, planteándose como una descripción genérica de sistemas que funcionan *como la web*. En este sentido, la tesis de Fielding no ahonda mayormente en la forma en que las aplicaciones que funcionan sobre la web deben comportarse para ser *RESTful*.

En este contexto, Sam Richardon y Leonard Ruby publican *RESTful Web Services* (2007). En esta publicación, pretenden establecer un enlace práctico entre las restricciones REST y el desarrollo de servicios web.

En primer lugar describen el estado actual en cuanto a estándares en materia de servicios web, fundamentalmente basados en las recomendaciones WS-* y SOAP, y también ciertos servicios concretos como los provistos por Amazon, Yahoo, Flickr y del.icio.us, algunos de los cuales declaran ser *RESTful*.

En este análisis proponen dividir las arquitecturas de servicios web en tres categorías: en primer lugar identifican la categoría de servicios web tradicionales, RPC (*Remote Procedure Call*). Estos servicios se caracterizan por ser opacos: la operación y la identificación de los recursos sobre los cuales se realizan van dentro de un sobre (usualmente SOAP) que se envía en el cuerpo de un mensaje POST a un *endpoint* único. Las operaciones son libres, tal como si fueran nombres de funciones o procedimientos, de donde se origina su nombre. De este modo, los servicios RPC terminan transformando a HTTP en un protocolo de transporte, ignorando todas sus características distintivas (incluyendo la posibilidad de hacer caché de las respuestas).

Luego detallan la categoría de servicios web completamente RESTful: éstos se distinguen por utilizar exclusivamente los verbos HTTP (GET, POST, PUT y DELETE) para expresar las acciones sobre los recursos y para indicar el *scoping* de acción se hace uso de la URI, con lo que, a diferencia del caso SOAP, existe un *endpoint* para cada recurso. Es esta riqueza de sustantivos la que permitiría soslayar la aparente limitación que impone el utilizar sólo los verbos HTTP.

Finalmente, Richardson y Ruby identifican una categoría intermedia entre los servicios puramente REST y los RPC, a la que llaman “híbridos REST-RPC”. Éstos toman de REST la identificación de recursos en la URI, pero descartan el uso de los verbos HTTP, incluyendo también la acción en la misma URI, por lo que éstas, tal como en el caso de RPC. Los *endpoints*, en este caso, se producen haciendo un cruce entre los sustantivos y los verbos.

Categoría de servicio web	Acciones disponibles	Expresión de acción a realizar	Expresión de recursos	Endpoints
RPC	Ilimitadas	Dentro de un sobre	Dentro de un sobre	Único
REST	GET, POST, PUT y DELETE	En el método HTTP	En la URI	Uno por cada recurso
Híbrido REST-RPC	Ilimitadas	En la URI	En la URI	Uno por cada par recurso-acción

Tabla 1: Comparación de categorías de servicios web

Como se puede ver en la tabla 1, estas categorías, desde el punto de vista concreto de la web, tienen criterios claros para clasificar los servicios web dependiendo del uso que hacen de las tecnologías fundamentales: el protocolo HTTP y el estándar de URIs.

Una de las ventajas de utilizar el esquema RPC consiste en la posibilidad de especificar un contrato de funcionamiento de los métodos publicados por el servicio web, el que indica claramente cuáles son los parámetros de entrada y el resultado esperados, tal como ocurre con las declaraciones de funciones en los lenguajes de programación. Una forma programática de expresar estos contratos consiste en el lenguaje WSDL (*Webservice Description Language*), de amplia difusión en el contexto de servicios web SOAP (Cerami, 2002). Contar con el WSDL asociado a un servicio web da un punto de partida sencillo para la generación automática de código en los lenguajes de programación, transformando al servicio web, en la práctica, en un símil de un objeto local, con estado y llamadas a métodos.

Las arquitecturas REST, en cambio, aún tienen camino por recorrer respecto a la definición de contratos. Podemos distinguir dos filosofías: en primer lugar, aquélla implementada por los lenguajes descriptivos WADL (*Web Application Description Language*) y WSDL (en su versión 2.0). Éstos consisten en la especificación a priori de la estructuras de URIs que identificarán los recursos interesantes definidos por un servicio web. Además, indican las acciones (HTTP) que es posible realizar sobre cada tipo de recurso y las representaciones aceptadas y entregadas por cada una.

Por otra parte, Fielding (2008), defendiendo el principio de conectividad propuesto por REST, donde los recursos se enlazan entre ellos a través de representaciones, manifiesta que "una API REST API no debe definir un nombre o jerarquía fija de recursos" pues esto implica una obvio acoplamiento entre clientes y servidores. Tilkov (2008) y Vinoski (2008) también dedican artículos a esclarecer la implementación correcta de servicios web REST en este contexto. En esta misma línea, Allamaraju (2008) propone una descripción alternativa de servicios web REST, en la que existe un conjunto acotado de URIs que dan el punto de partida desde donde se obtienen enlaces a los demás recursos. Para que esto funcione adecuadamente se hace necesaria una documentación acabada de los tipos de medios y las relaciones entre recursos. Con esto, los clientes ya no tienen información estructural sobre los servidores y de esta manera descubren navegando (al modo de la web humana) los recursos de interés.

2.2.3. La Arquitectura Orientada a Recursos

En consideración a esta situación es que Richardson y Ruby proponen la Arquitectura Orientada a Recursos (Resource-Oriented Architecture, ROA). La ROA pretende ser un método sistemático para desarrollar servicios en la web real que adhieran a los principios REST.

Como se puede intuir, la identificación de recursos aparece como un elemento esencial de la ROA. Un recurso consiste en una entidad que será lo "suficientemente importante para ser referenciada". Tal definición, ciertamente subjetiva, denota que esta

identificación es una labor fundamentalmente de diseño y debe estar muy relacionada con el dominio en donde se moverá la aplicación.

Para poder ser llamado como tal, cada recurso debe tener al menos un punto de acceso único que lo distinga de otros. En la web, la ROA identifica estos puntos de acceso con las URIs. En este punto podemos apreciar uno de los valores del enfoque REST. Las URIs, como un caso especial de penetración de terminología técnica en un ambiente global, ya han sido largamente utilizadas como puntos de referencia, habiéndose transformado en un elemento de intercambio entre humanos, similar a las direcciones postales, con el que queremos decir “aquí encontré algo; tú también podrás encontrarlo aquí”.

De igual modo, los nombres (URIs) de los recursos permiten que éstos se interrelacionen, formando una red interconectada navegable por agentes electrónicos, tal como funciona la web “humana”.

Ante las restricciones propias de los sistemas computacionales, no es esperable que en éstos se almacenen los recursos, dado que son entidades eminentemente abstractas. La forma en que la información acerca de ellos es transmitida se conoce como su representación. Estas representaciones pueden tomar muchas formas (texto libre o estructurado, imágenes, audio, etc.). Finalmente, establecer cuál representación o conjunto de representaciones es el más adecuado para un recurso se transforma en una labor de diseño más.

La ROA utiliza estos conceptos y define cuatro propiedades que un sistema que adscriba a ella debe satisfacer.

En primer lugar, está la *addressability*, que traduciremos como referenciabilidad. En la práctica, esto quiere decir que una aplicación que adscribe a la ROA debe exponer cada trozo de información relevante a través de una URI.

En segundo lugar, la propiedad de *statelessness*, la que podemos traducir como carencia de estado. En términos generales, esto quiere decir que cada solicitud HTTP debe realizarse de manera independiente de las otras, por lo que debe contener toda la información necesaria para ser llevada a cabo. El estado de la aplicación, por el

contrario, se debe expresar a través de las mismas URIs. Es por esto que la carencia de estado está íntimamente ligada con la referenciabilidad. Parte de la simplicidad de la construcción de servidores, clientes e intermediarios HTTP tiene que ver justamente con esta característica.

Luego, en tercer lugar, consideramos la propiedad de *connectedness* o conectividad. Ésta exige que los recursos no vivan de manera aislada, sino que establezcan enlaces entre ellos, entregando al cliente los estados vecinos al actual y posibilitando así la navegación. Sin esta característica, considerando la propiedad de carencia de estado, los clientes no tendrían cómo acceder a diferentes estados.

Finalmente, la propiedad de interfaz uniforme consiste en que todos los recursos respondan al mismo conjunto de acciones. En principio, esto no necesariamente se restringe a los verbos HTTP; una aplicación podría definir acciones adicionales siempre y cuando todos los recursos respondieran a ellas, en vez de definir acciones particulares para cada recurso. La objeción a una práctica como ésta (que aplicaciones como WebDAV implementan, añadiendo 8 métodos HTTP), está relacionada con la incompatibilidad que se produce entre la aplicación en cuestión y el resto de la web, reduciendo la interoperabilidad.

