Universidad Nacional de La Plata

Facultad de Informática

Tesina de la Licenciatura

Propuesta de rediseño de la nube de servicios de la UNLP

Carbone, Miguel
Cuesta Luengo, José Nahuel

Directoras: Banchoff Tzancoff, Claudia y Queiruga, Claudia
Asesor Profesional: Rodriguez, Christian Adrián

15 de abril de 2016

Índice

Introducción **1**
Objetivo 2

1. Capítulo I: Contexto **3**
1.1. Historia: ¿Cómo llegamos a dónde estamos? 3
1.1.1. Prefacio: ¿Qué es la nube de servicios de la UNLP? . . 3
1.1.2. El génesis: aplicaciones como islas 6
1.1.3. Primera iteración: eliminando la repetición y normalizando los datos 7
1.1.4. Segunda iteración: haciendo dinámicas las fuentes de datos 10
1.1.5. Tercera iteración: unificando el acceso a la información y desacoplando las componentes 12
1.1.5.1. Arquitectura de servicios web basada en REST 14
1.1.5.2. El diseño final: el Integrador 15
1.1.5.3. El Integrador: implementación 17
1.1.6. Cuarta iteración: este trabajo 18

2. Capítulo II: Marco teórico **21**
2.1. Historia del procesamiento distribuido 21
2.2. Arquitecturas Orientadas a Servicios 23
2.3. La evolución de SOA: Microservicios 25
2.4. ESB 29
2.4.1. El ESB como mediador 31
2.4.2. El ESB como interceptor 32
2.5. REpresentational State Transfer (REST) 33
2.5.1. Derivación de REST 33
2.5.1.1. Estilo base 33
2.5.1.2. Cliente-Servidor 33
2.5.1.3. Sin estado 34
2.5.1.4. Cache 34
2.5.1.5. Interfaz uniforme 35
2.5.1.6. Sistema en capas 35
2.5.1.7. Código bajo demanda 36
2.5.2. Elementos de la arquitectura REST 36
2.5.2.1. Elementos de datos 36
2.5.2.2. Conectores 37
2.5.2.3. Componentes 38

3. Capítulo III: Análisis de tecnologías 39
3.1. Para los servicios 39
3.1.1. Ruby on Rails 40
3.1.1.1. Un framework, muchas librerías 40
3.1.1.2. El costo de Rails 41
3.1.1.3. `rails --api` 42
3.1.2. Sinatra 42
3.1.2.1. Sencillez compleja 43
3.1.3. Conclusión 43
3.1.3.1. En términos de *performance* 44
3.1.3.2. Desde el punto de vista del desarrollador . . 44
3.2. Para la estructura de los servicios 45
3.2.1. HAL 46
3.2.1.1. *Media type* dedicado 46
3.2.1.2. Diseño sencillo y enfocado 46
3.2.1.3. El modelo 47
3.2.1.4. Madurez y actualidad 47
3.2.2. JSON API 47
3.2.2.1. *Media type* dedicado 48
3.2.2.2. Estructura general de los documentos JSON API 49
3.2.2.3. Los recursos 49
3.2.2.4. Obtención de recursos 52
3.2.2.5. Obtención de relaciones 54
3.2.2.6. Inclusión de recursos relacionados 55
3.2.2.7. Selección de campos 55
3.2.2.8. Especificación de orden 56
3.2.2.9. Paginación de colecciones 57
3.2.2.10. Filtrado de resultados 58
3.2.3. Conclusión 58
3.3. Para el nodo central 59
3.3.1. Mulesoft ESB 59
3.3.1.1. Licencia 59
3.3.1.2. Aplicabilidad del proyecto 59
3.3.2. Kong 59
3.3.2.1. Licencia 60
3.3.2.2. Extensiones disponibles (*plugins*) 60
3.3.2.3. Instalación y prueba 63
3.3.2.4. Integración con nuestro diseño 68
3.3.3. API Umbrella 69
3.3.3.1. Licencia 69