Con estos conceptos fundamentales, los autores proponen una metodología general para el desarrollo de servicios web que adhieran a la ROA. Esta metodología se estructura en los siguientes pasos:

- Describir el conjunto de datos
- Separar el conjunto de datos en recursos
- Para cada recurso:
 - Nombrarlo con una URI
 - Decidir cuáles métodos HTTP son válidos para interactuar con el recurso y bajo cuáles circunstancias (autorización)
 - Diseñar las representaciones que se aceptan desde el cliente
 - Diseñar las representaciones servidas hacia el cliente

- Integrarlo con otros recursos asociados, estableciendo enlaces incluidos en la representación servida al cliente
- Definir las respuestas posibles para el curso de eventos normal
- Definir las respuestas posibles antes condiciones de error

En la labor de diseño que se llevará a cabo en el capítulo 4 acudiremos a esta metodología para el diseño de un servicio web REST.

2.2.4. REST y Rails

Desde el año 2004, Ruby on Rails, un framework para el desarrollo de aplicaciones web basado en el patrón Modelo-Vista-Controlador (Fowler, 2002), ha ido ganando adherentes tanto al software en sí mismo como a su filosofía de desarrollo, marcando un modelo para otros frameworks en diversos lenguajes de programación. Esta filosofía tiene como base dos conceptos fundamentales: *convention over configuration* y *don't repeat yourself*. El primero de ellos hace alusión a la toma de decisiones por parte del framework para intentar cubrir la mayor cantidad de casos, dejando la necesidad de modificar los valores por defecto en la menor cantidad de casos posible. Estas decisiones –a veces polémicas– incluyen el formato de nombres de las tablas en las bases de datos, la definición de claves primarias como identificadores numéricos autoincrementables, el significado de las URIs y la pluralización de los sustantivos relevantes (Thomas y Heinemeier, 2009).

En la versión 1.2 de Rails, lanzada a principios del año 2007, se introduce por vez primera en el corazón del sistema a REST. Rails toma varios de los principios propuestos por Fielding y los incorpora (siguiendo la filosofía de *convention over configuration*) en el desarrollo de las aplicaciones:

- Estructurar los controladores de las aplicaciones en torno a *recursos*.
- Organizar estos controladores con una interfaz uniforme, que responderán a los métodos HTTP adecuados.

- Entregar formas de expresar en HTML el acceso a los recursos por medio de la interfaz uniforme, incluyendo el uso de los métodos HTTP que no son soportados por los navegadores web.
- Proveer un mecanismo adecuado para servir diferentes representaciones de los recursos dependiendo de los clientes que acceden a ellos.

Rails, como un framework orientado al desarrollo de aplicaciones web que utilizan sistemas de bases de datos relacionales, establece una fuerte relación entre los métodos HTTP y las operaciones básicas de estos sistemas: crear, leer, actualizar y eliminar (CRUD, por sus iniciales en inglés). La adopción de las restricciones REST traerá aparejada la necesidad de repensar la manera en que se diseñan los modelos. Un ejemplo habitual de esto es, por ejemplo, el identificar con un sustantivo las tablas utilizadas para enlaces N a N entre otras dos tablas. A modo de ejemplo, donde el modelo tradicional identificaba una tabla de enlace entre artículos y lectores, cuyo nombre no tenía mayor importancia, REST privilegia llamar este enlace como una “suscripción” e identificar las acciones que tiene sentido hacer sobre ella.

Finalmente, en diciembre de 2007, se lanza Ruby on Rails 2.0. Esta versión incorpora todos los elementos necesarios para la construcción de aplicaciones web RESTful, al menos bajo la concepción entendida por los propios creadores del framework. Tres años después, ad portas del lanzamiento de la versión 3.0, la funcionalidad añadida en el área REST ha tenido que ver con simplificar el trabajo de los desarrolladores, nuevamente por medio de la selección de valores por omisión razonables, pero en lo fundamental conserva la misma aproximación.

La versión 2.0, además, incorporó una tecnología nacida de la integración del estilo REST. ActiveResource es una capa de abstracción, al modo que hace ActiveRecord con las bases de datos relacionales, de servicios web RESTful. Esto permite a los desarrolladores trabajar con modelos remotos de una manera muy similar a la utilizada por ActiveRecord de manera transparente. En la práctica, ActiveResource impone ciertas condiciones adicionales por sobre las de REST que son satisfechas por los propios modelos desarrollados con Rails, por lo que se transforma en la metodología ideal para

integrar aplicaciones desarrolladas con este framework (Thomas y Heinemeier, 2009). La apuesta de Rails es tan grande con respecto a esta tecnología que incluso eliminó de su *core* las funcionalidades de integración con servicios web tradicionales basados en SOAP.

En lo que resta de este trabajo, será parte de la evaluación del framework Rails revelar qué tan restrictivas pueden ser las condiciones adicionales a la hora de crear *plugins* distribuidos como servicios web REST.

3. ANÁLISIS DE LA APLICACIÓN EXISTENTE

Para el análisis de este trabajo, se tomará como caso de estudio una aplicación desarrollada por el autor y actualmente en producción, utilizada en la Fundación Educacional Loyola para gestionar las donaciones que ingresan a ella a través de sistemas bancarios. A continuación se detallará el problema a resolver y la solución original.

3.1. El problema

El problema original que la aplicación soluciona es la automatización de ciertas tareas rutinarias que el administrador de la fundación debía realizar para concretar los mandatos de donaciones ordenados a través de los sistemas Pago Automático de Cuentas (PAC) del Banco de Chile y Pago Automático con Tarjeta (PAT) de Transbank.

Ambos tipos de proceso de donación difieren en lo concerniente a las interacciones con sus entidades y los datos que requieren para llevarse a cabo.

3.1.1. Proceso PAC

El proceso PAC (Figura 1), involucra la recepción de la llamada "base de datos universo", desde el Banco de Chile, la que se procede a contrastar –originalmente, en forma manual– con la información que posee la Fundación. Tras este proceso de filtrado, se procede a generar y luego enviar el "archivo de cargos" al banco. Este archivo contiene, en un formato de texto con campos de ancho fijo definido por el mismo banco, la información de los cuentacorrentistas y los cargos a ser realizados en sus respectivas cuentas (como se puede apreciar, el formato es dependiente del banco al que pertenece la cuenta del mandato). Transcurrido un plazo de 3 días hábiles, el banco procede a informar, por medio de un "archivo de rendiciones", cuáles han sido los resultados de intentar hacer los cargos correspondientes en las cuentas corrientes indicadas, estableciéndose el monto final que el banco deposita en la cuenta corriente de la Fundación.

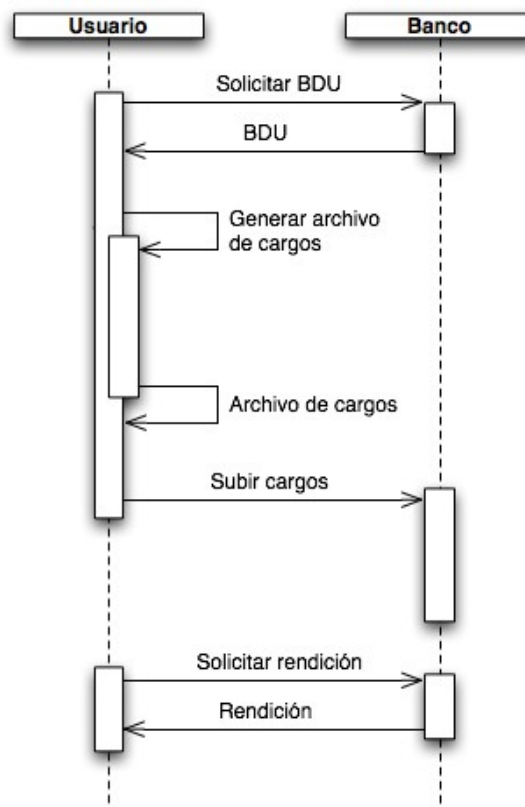


Figura 1: Diagrama de secuencia de proceso PAC manual

3.1.2. Proceso PAT

El proceso PAT (Figura 2) es relativamente más sencillo. En el esquema de Transbank, el receptor de los mandatos –la Fundación, en este caso– no necesita hacer un prefiltrado de sus datos previo al envío de la información; da inicio al proceso con el envío de un "archivo de cargos" sólo considerando la información disponible en sus registros. Este archivo, a diferencia del caso PAC, es de texto con campos separados por comas, en los cuales se indican el tipo de tarjeta (Visa, MasterCard, etc.), el número, fecha de vencimiento y el monto de cargo. Luego, transcurrido un período de tiempo establecido, de manera similar al caso PAC, Transbank da cuenta de los resultados de intentar hacer los cargos por medio del envío de un "archivo de rendiciones" a la Fundación. El proceso PAT tiene la dificultad adicional de requerir del almacenamiento

seguro de números de tarjetas de crédito y su obligatoria recuperación en texto plano al momento de ser ejecutado, para hacer efectivos los cargos.

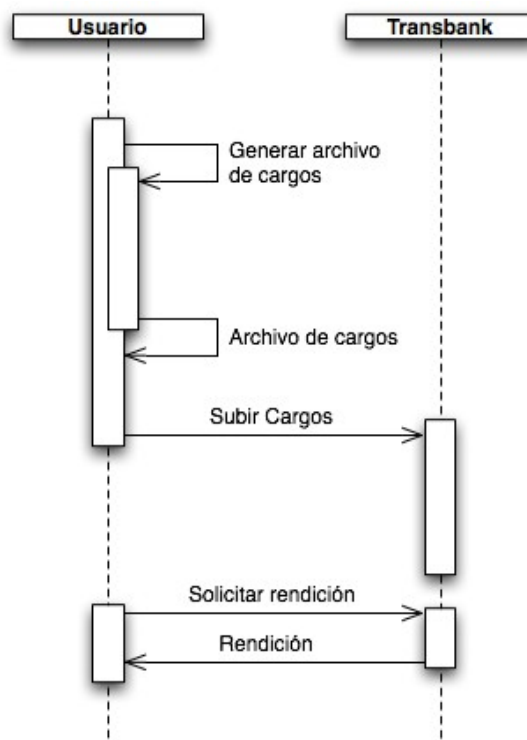


Figura 2: Diagrama de secuencia de proceso PAT manual

Antes de la confección del software, todos estos procesos se realizaban en forma manual, incluyendo la generación de los archivos de cargos. Además, la información detallada de los resultados entregada por ambas instituciones no recibía mayor análisis dada la complejidad de su formato, haciendo muy difícil la posibilidad de hacer seguimiento de los donantes, identificando aquéllos con problemas para responder a sus mandatos o contactando a quienes tenían mandatos próximos a expirar.

3.2. La solución

3.2.1. Diseño procedural

Para el caso del proceso PAC (Figura 3) se puede ver que esto se basa en dos interacciones con la aplicación (el Sistema de Donaciones): una para subir la Base de Datos Universo, recuperada desde el sistema del Banco, con la que el sistema contrasta la información que éste contiene. A partir de esto se genera un archivo de cargos que el usuario descarga para luego subir hacia el sistema del Banco.

Transcurrido el plazo de ejecución de los mandatos, el sistema del Banco pone a disposición del usuario el archivo de rendiciones, el que luego será subido al sistema de donaciones, información con la cual se podrán realizar gestiones tendientes a llevar un control más acabado del comportamiento de los donantes en el tiempo y, eventualmente, hacer seguimiento de manera personalizada.

Las intervenciones son similares en el caso del proceso PAT (Figura 4). La gran diferencia la marca la necesidad de utilizar un archivo con una clave privada para obtener el archivo de cargos. En el proceso original esta información no se almacenaba encriptada, con los consiguientes riesgos de seguridad que esto acarrea.

Es importante mencionar que se optó intencionalmente por no automatizar (por medio de *screen scraping*) la interacción con el sistema del banco. Por una parte, los posibles cambios en las interfaces podrían acarrear problemas de mantenibilidad y, por otra, la posibilidad de perder un mes de donaciones por incumplimiento de plazos aparecía excesivamente riesgosa.

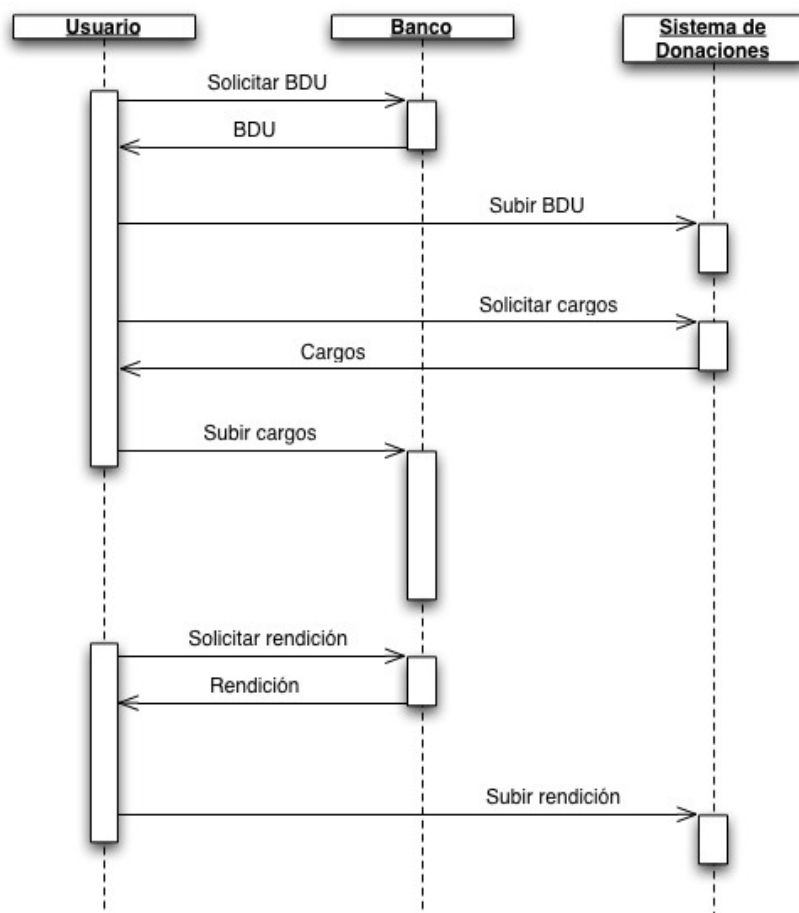


Figura 3: Diagrama de secuencia del proceso PAC en la aplicación original

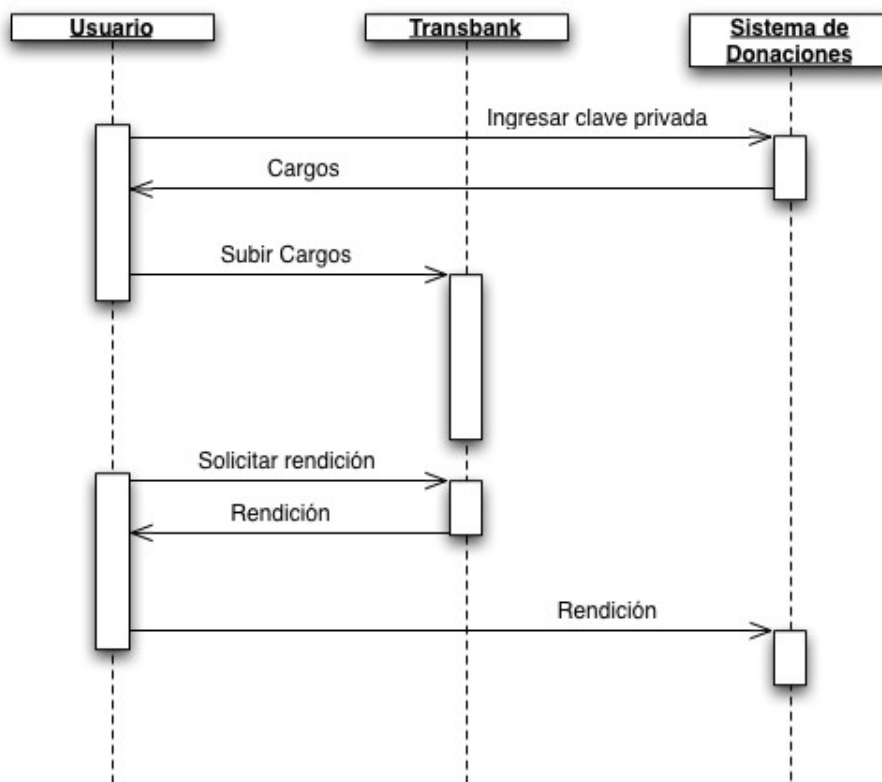


Figura 4: Diagrama de secuencia de proceso PAT en la aplicación original

3.2.2. Diseño estructural

En la Figura 5 se puede apreciar el modelo de clases de la aplicación original. Por una parte, se tiene un área claramente separada donde se administran los usuarios del sistema, sus roles y los privilegios asociados a ellos. El esquema de autorización de esta aplicación es relativamente sencillo, expresando los privilegios, sencillamente, como los métodos de cada controlador. Considerando que el sistema es utilizado por un grupo restringido de usuarios con una jerarquía simple, este esquema ha probado ser suficientemente adecuado.

La clase Mandato es el concepto central de la aplicación. En torno a este modelo tenemos las personas a quienes pertenecen y una asociación única con un medio de pago

(también pertenecientes a una persona específica) con el que el mandato debe llevarse a cabo.