3.3.3.2. Arquitectura interna . . . 70
3.3.3.3. Instalación y prueba . . . 72
3.3.3.4. Madurez . . . 76
3.3.4. API Axle . . . 77
3.3.4.1. Licencia . . . 78
3.3.4.2. Estado del proyecto . . . 78
3.3.5. Tyk . . . 78
3.3.5.1. Licencia . . . 79
3.3.5.2. Características principales . . . 79
3.3.5.3. Instalación y prueba . . . 81
3.3.5.4. Integración con nuestro diseño . . . 83
3.3.6. WSO2 ESB . . . 84
3.3.6.1. Licencia . . . 85
3.3.6.2. Características principales . . . 85
3.3.6.3. Instalación . . . 86
3.3.6.4. Integración con nuestro diseño . . . 88
3.3.7. Conclusión . . . 88
3.4. Para la cache compartida . . . 90
3.4.1. Squid . . . 91
3.4.1.1. Instalación . . . 92
3.4.2. Varnish . . . 93
3.4.2.1. Instalación . . . 94
3.4.3. Conclusión . . . 94
3.5. Para balancear la carga . . . 95
3.5.1. Apache . . . 97
3.5.2. NGINX . . . 98
3.5.3. Conclusión . . . 98
3.6. Para documentar . . . 99
3.6.1. RAML . . . 99
3.6.1.1. Licencia . . . 99
3.6.1.2. Estructura . . . 100
3.6.1.3. Herramientas . . . 100
3.6.2. *The OpenAPI Specification* . . . 101
3.6.2.1. Licencia . . . 101
3.6.2.2. Organización . . . 101
3.6.2.3. Herramientas . . . 102
3.6.3. Conclusión . . . 102

4. Capítulo IV: Propuesta de rediseño 104
4.1. Propuesta . . . 104
4.1.1. Redundancia y escalabilidad . . . 105

4.1.2. Desacoplamiento . . . 107
4.1.3. Simplicidad . . . 109
4.1.4. Tolerancia a fallos . . . 109
4.1.5. Estandarización . . . 110

5. Capítulo V: Caso testigo 112
5.1. Caso testigo . . . 112
5.1.1. Alcance . . . 112
5.1.2. Arquitectura . . . 117
5.1.3. Desarrollo del prototipo funcional . . . 121
5.1.4. Experiencia . . . 123
5.1.5. Resultado . . . 124

6. Capítulo VI: Conclusión y trabajos futuros 130
6.1. Conclusión . . . 130
6.2. Trabajos futuros . . . 133
6.2.1. Automatización de la documentación . . . 133
6.2.2. Automatización de la arquitectura . . . 134
6.2.3. Extensión de la gema cliente desarrollada . . . 134
6.2.4. Implementación de pruebas . . . 134

A. Anexo I 136
A.1. Aplicaciones cliente de la nube de servicios de la UNLP . . . 136
A.1.1. Albergue Universitario . . . 136
A.1.2. Asociador . . . 136
A.1.3. Becas UNLP . . . 137
A.1.4. Libretas Sanitarias . . . 137
A.1.5. Licencias Médicas . . . 137
A.1.6. Programa “Mejor Aire” . . . 138
A.1.7. Proyectos de Extensión . . . 138
A.1.8. Recibos de sueldo . . . 139
A.1.9. Responsables . . . 139
A.1.10. Acceso Único (SSO) . . . 139
A.1.11. Sueldos . . . 140
A.1.12. Títulos . . . 140
A.1.13. Dependencias con la nube de servicios . . . 140

B. Anexo II 142
B.1. *Endpoints* de la nube de servicios actual . . . 142

Glosario 148

Introducción

Una de las grandes problemáticas que encontramos a diario en nuestro trabajo como desarrolladores de sistemas informáticos en la Dirección de Desarrollo del CeSPI, Universidad Nacional de La Plata, es el uso y mantenimiento de la nube de servicios que nuestra Dirección ha implementado hace ya más de cuatro años, y que de a poco se ha ido convirtiendo en un obstáculo que retrasa el avance de mejores soluciones integrales.

Nuestro equipo de trabajo esta integrado por más de una decena de profesionales en informática, que estamos a cargo de los diferentes proyectos. En particular, nosotros nos hemos dedicado a investigar soluciones tecnológicas y a realizar el análisis que aquí presentamos con el fin de, en una etapa posterior, continuar con el resto del proceso de rediseño de la nube de servicios, incorporando a otros integrantes del equipo en estas tareas. De manera similar nos hemos propuesto realizar el desarrollo del caso testigo, para luego transmitir la experiencia al resto del equipo.