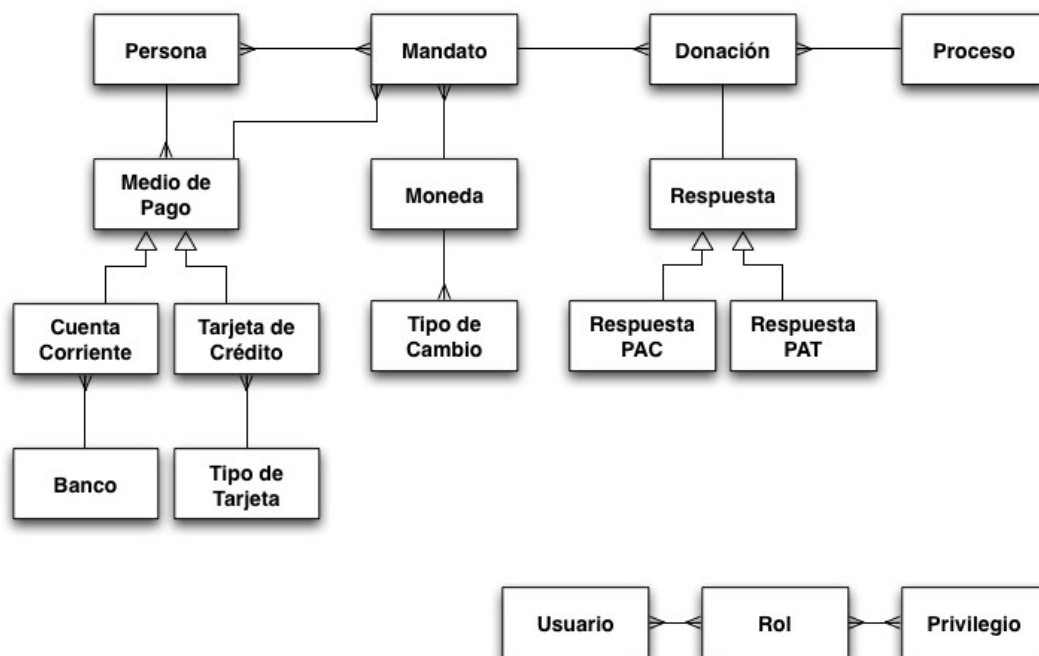


Figura 5: Modelo de clases de la aplicación original

La clase Medio de Pago corresponde a una abstracción de los dos medios de pago originalmente disponibles: cargo en Tarjeta de Crédito y cargo en Cuenta Corriente. Las tarjetas de crédito están asociadas a un tipo (Visa, MasterCard, etc.), que además indican el formato de su numeración, información que se utiliza en la interfaz de usuario para ingresar y hacer una validación básica. Las cuentas corrientes, por otra parte, están asociadas a un banco, cada uno de ellos con un código oficial. Además, en el proceso de generación de los archivos de cargos descritos en la sección anterior, cada banco tiene un formato en el que entrega la información de los cuenta-correntistas; esto también se guarda en este modelo.

Para solucionar el problema del almacenamiento seguro de los números de tarjetas de crédito descritos en la sección anterior, se utiliza un esquema de encriptación asimétrica. Al momento de ingresar los números de tarjetas de crédito a través de la interfaz de

usuario, éstos son encriptados con una clave pública y guardados en la base de datos. Para obtenerlos (lo que es exigido por Transbank para llevar a cabo los cargos), el sistema pide ingresar el archivo con la clave privada correspondiente. Es posible, además, hacer una reencriptación de los números de tarjeta y obtener un nuevo par de claves pública y privada conociendo la clave privada original.

La separación entre los mandatos y los medios de pago fue una decisión tomada por un esquema más teórico que práctico: la posibilidad de que un mismo medio de pago fuera utilizado para diferentes mandatos de la misma persona. Éste es un punto que será intervenido en el rediseño a llevarse a cabo en el siguiente capítulo.

A la derecha del diagrama vemos la clase Proceso. Esta clase modela los procesos de ejecución de los mandatos que mensualmente realiza la Fundación para obtener los fondos de parte de los mandantes. Como se puede ver, el proceso contiene su Tipo (PAT o PAC) sólo como un atributo, y no contenía la lógica de negocio.

La Donación es la ejecución real de un mandato, asociada a un proceso mensual específico. Éstas contienen la información histórica de cada mandato, los montos reales obtenidos en cada uno de ellos y las respuestas obtenidas por medio de los archivos de rendiciones de cada tipo de proceso mencionados en la sección anterior.

Para manejar mandatos en diferentes monedas con tipos de cambio variable, se utilizaron las clases Moneda y Tipo de Cambio, las que se aprecian arriba a la izquierda. Además, se puede ver la inclusión de modelos para las comunas donde habitan los mandantes y la pertenencia de los mandatos a ciertos proyectos, información utilizada para la recolección de ciertas estadísticas y gestiones directas con los mandantes.

3.3. Limitaciones

Como se puede ver, el diseño está íntimamente ligado a los dos tipos de proceso existentes y la posibilidad de extender su funcionalidad pasa por la adición de nuevas clases (y sus tablas), controladores y vistas para administrar y operar sus flujos.

En el capítulo siguiente se intentará diseñar una aplicación que desempeñe la misma funcionalidad, pero cuyas posibilidades puedan ser aumentadas a través de la adición de componentes intercambiables sin necesidad de intervenir su diseño estructural.

4. REDISEÑO DE LA SOLUCIÓN

En el camino de rediseñar la solución original, detallada en el capítulo anterior, con el objetivo de llegar a una aplicación basada en plugins para los diferentes tipos de proceso, se realizarán cuatro pasos.

En primer lugar, corregir el modelo a la luz de ciertas lecciones obtenidas tras tener el sistema en producción. Luego, para efectos de simplificar lo expuesto en este trabajo, se eliminarán algunas de las funcionalidades que no son importantes para el interés principal. Tras esto, se obtendrá un diseño sencillo, pero aún basado en un esquema de extensibilidad a través de herencia.

El tercer paso consistirá en separar aquellas áreas pertenecientes al core de la aplicación de aquellas relevantes para los dos tipos de proceso que se transformarán en plugins. Finalmente, revisaremos el diseño de los plugins siguiendo la metodología de la Arquitectura Orientada a Recursos, para obtener servicios web REST.

4.1. Correcciones al modelo

Una de las principales lecciones obtenidas al tener la aplicación original en producción es la excesiva flexibilidad –y consiguiente complejidad– que entregaba la separación entre los medios de pago y los mandatos asociados. La intención original fue dar la posibilidad de utilizar un mismo medio de pago en diferentes mandatos, siendo incluso posible compartir medios de pago entre los donantes.

Esta factibilidad teórica, sin embargo, contrasta fuertemente con la realidad, donde los medios de pago siempre tienen una asociación única con un mandato y nunca son compartidos. Además, un mandato, originalmente conceptualizado como un monto mensual comprometido por una persona, incluye en la realidad, como documento, la información de pago, pasando a ser parte elemental de éste.

Tomando en cuenta lo anterior, en la figura 6 se puede apreciar que la clase Medio de Pago desaparece y toma su lugar el Mandato, con cierta información base; la especialización entre los diferentes tipos de proceso está dada por las subclases de

mandatos PAT y PAC, las que reemplazan a las clases Tarjeta de Crédito y Cuenta Corriente, respectivamente.