La implementación actual presenta importantes falencias que dificultan su mantenimiento e incorporación de nuevos servicios. Tanto es así, que ante la necesidad de brindar un nuevo servicio, éste se desarrolla integrado (*dentro*) a la aplicación que produce la información, lo cual genera un alto grado de acoplamiento. Esto se debe a que la arquitectura actual de la nube no permite la incorporación de nuevos servicios que estén fuera de la aplicación monolítica que brinda sus servicios. Entre otras problemáticas que desarrollaremos con mayor detalle en los siguientes capítulos, podemos enumerar la falta de un protocolo estándar de peticiones y respuestas para el acceso a los datos, las inconsistencias presentes, falta de documentación, desactualización tecnológica y falta de un diseño pensado para la escalabilidad.

En esta tesina analizaremos de manera crítica el estado actual de la implementación de la nube de servicios y abordaremos sus problemáticas con una propuesta basada en un enfoque más actualizado, bien fundado y planificado, pensado para adaptarse al constante cambio y crecimiento de los servicios a brindar. Somos conscientes que una capa dinámica de servicios debidamente planificada es crítica en el ecosistema de aplicaciones que desarrollamos en nuestra Dirección y una necesidad impostergable; y es en ese sentido que planteamos la temática para el presente trabajo, el cual será el punto de partida para una reimplementación completa de esta nube de servicios.

Objetivo

Este trabajo tiene como principal objetivo proponer un nuevo diseño para la arquitectura de la nube de servicios para aplicaciones de la Universidad Nacional de La Plata que mejore el que actualmente se encuentra en producción y solucione los problemas que en él encontramos.

Abordaremos nuestra propuesta apuntando a cumplir con los siguientes principios:

- **Escalabilidad:** el diseño debe permitir escalar horizontalmente los nodos involucrados en la provisión de los servicios.
- **Redundancia:** los servicios críticos deben poder tener instancias redundantes para garantizar la máxima disponibilidad posible.
- **Desacoplamiento:** las aplicaciones de gestión de los datos deben separarse de los servicios que proveen esos datos. Como un efecto directo de esto, crecen las posibilidades de escalar los servicios independientemente de las aplicaciones que los utilizan. Junto con la redundancia, esto tiende a eliminar cualquier *Single Point Of Failure* (SPOF) y posibles cuellos de botella en la línea de atención de los requerimientos que la nube reciba.
- **Simplicidad:** tanto el desarrollo como la incorporación de nuevos servicios a los existentes deben ser sencillos. Eliminar todo lo que no sea estrictamente necesario, evitando el bloating de la nube de servicios.
- **Estandarización:** seguir estándares existentes para los distintos puntos de intercambio de información, tanto a nivel de estructura de las respuestas, como de comunicación y autenticación de los servicios. De esta forma, la documentación y posible publicación de los servicios será más sencilla y amigable para los desarrolladores, así como también estará apoyada en definiciones razonables tomadas a partir de la experiencia de *jugadores más grandes* de la industria.

1. Capítulo I: Contexto

Como punto de partida en nuestro análisis comenzaremos por recapitular la evolución histórica de la nube de servicios de la Universidad Nacional de La Plata, de manera tal que pueda comprenderse cómo el crecimiento del conjunto de aplicaciones que desarrollamos fue afectando el enfoque tomado y llevándonos a replantear el diseño que hasta ese momento tenían las fuentes de datos compartidos entre esas aplicaciones para adaptarse a nuevas necesidades.

Este capítulo inicial tiene como objetivo dos puntos principales: contextualizar al lector en el dominio del presente trabajo y, aprovechando ese desglose lógico que haremos para explicar la composición de la nube de servicios, analizar las falencias y los problemas que en ella existen. A partir de las conclusiones de este capítulo, desarrollaremos en los subsiguientes nuestra propuesta para el nuevo diseño de este concentrador de servicios, elemento crítico para nuestras aplicaciones.

1.1. Historia: ¿Cómo llegamos a dónde estamos?

Como desarrolladores y analistas del CeSPI, dependencia de la Universidad Nacional de La Plata encargada de fomentar, implementar y administrar TIC, hemos participado del relevamiento, la implementación, la puesta en producción y el mantenimiento de varias aplicaciones web de uso diario por los agentes de las distintas Unidades Académicas y demás Dependencias administrativas de esta Alta Casa de estudios. Así, en el transcurso de más de 7 años de experiencia, hemos migrado entre distintos paradigmas arquitectónicos en el desarrollo de las aplicaciones y su intercomunicación.