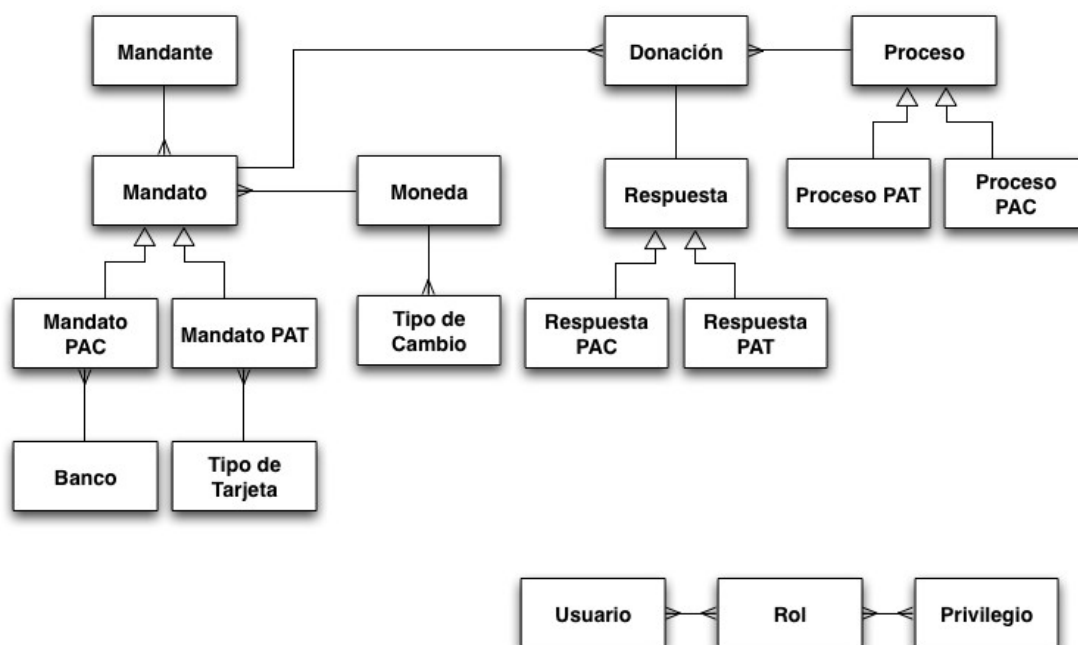


Figura 6: Modelo de clases corregido

Una segunda corrección, que también podemos apreciar en la figura 6, consiste en la conceptualización de los procesos como clases activas, con comportamiento asociado y diferenciado según el tipo. La lógica de negocio asociada a ellos fue originalmente incluida en los controladores de la aplicación; la clase Proceso era simplemente un agrupador de donaciones correspondientes a un mismo período y tipo. Este diseño está claramente reñido con lo propuesto por el patrón MVC, respecto a dejar toda la lógica de negocio exclusivamente en los modelos. En el modelo corregido, las subclases Proceso PAT y PAC contendrán esta lógica. Esto es de especial relevancia para el trabajo posterior que exigirá, aún más, remover completamente esta lógica de la aplicación central para pasar a ser parte de los plugins respectivos.

4.2. Simplificación del modelo

Con el objetivo de hacer más sencillo el análisis y centrar el foco en aquellas áreas que son relevantes para el diseño de la aplicación extensible por plugins, en esta sección se eliminará del modelo la administración y control de usuarios, roles y privilegios. Esto, ciertamente, no pretende presentar este problema como trivial. Más aún, para que el diseño pueda realmente ser utilizado por aplicaciones en producción, la comunicación entre la aplicación central y los plugins debiera incluir este elemento, garantizando la correcta autenticación y autorización de los usuarios.

En otras palabras, resolver el problema en cuestión se hace esencial para que lo planteado en este trabajo sea utilizable en la realidad, pero se trata de un problema transversal que puede ser discutido de manera independiente.

La otra simplificación a realizar, de considerable menor relevancia, consiste en eliminar la posibilidad de suscribir mandatos con diferentes divisas. Si bien esta funcionalidad efectivamente se utiliza en la aplicación original (con pesos chilenos, dólares estadounidenses y unidades de fomento), la incidencia que tiene esta funcionalidad en el problema en cuestión es meramente tangencial. En la en la figura 7 podemos apreciar estos cambios y los de la sección 4.1.

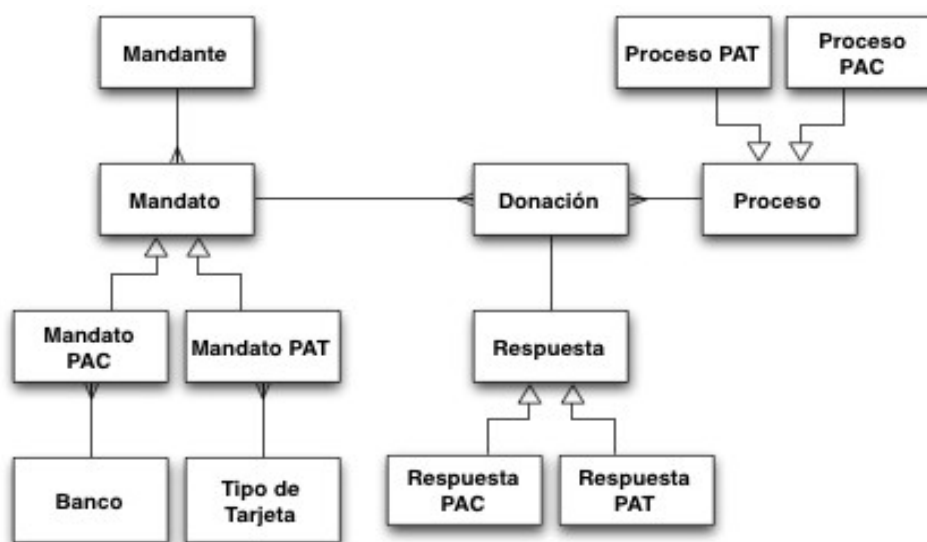


Figura 7: Modelo de clases corregido y simplificado

4.3. Separación entre aplicación central y plugins

4.3.1. Identificación estructural de la aplicación central

A partir del diseño simplificado presentado en la sección anterior, se propone en ésta identificar aquellas áreas correspondientes a la aplicación central, dejando de lado aquellas correspondientes a los diferentes plugins.

El trabajo realizado en las secciones 4.1 y 4.2 ha entregado un modelo de clases en el que ya es posible identificar el área central de manera muy sencilla. Se ve que la construcción de subclasses ha llevado a un diseño efectivamente extensible por la vía de la herencia, ya discutida en la sección 2.1.1. Aprovechando esto, en la figura 8 se presenta el modelo de clases para la aplicación central.

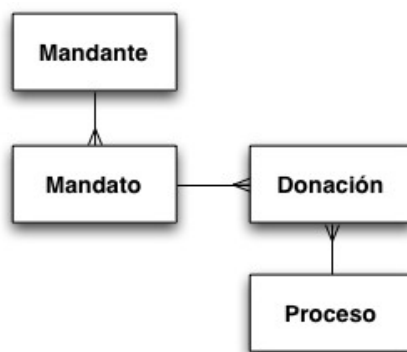


Figura 8: Modelo de clases de la aplicación central

4.3.2. Identificación funcional de los plugins

Las clases dejadas de lado en el modelo central no son suficientes para explicar la estructura que tendrán los plugins: más aún, se obtendría para cada uno cuatro clases completamente disociadas. Esto tiene sentido, pues la aplicación central tomará sólo algunos puntos de entrada de los plugins para trabajar, aún cuando la éstos necesiten ciertas clases para funcionar internamente.

A continuación se presenta un resumen de la funcionalidad de la aplicación desde el punto de vista de las clases. A partir de este resumen será posible identificar cuáles son los recursos que proveerá cada plugin.

- Crear, editar y eliminar datos paramétricos: bancos, tipos de tarjeta de crédito y los código de respuesta de los diferentes tipos de proceso.
- Reencriptar las tarjetas de crédito.
- Crear y editar donantes.
- Crear, editar y eliminar mandatos asociados a los donantes.
- Obtener las donaciones efectivas que han sido cursadas a partir de un mandato.
- Iniciar procesos de donación.
- Realizar gestiones adicionales al proceso de donación, hasta su finalización (el flujo es particularmente dependiente del tipo de proceso).
- Obtener los detalles de un proceso de donación, en cualquier etapa, los que incluyen las donaciones asociadas.

En la tabla 2 se muestran estas funcionalidades clasificadas de acuerdo a cuál área de la aplicación corresponden.

Funcionalidad	Core	PAC	PAT
Administración bancos		X	
Administración tipos de tarjeta			X
Administración respuestas PAC		X	
Administración respuestas PAT			X
Reencriptar tarjetas de crédito			X
Administrar donantes	X		
Administrar mandatos PAC		X	
Administrar mandatos PAT			X
Obtener donaciones de un mandato PAC		X	
Obtener donaciones de un mandato PAT			X
Iniciar, gestionar y obtener detalles de proceso PAC		X	
Iniciar, gestionar y obtener detalles de proceso PAT			X

Tabla 2: Distribución de funcionalidad de la aplicación entre sus componentes

4.4. Construcción de los plugins como servicios web REST siguiendo la ROA

Como se mencionó en la sección 2.2.3, los recursos son entidades “lo suficientemente importantes para ser referenciadas”. A partir de la información presentada en la sección previa, se ha optado por distinguir los siguientes recursos:

- El plugin de un tipo de proceso. Éste corresponde al recurso principal, desde el cual se puede acceder a los demás.
- El conjunto de procesos de donación de un plugin
- Cada proceso de donación en particular
- El conjunto de mandatos asociados al plugin
- Cada mandato en particular
- El conjunto de donaciones asociadas a un mandato
- El conjunto de donaciones asociadas a un proceso de donación
- Cada donación en particular
- El conjunto de parámetros a ser configurados para el plugin
- Cada parámetro en particular

- El conjunto de tareas administrativas asociadas al plugin
- Cada tarea administrativa

Se puede ver que hay un conjunto de recursos homogéneo –plugin, procesos y mandatos– que todos los plugins deberán implementar, pero también está el espacio para la configuración de cada uno de ellos en particular y ciertas tareas administrativas que éstos puedan requerir. Esto muestra parte de la flexibilidad que otorgan los servicios web REST, dado que habrá recursos que sólo serán obtenidos por medio de la navegación y no estarán previamente definidos en el contrato.