En este capítulo haremos una reseña de lo ocurrido en este tiempo, separando en etapas marcadas por los distintos enfoques que fuimos dando a la problemática de alimentar las distintas aplicaciones que desarrollamos para la UNLP, indicando las razones y decisiones que tomamos en cada ocasión, y que dan origen a este análisis que será la base para la futura etapa de implementación de la nube de servicios integrados.

1.1.1. Prefacio: ¿Qué es la nube de servicios de la UNLP?

Liminarmente creemos conveniente y necesario explicar *qué* es la nube de la que hablaremos en este trabajo. Por simplicidad, y para no develar

detalles técnicos que aún no queremos mencionar, nos centraremos en los aspectos funcionales, en el *qué* y no en el *cómo*, de la fuente de información unificada que utilizamos en las aplicaciones que desarrollamos a diario para la Universidad Nacional de La Plata.

La nube de servicios es un solo sistema web que concentra la información que permite a distintas aplicaciones web, desarrolladas en la Dirección de Desarrollo del CeSPI para uso interno de la UNLP, unificar datos y a partir de esta unificación combinar y correlacionar la información que cada una posee. Contiene y provee datos que van desde identificadores únicos para diferentes tipos de documento (a modo ilustrativo, *"1 equivale a Documento Nacional de Identidad"*, *"2 a Libreta de Enrolamiento"* o *"5 a Pasaporte"*), pasando por valores concretos para identificar las Unidades Académicas o Dependencias de la Universidad (*"33 para la Facultad de Informática"*, *"26 para el CeSPI"*, etcétera), hasta datos concretos de las personas relacionadas a la UNLP (*"0000000000000000000031988189 es José Nahuel Cuesta Luengo, alumno de la Facultad de Informática, docente con dos cargos de dedicación simple en la misma Unidad Académica"*, o *"0000000000000000000027855859 es Miguel Carbone, alumno de la Facultad de Informática, docente con un cargo de dedicación simple en esa UA"*, por tomar dos casos). Mediante los servicios que brinda esta nube se pueden consultar, sin posibilidades de realizar operaciones modificatorias o destructivas, los siguientes grupos de datos:

- Datos de referencia:
  - Tipos de documento
  - Géneros
  - Estados civiles
  - Países, provincias, partidos y localidades
  - Unidades Académicas de la UNLP
- Información académica[1]:
  - Carreras
  - Planes de estudios

[1] Este grupo de servicios será eliminado en el futuro, debido al desacoplamiento de estos servicios de nuestra nube y su delegación en el Grupo de Sistemas Académicos del CeSPI, los reales *dueños* de la información.

    - Materias
    - Títulos otorgados

- Sobre las personas vinculadas a la UNLP (Alumnos y Personal):
    - Datos personales
    - Datos de contacto

- Sobre los cargos del personal de la UNLP (Docentes, No docentes y Autoridades Superiores):
    - Información histórica
    - Quién ocupa el cargo
    - A qué Unidad Académica pertenece
    - En qué situación se encuentra
    - Los recibos de sueldo del cargo[2]

Las aplicaciones que consumen esta información, los *clientes de la nube*, acceden mediante distintos servicios web a los datos que desean. Por ejemplo, un servicio provee todos los tipos de documento que la nube conoce, incluyendo el identificador único de cada tipo de documento y su descripción; mientras que otro servicio provee la información de contacto detallada de un empleado de la UNLP. De esta forma los clientes deben conocer qué servicios brinda la nube y cómo acceder a cada uno de ellos para poder acceder a la información.

Dada la cantidad de aplicaciones que hoy día utilizan los servicios de esta nube para su funcionamiento básico, es de suma importancia -para las tareas que desempeña nuestra Dirección- que su funcionamiento y *performance* sean óptimos, que la dificultad para mantenerla sea mínima y que la tolerancia a fallos o resiliencia de los servicios sea adecuada.

Habiendo hecho esta breve descripción de qué es la nube de servicios, hemos brindado el contexto necesario para comenzar a explicar su evolución, ahora sí incluyendo detalles técnicos sobre su implementación.

---

[2] Si bien este servicio está actualmente activo, ha quedado obsoleto al ser reemplazado por la implementación de una nueva aplicación para los recibos de sueldo que cubre su funcionalidad y elimina su necesidad.

1.1.2. El génesis: aplicaciones como islas

En un principio, cada aplicación funcionaba como un sistema autónomo en su totalidad: no existía comunicación entre los sistemas que estábamos implementando, que hasta ese momento eran relativamente pocos. Cada una definía sus propios datos, tanto los de su dominio particular como aquellos más generales - entiéndase por estos últimos información que categoriza los datos de dominio ya sea georeferenciando las ubicaciones, a las personas por género, por su Dependencia de trabajo o estudio, los documentos de identidad por su tipo, etcétera -. Si bien a simple vista esto puede presentar un claro punto de refactorización para evitar un inminente problema de duplicación y desnormalización de los datos, en ese punto de madurez de los requerimientos que llegaban a nuestra oficina la necesidad no era evidente y mucho menos imprescindible.

El problema no se hizo esperar, al poco tiempo, fue creciendo la necesidad de comunicar las aplicaciones por diferentes razones que se desprendían de los inconvenientes que comenzaban a surgir con el diseño planteado inicialmente para las aplicaciones:

- **Normalización de datos:** las distintas aplicaciones manejaban los datos generales (o de referencia, como los llamaremos de aquí en más) de diferentes maneras, con distintas convenciones, y - lo que es aún más problemático - con diferentes valores concretos para indicar los mismos datos. Por ejemplo, en una aplicación el género femenino era representado con un valor entero `1`, mientras que en otra ése era el valor asignado al género masculino. Otro ejemplo más complejo eran las Dependencias de la Universidad, que en cada aplicación tenían diferentes identificadores y descripciones. Esta falta de normalización en los datos hacía complejo cruzar la información entre distintas aplicaciones y hacía más compleja cualquier actualización necesaria a esos datos de referencia.
  A esto se le suma el mantenimiento de los datos, es decir, siguiendo con el ejemplo de las Dependencias, si era necesario incorporar una nueva, la misma debía ser cargada en cada una de las aplicaciones que utilizaban ese dato de referencia.

- **Unificación de la forma de obtener la información:** este esquema disconexo también acarreaba otro problema oculto en su organización que era la falta de una interfaz unificada de acceso a los datos de referencia. Así como cada aplicación definía sus datos, esto también implicaba

definir el acceso a los mismos, lo cual acababa en tantos métodos distintos de acceso a los datos de referencia como aplicaciones se tenían. Si bien se intentaba mantener un criterio uniforme, las más pequeñas mejoras o personalizaciones en la forma de acceso a un dato de referencia realizadas en una aplicación hacían que ésta fuera diferente del resto.

Por ejemplo, en algunas aplicaciones se implementaban mecanismos opcionales de *caching* para agilizar algunas consultas repetitivas a los datos de referencia, requiriendo de un parámetro específico para indicar si se deseaba o no utilizar esa *cache*; mientras que en otras aplicaciones esta noción no existía, y en su lugar implementaban agregaciones de los datos de referencia diferentes al resto porque la aplicación los necesitaba. Este era claramente un escenario en el que la productividad comenzó a comprometerse, teniendo en cuenta que nuestro equipo de trabajo era relativamente pequeño, en el que la mayoría participábamos en los diferentes proyectos. En ese contexto, el pivoteo de una aplicación a la otra tenía un *overhead* innecesario a la hora de analizar qué intentaba realizar la misma operación de obtención de datos en una u otra implementación.

- **Eliminar la repetición de datos y de código:** como se esbozó en los puntos anteriores, la falta de estandarización y uniformidad de la información se vio reflejada en las diferentes implementaciones de unidades funcionalmente similares (por no decir idénticas). Esto hizo que los diferentes proyectos de las aplicaciones tuvieran diversas implementaciones (a nivel de código) para realizar las mismas tareas, y que los datos de referencia que éstas manejaban se repitieran (aunque con las diferencias antes mencionadas) en cada una.

1.1.3. Primera iteración: eliminando la repetición y normalizando los datos

Ante la creciente cantidad de aplicaciones, los problemas antes enumerados se hacían cada vez más evidentes. Fue entonces que se decidió pasar a un nuevo enfoque sobre el problema: unificar los datos y el código utilizados en las distintas aplicaciones mediante la implementación de clases y objetos reutilizables en ellas.