4.4.1. URIs y enlaces

Siguiendo la ROA, en la tabla 3 se presentan los recursos que ambos plugins tratan de igual manera, junto a la URI que los identifica y los enlaces a otros recursos. Luego, en las tablas 4 y 5, respectivamente, se presenta esta misma información para los recursos específicos de los plugins PAC y PAT.

Recurso	URI	Enlaces
Plugin	/	<ul style="list-style-type: none"> • Conjunto de procesos • Conjunto de mandatos
Conjunto de procesos	/procesos	<ul style="list-style-type: none"> • Cada proceso individual
Proceso individual	/proceso/{id_proceso}	<ul style="list-style-type: none"> • Listado de donaciones del proceso
Listado de donaciones asociadas a un proceso	/proceso/{id_proceso}/donaciones	<ul style="list-style-type: none"> • Proceso asociado a cada donación • Mandato asociado a cada donación
Conjunto de mandatos	/mandatos	<ul style="list-style-type: none"> • Cada mandato individual
Mandato individual	/mandatos/{id_mandato}	<ul style="list-style-type: none"> • Listado de donaciones del mandato
Listado de donaciones asociadas a un mandato	/mandatos/{id_mandato}/donaciones	<ul style="list-style-type: none"> • Mandato asociado a cada donación • Proceso asociado a cada donación

Tabla 3: Recursos, URIs y enlaces comunes entre plugins PAC y PAT

Recurso	URI	Enlaces
Plugin	/	<ul style="list-style-type: none"> • Conjunto de parámetros • Conjunto de tareas
Conjunto de parámetros	/parametros	<ul style="list-style-type: none"> • Conjunto de bancos • Conjunto de tipos de respuesta
Conjunto de bancos	/bancos	<ul style="list-style-type: none"> • Cada banco individual
Banco individual	/bancos/{id_banco}	<ul style="list-style-type: none"> • Conjunto de bancos
Conjunto de tipos de respuesta	/tipos_respuesta	<ul style="list-style-type: none"> • Cada tipo de respuesta individual
Tipo de respuesta individual	/tipos_respuesta/{id_tipo_respuesta}	<ul style="list-style-type: none"> • Conjunto de tipos de respuesta
Conjunto de tareas	/tareas	Sin enlaces; el plugin no tiene tareas adicionales.

Tabla 4: Recursos, URIs y enlaces específicos del plugin PAC

Recurso	URI	Enlaces
Plugin	/	<ul style="list-style-type: none"> • Conjunto de parámetros • Conjunto de tareas
Conjunto de parámetros	/parametros	<ul style="list-style-type: none"> • Conjunto de tipos de tarjeta • Conjunto de tipos de respuesta
Conjunto de tipos de tarjeta de crédito	/tipos_tarjeta	<ul style="list-style-type: none"> • Cada tipo de tarjeta individual
Banco individual	/tipos_tarjeta/{id_tipo_tarjeta}	<ul style="list-style-type: none"> • Conjunto de tipos de tarjeta
Conjunto de tipos de respuesta	/tipos_respuesta	<ul style="list-style-type: none"> • Cada tipo de respuesta individual
Tipo de respuesta individual	/tipos_respuesta/{id_tipo_respuesta}	<ul style="list-style-type: none"> • Conjunto de tipos de respuesta
Conjunto de tareas	/tareas	<ul style="list-style-type: none"> • Tarea de reencriptación de tarjetas
Tarea de reencriptación de tarjetas	/tareas/reencriptacion_tarjetas	<ul style="list-style-type: none"> • Conjunto de tareas

Tabla 5: Recursos, URIs y enlaces específicos del plugin PAT

Para obtener una mejor visualización de la navegación posible entre los recursos presentados en las tablas anteriores, ésta se presenta de manera gráfica en los diagramas 9, 10 y 11.

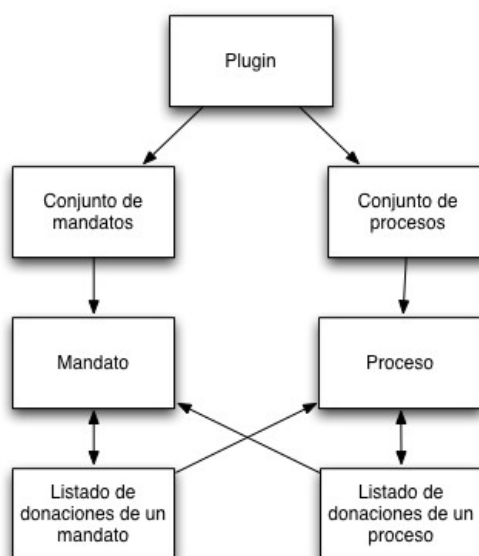


Figura 9: Navegación entre recursos comunes entre plugins PAC y PAT

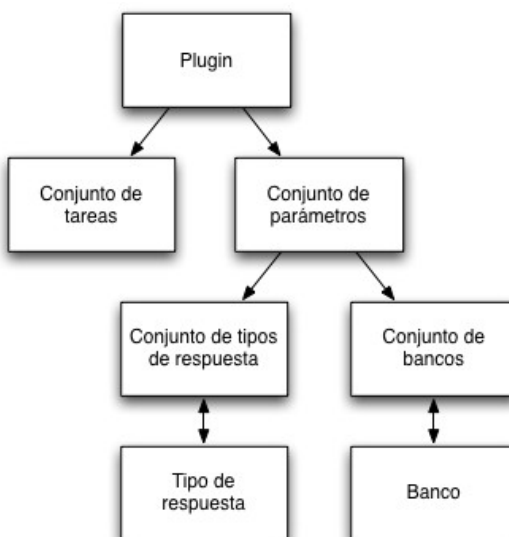


Figura 10: Navegación entre recursos específicos del plugin PAC

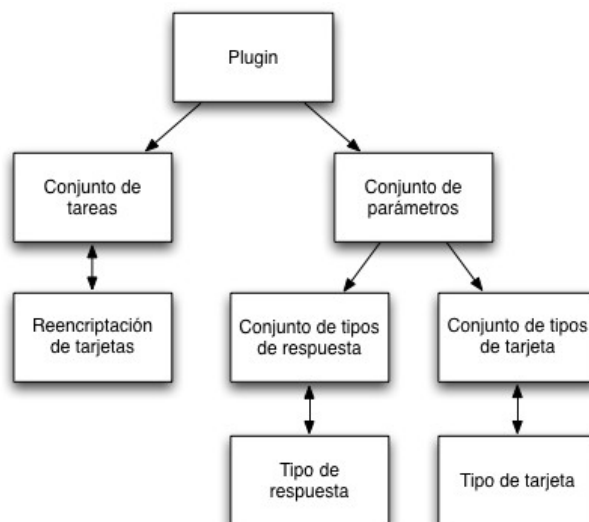


Figura 11: Navegación entre recursos específicos del plugin PAT

4.4.2. Métodos HTTP y representaciones

Siguiendo la metodología propuesta por la Arquitectura orientada a recursos, se definen a continuación, para cada uno de los recursos identificados en la sección anterior, los métodos HTTP que éstos aceptan y la representación de entrada y salida en cada caso. Dado que las representaciones difieren dependiendo del plugin en cuestión, tablas 6 y 7, respectivamente, definen por separado todos los recursos de cada plugin, incluso aquellos que se distinguieron como comunes en la sección anterior.

Para las representaciones de entrada, salvo que se indique lo contrario, siempre se utilizará el tipo de medios `application/x-www-form-urlencoded`, el mismo que utilizan los formularios web tradicionales. Esta opción tiene sentido dado que los plugins proveerán interacción por medio de estos formularios a los clientes.

De igual manera, para las representaciones de salida, se utilizará en todos los casos en que no se indique lo contrario el tipo de medios `application/xhtml+xml`, donde se proveerá la información necesaria con listados y enlaces HTML clásicos y, para la interacción, formularios tradicionales.

Recurso	Método	Representación de entrada	Representación de salida
Plugin	GET	N/A	Nombre y enlaces a: conjunto de procesos, de mandatos, de parámetros y de tareas.
Conjunto de procesos	GET	N/A	Enlaces a cada uno de los procesos existentes.
	POST	Período de cobro	Idéntico a GET de proceso en estado “Por iniciar”.
Proceso	GET	N/A	Período, cantidad de registros comprometidos y efectivos, monto total comprometido y efectivo, fecha de cargo y enlace a listado de donaciones. Además, se agrega una de las siguientes: <ul style="list-style-type: none"> • En estado “Por iniciar”: interfaz para subir BDU. • En estado “Iniciado”: enlace para bajar archivo de cargos e interfaz para subir archivo de rendiciones. • En estado “Finalizado”: enlace para bajar archivo de cargos.
	PUT	Archivo BDU (formato Banco de Chile)	Idéntico a estado “Iniciado”.
		Archivo de rendiciones (formato Banco de Chile)	Idéntico a estado “Finalizado”.
Listado de donaciones de un proceso	GET	N/A	Enlace al proceso y listado con cada donación: <ul style="list-style-type: none"> • Monto efectivo • Respuesta banco • Monto comprometido • Enlace a mandato
Conjunto de mandatos	GET	N/A	Enlaces a cada uno de los mandatos
	POST	Monto, período inicial, período final, banco y número de cuenta corriente.	Idéntico a GET mandato.