Este tipo de solución fue relativamente fácil de implementar dada la homogeneidad de frameworks y librerías que nuestras aplicaciones poseían de base. En ese entonces nuestro *stack* de desarrollo estaba principalmente con-

formado por PHP 5.3, el *framework* symfony y bases de datos MySQL, lo cual nos permitió escribir una única vez una librería (o *plugin*, en la terminología del *framework* utilizado) e incluirla en todos los proyectos muy fácilmente. Al centralizar los datos y la lógica de acceso a los mismos en estas clases reutilizables, eliminábamos la repetición de código y datos, y normalizábamos los datos comunes que las aplicaciones utilizaban; y al mismo tiempo simplificábamos el mantenimiento de estas aplicaciones ya que cualquier cambio o solución a un error detectado en las clases de referencia se efectuaba en un único lugar (el *plugin* que las contenía) y se replicaba en las aplicaciones con sólo actualizar la versión del *plugin* disponible en cada aplicación desde nuestro sistema de control de versiones de código[3].

Las clases de referencia consistían en una interfaz común de acceso a la información que éstas contenían y los datos propiamente dichos escritos en el código. A modo ilustrativo, presentamos aquí un extracto de la clase que contenía los tipos de documento, y un ejemplo de uso de la misma:

[3]subversion y git son las dos herramientas para versionar el código de los proyectos que hemos utilizado. El pasaje de subversion a git fue por los beneficios que este último ofrecía en comparación al primero, principalmente el sistema de ramas (*branching*) que utiliza, su esquema descentralizado y el sustancialmente menor tamaño final de los repositorios de código.

```php
<?php

// Clase de referencia para los tipos de documento
class DocumentType {
  // Las constantes representan los valores de los datos de referencia
  const DNI = 1,
        LC = 2,
        LE = 3,
        PASAPORTE = 4;

  // Método utilizado para mostrar los datos con sus descripciones
  static public function getAll() {
    return array(
      DNI => 'DNI',
      LC => 'LC',
      LE => 'LE',
      PASAPORTE => 'Pasaporte'
    );
  }
}

// Ejemplo: asumiendo un objeto $person que admite un tipo de documento
//          se obtiene el identificador del tipo de documento "DNI" y
//          se lo asigna.
$dni = DocumentType::DNI;
$person->setDocumentType($dni);
```

Bloque de código 1: Ejemplo de clase PHP de referencia de la etapa 1 de la nube de servicios

Si bien en principio este acercamiento al problema es altamente beneficioso en comparación a la situación que intenta mejorar, está claramente lejos de ser una solución óptima. En cierto modo, este nuevo enfoque fue el pilar fundamental para la evolución hacia soluciones mejores y más complejas.

El inconveniente con este enfoque era que pese a eliminar la repetición que existía y normalizar los datos, introducía nuevos problemas:

- Si bien los datos de referencia ahora se encontraban unificados a lo largo de todas nuestras aplicaciones, éstos se encontraban *embebidos* estáti-

camente en el código[4]. Cada cambio en la información implicaba lanzar una nueva versión del *plugin* para poder reflejarlo en las aplicaciones.

- Cada actualización en la lógica de obtención de los datos (o en los datos mismos, por lo detallado en el punto anterior) implicaba actualizar todas las aplicaciones que hacían uso de la librería. Este acoplamiento entre las aplicaciones y la fuente de datos de referencia era otro grave problema que tenía esta organización, ya que todas las aplicaciones seguían incluyendo los datos dentro de sí.

1.1.4. Segunda iteración: haciendo dinámicas las fuentes de datos

Luego de la primera iteración en que logramos unificar los datos de referencia, y una vez pasado el período inicial de estabilización de la nueva solución, comenzamos a planificar la siguiente mejora a la forma en que disponíamos de la información: hacer dinámicas las fuentes de datos de referencia.