Recurso	Método	Representación de entrada	Representación de salida
Mandato	GET	N/A	Monto, período inicial, período final, banco, número de cuenta corriente, fecha de creación, estado activo, enlace a donaciones asociadas e interfaz para su edición.
	PUT	Estado activo (bool)	Idéntico a GET.
Listado de donaciones de un mandato	GET	N/A	Enlace al mandato y listado con cada donación: <ul style="list-style-type: none"> • Monto efectivo • Respuesta banco • Monto comprometido • Enlace a proceso
Conjunto de parámetros	GET	N/A	Enlace a conjunto de bancos y conjunto de tipos de respuesta.
Conjunto de bancos	GET	N/A	Listado de bancos con enlace a cada uno.
	POST	Código, nombre, alineación en output CSV (izquierda o derecha)	Idéntico a GET de banco.
Banco	GET	N/A	Código, nombre, alineación e interfaz para su edición.
	PUT	Código, nombre, alineación en output CSV (izquierda o derecha)	Idéntico a GET.
	DELETE	N/A	Enlace a conjunto de bancos.
Conjunto de tipos de respuesta	GET	N/A	Listado de tipos de respuesta con enlace a cada uno.
	POST	Código, glosa y valor de éxito (bool)	Idéntico a GET de tipo de respuesta.

Recurso	Método	Representación de entrada	Representación de salida
Tipos de respuesta	GET	N/A	Código, glosa, valor de éxito e interfaz para su edición.
	PUT	Código, glosa y valor de éxito (bool)	Idéntico a GET.
	DELETE	N/A	Enlace a conjunto de tipos de respuesta.
Conjunto de tareas	GET	N/A	Vacío.

Tabla 6: Recursos, métodos HTTP y representaciones del plugin PAC

Recurso	Método	Representación de entrada	Representación de salida
Plugin	GET	N/A	Nombre y enlaces a: conjunto de procesos, de mandatos, de parámetros y de tareas.
Conjunto de procesos	GET	N/A	Enlaces a cada uno de los procesos existentes.
	POST	Período de cobro	Idéntico a GET de proceso en estado “Iniciado”.
Proceso	GET	N/A	<p>Período, cantidad de registros comprometidos y efectivos, monto total comprometido y efectivo, fecha de cargo y enlace a listado de donaciones.</p> <p>Además, se agrega una de las siguientes:</p> <ul style="list-style-type: none"> • En estado “Iniciado”: interfaz para bajar archivo de cargos con clave privada e interfaz para subir archivo de rendiciones. • En estado “Finalizado”: interfaz para bajar archivo de cargos con clave privada.

Recurso	Método	Representación de entrada	Representación de salida
	PUT	Archivo de rendiciones (formato Transbank)	Idéntico a estado "Finalizado".
Listado de donaciones de un proceso	GET	N/A	Enlace al proceso y listado con cada donación: <ul style="list-style-type: none"> • Monto efectivo • Respuesta Transbank • Monto comprometido • Enlace a mandato
Conjunto de mandatos	GET	N/A	Enlaces a cada uno de los mandatos
	POST	Monto, período inicial, período final, datos de tarjeta de crédito: tipo, número y fecha de vencimiento;	Idéntico a GET mandato.
Mandato	GET	N/A	Monto, período inicial, período final, tipo de tarjeta, últimos 4 dígitos de la tarjeta, fecha de creación, estado activo y enlace a donaciones asociadas.
	PUT	Estado activo (bool)	Idéntico a GET.
Listado de donaciones de un mandato	GET	N/A	Enlace al mandato y listado con cada donación: <ul style="list-style-type: none"> • Monto efectivo • Respuesta transbank • Monto comprometido • Enlace a proceso
Conjunto de parámetros	GET	N/A	Enlace a conjunto de tipos de tarjeta y conjunto de tipos de respuesta.
Conjunto de tipos de tarjeta de crédito	GET	N/A	Listado de tipos de tarjeta con enlace a cada uno.
	POST	Nombre y formato de número de tarjeta.	Idéntico a GET de tipo de tarjeta.
Tipo de tarjeta de crédito	GET	N/A	Nombre y formato de número de tarjeta e interfaz para su edición.
	PUT	Nombre y formato de número de tarjeta.	Idéntico a GET.

Recurso	Método	Representación de entrada	Representación de salida
	DELETE	N/A	Enlace a conjunto de tipos de tarjeta.
Conjunto de tipos de respuesta	GET	N/A	Listado de tipos de respuesta con enlace a cada uno.
	POST	Código, glosa y valor de éxito (bool)	Idéntico a GET de tipo de respuesta.
Tipos de respuesta	GET	N/A	Código, glosa, valor de éxito e interfaz para su edición.
	PUT	Código, glosa y valor de éxito (bool)	Idéntico a GET.
	DELETE	N/A	Enlace a conjunto de tipos de respuesta.
Conjunto de tareas	GET	N/A	Enlace a reencriptación de tarjetas de crédito.
Reencriptación de tarjetas de crédito	GET	N/A	Interfaz para subir archivo de clave privada actual.
	PUT	Clave privada (formato de clave PKCS)	Enlace para descargar nueva clave privada.

Tabla 7: Recursos, métodos HTTP y representaciones del plugin PAT

4.4.3. Códigos de respuesta

La última tarea para acabar el diseño de los plugins como servicios web REST de acuerdo a la Arquitectura Orientada a Recursos es definir los códigos de respuesta para las diferentes interacciones. Para esta aplicación se ha optado por un subconjunto de los provistos por HTTP.

En los casos normales de ejecución, para las operaciones GET y DELETE se obtendrá código “200 OK”; GET obtendrá la representación solicitada y DELETE un enlace a un recurso padre asociado al que fue borrado. Para la creación de recursos por medio de POST (no se utiliza en ningún caso PUT, dado que nunca se conoce de antemano la URI), el código de respuesta exitoso será “201 Created” junto con una representación que permita trabajar, con el recurso, idéntica a la que obtendrán subsecuentes operaciones GET sobre el mismo.

Para los casos erróneos se utilizará el código “400 Bad Request” para los casos en que la representación de entrada tenga problemas de formato. El código “404 Not Found” se utilizará para los accesos a recursos no existentes. Finalmente, al intentar operaciones con métodos no implementados por los recursos, se utilizará el código “405 Method Not Allowed”. Dado que para este trabajo se está dejando de lado la problemática de la autenticación, no se utilizarán los códigos “401 Unauthorized” ni “403 Forbidden”.

4.5. Uso de plugins desde aplicación anfitriona

En esta sección se presentan las modificaciones necesarias a las clases que componen la aplicación anfitriona.

En primer lugar, se agregará al modelo una clase Plugin, que representará la conexión a uno de ellos. Siguiendo las nociones REST, cada instancia de esta clase sólo contendrá un identificador local y una URI que apunta a la ruta base del plugin referenciado. La única manera de acceder a los demás recursos es por medio de la navegación presentada en la sección 4.4.1. Si bien algunas de estas rutas son estáticas, como los conjuntos de mandatos y procesos, y se pueden obtener a priori sin la navegación, hay otras, como las opciones paramétricas y las tareas, que sólo se pueden obtener navegando en las secciones adecuadas del plugin.

Por otra parte, dado que la información de los mandatos será manejada por los plugins, la clase actual perderá todo almacenamiento de datos y sólo contendrá un identificador local, un identificador del donante al que está asociado y un identificador del plugin correspondiente, desde el cual se obtienen los datos.

Algo similar sucede con la clase de procesos, sólo conteniendo un identificador local y, nuevamente, un identificador del plugin correspondiente.

Las donaciones, por último, sólo se obtendrán haciendo las llamadas correspondientes a los recursos asociados en cada uno de los plugins; la clase dejará de tener respaldo en la base de datos y sólo hará de intermediario con la interfaz REST de los plugins.

Para obtener la comunicación desde la aplicación anfitriona, se intentó en primera instancia utilizar la librería ActiveResource provista por Rails. El objetivo de

ActiveResource, como se mencionó en la sección 2.2.4, es abstraer modelos remotos para trabajar con ellos como si fueran modelos Rails locales. Sin embargo, ActiveResource impone condiciones que lo hacen ideal para modelar recursos que tienen una asociación directa con filas de un esquema relacional, pero que limitan la posibilidad de interactuar con servicios web REST genéricos. Dado el esquema planteado en la sección 4.4 para los plugins, los recursos son más abstractos, proveen más funcionalidad y esperan una navegación provista por la representación entregada, lo que finalmente impidió el uso de ActiveResource.

Como alternativa, se optó por hacer un interpretación manual de las interpretaciones entregadas, utilizando métodos nativos de Ruby para generación de las representaciones de entrada (formato `application/x-www-form-urlencoded`) y la librería XML Nokogiri¹ para la interpretación de las representaciones de salida.

¹ <http://nokogiri.org>

5. CONCLUSIONES

5.1. Logros

El presente trabajo ha tenido por objetivo mostrar la factibilidad de desarrollar una aplicación basada en plugins distribuidos, servidos como servicios web bajo el estilo arquitectónico REST. En esta sección se ahonda en los logros específicos que se han conseguido.

5.1.1. Factibilidad de la solución

En primer lugar, se hace necesario mostrar que el diseño presentado en el capítulo anterior soluciona de manera efectiva el problema originalmente planteado: posibilitar la extensión de aplicaciones web a través de la apertura de esta a plugins distribuidos como servicios web REST. Las aplicaciones así diseñadas pueden ser instaladas una vez y luego ser extendidas por terceros desarrollando a distancia, simplemente añadiendo una URI base.

Un diseño de este tipo da un punto de partida (y no una solución completa, tal como se detallará en la sección 5.2) para el diseño de soluciones informáticas que permitan descentralizar el desarrollo y la extensión de aplicaciones mediante un esquema de distribución controlado sin instalación local, posibilitando el resguardo de algoritmos propietarios, el aprovechamiento de externalidades asociadas a las redes sociales y la inclusión de plugins en aplicaciones web que hoy cumplen funciones que anteriormente eran exclusivas de aplicaciones de escritorio.

5.1.2. REST

En un sentido más específico, el trabajo desarrollado ha permitido demostrar que un arquitectura REST -en particular la Arquitectura Orientada a Recursos- cumple cabalmente con las condiciones necesarias para el desarrollo de servicios web que se comportan como plugins. Los plugins de aplicaciones de escritorio tradicionales se han desarrollado normalmente utilizando APIs orientadas a funciones antes que a esquemas

orientados a objetos. Ante este hecho, se hace particularmente relevante que la solución se base en una arquitectura REST, pues propone un retorno a un enfoque ligado a clases y objetos: la identificación de las entidades relevantes en un sistema y las relaciones entre ellas. Esto posibilita una expresión más rica en términos de vocabulario, y más cercana al dominio del problema, que la se puede lograr con APIs funcionales tradicionales.

Por otra parte, considerando que la meta es la interacción entre aplicaciones que viven en la web, es relevante que éstas se comuniquen aprovechando el potencial completo que ésta provee, en vez de forzarla a funcionar en esquemas procedurales (RPC) como lo hace SOAP.

5.1.3. Rails y ActiveResource

El trabajo, por otro lado, ha permitido mostrar las ventajas de utilizar un framework específicamente diseñado para la interacción REST en vez de adaptar uno a este esquema. En el diseño presentado fue posible percatarse que la mayor parte del código necesario fue escrito sin modificar las opciones por defecto del framework.

La expresividad y flexibilidad provistas por el lenguaje Ruby gracias a la metaprogramación, permiten generar clases en tiempo de ejecución, asunto particularmente necesario en el área donde las clases remotas son definidas dinámicamente a partir de lo entregado por los plugins.

ActiveResource no demostró ser de gran ayuda a la hora de programar la interacción entre la aplicación host y los plugins. Las restricciones de ActiveResource, tendientes a facilitar la interacción entre aplicaciones que siguen las convenciones de Rails, se transformaron en un obstáculo a la hora de obtener representaciones dinámicas, como los datos paramétricos y las tareas de los plugins. Además, la intención de ActiveResource es la presentación de modelos remotos como locales, meta insuficiente a la hora de transmitir, dentro de la representación, elementos como la interfaz de modificación en HTML.

Por otra parte, debido a esto, la solución desarrollada puede ser implementada utilizando cualquier lenguaje, librería o framework, simplemente adscribiendo al contrato REST genérico.

5.1.4. Aplicaciones de escritorio

En línea con lo anterior, cabe mencionar que el esquema propuesto está pensado para aplicaciones que viven "en la web" y no necesariamente son aplicaciones web propiamente tal. En este sentido, una aplicación de escritorio perfectamente puede utilizar el diseño de plugins distribuidos como servicios web REST aquí propuesto, aprovechando todas las ventajas ya descritas en el presente capítulo.

5.2. Áreas por mejorar

Pese a los logros reflejados en la sección anterior, prevalecen aún ciertos elementos que impiden que el esquema sea aplicable en un entorno real y, en particular, que la aplicación desarrollada cumpla con los mismos requisitos a los que responde la aplicación original. Se detallan a continuación los que se han considerado de mayor relevancia.

5.2.1. Esquema de usuarios

Una de los supuestos que permitieron simplificar el trabajo para el modelado de los plugins fue dejar de lado el esquema de usuarios original, tanto en su componente de autenticación (verificación de identidad) como de autorización (quién puede realizar cuáles acciones).

Queda como trabajo pendiente el revisar y determinar cómo incluir un esquema de usuarios que resuelva los mismos problemas de la aplicación original, considerando tanto su administración como la gestión de los roles y privilegios.

En principio, una solución que parece razonable consiste en la creación de un módulo descentralizado de autenticación y autorización que pueda ser utilizado tanto por la aplicación central como por los plugins.

5.2.2. Seguridad de comunicación

La aplicación, tal como se ha propuesto en este trabajo, no toma en cuenta los riesgos de la comunicación de los datos sensibles entre la aplicación central y los plugins. Esta simplificación tiene sentido en tanto el trabajo tuvo por objetivo mostrar las posibilidades de una configuración basada en REST.

A diferencia de lo planteado anteriormente respecto al esquema de usuarios, en esta área ya existen soluciones, fundamentalmente basados en una combinación del uso de HTTPS junto a un intercambio de llaves criptográficas asimétricas, al modo de los servicios web provistos por Amazon.

5.2.3. Tolerancia a fallas y eficiencia de comunicación

En el diseño de la arquitectura de plugins y de comunicación entre los diferentes componentes se dejó de lado la problemática originada cuando éstos se encuentran separados por redes inestables como la Internet.

Uno de estos problemas tiene que ver con la reacción esperada de la aplicación cuando falla la comunicación con los plugins; en el diseño actual la aplicación sencillamente no funciona. Un diseño acabado deberá ocuparse de las posibilidades de falla y de garantizar el funcionamiento de la aplicación central y de todos los plugins que sí están en funcionamiento, evitando idealmente agotar un tiempo de espera de solicitud hacia los plugins que se encuentren offline en cada operación de la aplicación central.

Por otra parte, la comunicación entre la aplicación y los plugins no considera optimizaciones a nivel de caché, realizando las operaciones y entregando las repuestas completas cada vez que son solicitadas. Las operaciones costosas, como el cálculo sobre los montos de donaciones, así como las representaciones de recursos que no cambian, debieran aprovecharse de estas condiciones para proveer un uso más eficaz de los recursos de cómputo y comunicación,

BIBLIOGRAFÍA

- Allamaraju, S. (2008). *Describing RESTful Applications*. Obtenido desde <http://www.infoq.com/articles/subbu-allamaraju-rest> en enero de 2010.
- Booch, G. (2007). *Object-Oriented Analysis and Design with Applications*. 3rd ed. Addison-Wesley.
- Cerami, E. (2002). *Web Services Essentials*. O'Reilly.
- Fielding, R. (2000). *Architectural styles and the design of network-based software architectures*. Tesis doctoral, University of California, Irvine, California.
- Fielding, R. (2008). *REST APIs must be hypertext-driven*. Obtenido desde <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven> en enero de 2010.
- Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
- Hunt, D. y Thomas. A. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.
- Mac-Vicar, D., y Navón, J. (2005). A simple pluggable architecture for business rich clients. In D. Lowe & M. Gaedke (Eds.), *Web Engineering: 5th International Conference, ICWE 2005, Sydney, Australia, July 27-29, 2005: Proceedings* (p. 500–505). Birkhäuser.
- Richardson, L., y Ruby, S. (2007). *RESTful Web Services*. O'Reilly.
- Schach, S. (2004). *Object-Oriented and Classical Software Engineering*. McGraw-Hill.
- Thomas, D. y Heinemeier, D. (2009). *Agile Web Development with Rails*. 3rd ed. Pragmatic Bookshelf.
- Tilkov, S. (2008). *REST Anti-Patterns*. Obtenido de <http://www.infoq.com/articles/rest-anti-patterns> en enero de 2010.
- Vinoski, S. (2008). RESTful Web Services Development Checklist. *IEEE Internet Computing* 12(6), 94–95.