Si bien el nuevo enfoque hasta este momento subsanaba los inconvenientes que conllevan la repetición y falta de normalización en los datos, éste traía acarreada la poco deseable nueva situación de que los datos de referencia de nuestras aplicaciones eran estáticos y se encontraban escritos directamente en el código. Como se detalló en la sección anterior, esto dificultaba la actualización de cualquier dato en nuestras aplicaciones y acoplaba el código con los datos, lo cual es considerado un antipatrón de diseño de software. Entonces el paso lógico era llevar esos datos a una fuente dinámica, como una base de datos, administrable desde alguna interfaz amigable, a la que las aplicaciones tuvieran acceso y pudieran consultar.

Fue así que una vez más la homogeneidad de nuestros desarrollos nos facilitó la tarea: modificamos nuestro *plugin* de symfony existente para que las clases que antes contenían los datos directamente embebidos dentro de ellas, ahora fuesen abstracciones de tablas en una base de datos dedicada a los datos de referencia. Con este -relativamente sencillo- cambio en nuestro código, la librería común ya soportaba un *backend* dinámico para las fuentes de datos y por ende nuestras aplicaciones daban un salto de calidad al utilizar estos nuevos datos de referencia administrables sin tocar código.

---

[4]En términos más técnicos, nos encontrábamos ante un indeseable caso de *hard-coded data*. Estábamos unificando nuestra lógica de negocios (código) con los datos del dominio, todo escrito en el fuente de nuestra librería.

Así, pese los beneficios obtenidos, apareció un nuevo problema: para poder brindar soporte a esta nueva solución debíamos, para cada aplicación que los utilizace, incluir los datos de conexión a la base de datos de referencia y permitir en nuestra infraestructura que la aplicación tenga acceso a esa base de datos central[5]. Además de estos nuevos requerimientos para cada aplicación, se acarreaban nuevos potenciales inconvenientes:

- Si bien nuestras aplicaciones sólo consultaban la información de referencia que esta base de datos contenía, en caso que los privilegios de acceso a la base de datos fueran demasiado permisivos, se corría el riesgo que cualquier sistema pudiera (accidentalmente o mediante un ataque malintencionado) modificar o borrar la información común a todas las aplicaciones.

- Esta nueva arquitectura ponía a ese nodo central de la base de datos bajo gran stress en momentos que el uso de las aplicaciones se incrementaba, por lo que se debía implementar un mecanismo de *caching* artesanal, local a cada aplicación, que aliviase esa carga. Esta técnica, si bien mitigaba el problema, estaba lejos de ser una solución al mismo, ya que cada aplicación consultaría por separado el mismo conjunto de datos al menos una vez cada cierto período de tiempo, lo almacenaría en su caché local, y manejaría de forma desconexa el tiempo que esos datos se consideraban *frescos*, independientemente del resto de las aplicaciones.

El beneficio obtenido al dinamizar la fuente de nuestros datos de referencia, quitando los datos concretos del código del *plugin* de acceso a los mismos, y simplificando la actualización de esta información de forma independiente a nuestras aplicaciones, fue enorme. Pero esta solución aún no eliminaba por completo el acoplamiento entre nuestras aplicaciones y la fuente de datos de referencia. De hecho, introducía nuevos niveles de acoplamiento al hacer que nuestras aplicaciones deban tener acceso a una base de datos (común) y mantener la información de acceso a ésta; al hacer que las distintas aplicaciones puedan potencialmente modificar esa información común sin que esto sea deseable; al requerir que las políticas de infraestructura permitan la comunicación directa desde múltiples aplicaciones al nodo de la base de datos; al necesitar agregar privilegios de acceso a la base de datos de referencia; y al

[5] Esto implicaba habilitar reglas en firewalls y agregar privilegios a usuarios de la base de datos para conectarse desde distintos equipos.

obligar a las aplicaciones a conocer la implementación interna de cada tipo de dato (su estructura en la base de datos) para poder accederla directamente.

1.1.5. Tercera iteración: unificando el acceso a la información y desacoplando las componentes

En este punto, la cantidad de aplicaciones conectadas a la nube de servicios que habíamos desarrollado había alcanzado prácticamente la decena. Este creciente número de sistemas que debían acceder directamente a las fuentes compartidas de información hacía evidente la necesidad de un nuevo refactor: cada aplicación necesitaba ser mantenida no sólo por su dominio propio sino también ante cualquier modificación realizada a las fuentes de información de referencia, además debíamos tener el cuidado y la conducta de no realizar operaciones de escritura desde ninguna de las aplicaciones satélite de la nube sobre los datos que esta última contiene.

Estas complicaciones adicionales a la implementación de los sistemas propiamente dichos nos dejaban en claro la premisa principal con la cual debíamos replantear el diseño de la nube: *la información debía aislarse, asegurarse y ser de sólo lectura, pero sin hacer más complejo el acceso a la misma.*

Con esa premisa como *leitmotiv*, decidimos centralizar en un lugar los datos que las aplicaciones necesitaban consumir: una única fuente que serviría la información mediante una interfaz web a sus clientes. Con este cambio en el diseño estaríamos cumpliendo tres de los cuatro pilares que guiaban esta etapa del desarrollo:

- La información se encontraría aislada ya que al tener una única aplicación accediéndola (el nuevo proveedor centralizado de información) dejaría de ser necesario que cada aplicación se conecte directamente a la base de datos que hasta ese momento era compartida.
- Al tratarse de una aplicación, asegurar la información sería cuestión de definir un protocolo de acceso a la misma con políticas concretas sobre quién y cómo podría consumirla.
- De manera similar al punto anterior, hacer que el acceso a la información fuera de sólo lectura sería cuestión de no proveer medios para que los clientes realicen escrituras sobre la misma.

Para el punto restante necesitábamos definir la interfaz y el protocolo mediante los cuales las aplicaciones satélite accederían a la información.

Tratándose de aplicaciones web separadas en distintos servidores la forma evidente de implementar la comunicación sería utilizando la web como medio, pero restaba definir cómo dialogarían las aplicaciones cliente con el proveedor para obtener los datos. En nuestra experiencia hasta ese momento habíamos trabajado con Web Services para realizar comunicaciones entre diferentes aplicaciones mediante la web, pero a partir de esa experiencia teníamos nuestras reservas sobre este estándar, principalmente:

- Su implementación nos resultaba excesivamente complicada, al involucrar muchos puntos de acción y acababa siendo propensa a errores humanos. El protocolo general de los Web Services contiene diversos elementos que intervienen en cada parte de la comunicación entre los dos sistemas:
    - El proveedor del servicio utiliza *Web Services Description Language* (WSDL) para describir qué servicios brinda, de qué forma deben accederse, qué formato deben tener los parámetros, y cómo estará estructurada la respuesta. Esta definición de los servicios se mantenía en un documento *eXtensible Markup Language* (XML) que debía ser actualizado cada vez que los datos, puntos de acceso, parámetros esperados y/o la estructura del proyecto cambiaba, agregando un punto más de falla humana al proceso de desarrollo.
    - El proveedor y el cliente se comunican utilizando el framework *Simple Object Access Protocol* (SOAP) para intercambio de mensajes y codifican en documentos XML los mensajes que intercambiarán. Este protocolo de acceso a la información agrega un *overhead* a la comunicación por sobre lo que cualquier comunicación web, basada en el protocolo *HyperText Transfer Protocol* (HTTP)[6], que sabíamos podía ser evitado si utilizásemos otro mecanismo para estos fines.
- Tal como indicamos en el punto anterior, este protocolo agrega pasos que no considerábamos estrictamente necesarios y esto repercutía en los tiempos de acceso a la información. Si tenemos en consideración que ahora tendríamos una aplicación a la que principalmente se accedería para consumir estos servicios, las respuestas deberían ser lo más *magras*

[6]En secciones siguientes desarrollaremos en mayor profundidad las partes principales de este protocolo que atañen al presente trabajo a fin de dar un marco tecnológico concreto a nuestras definiciones.

posibles, eliminando todo consumo innecesario de recursos para generar y transmitir las mismas, y este protocolo no encuadraba en nuestro planteo.

Por estos motivos fue que descartamos la utilización de Web Services como el protocolo de comunicación entre el servidor de la información y los clientes.

Fue entonces que optamos por probar a una alternativa que venía creciendo en popularidad por el último tiempo: las *Application Programming Interfaces* (APIs) *REpresentational State Transfer* (REST). En un sentido general, una API es la interfaz que brinda un programa, librería o framework para que podamos operar programáticamente con su lógica y datos, y si bien es un término amplio, a partir de la tesis doctoral de Roy T. Fielding [6] las APIs REST tomaban un sentido especialmente fundamental en el funcionamiento de las aplicaciones web. Fielding tomó una tecnología existente, HTTP, y utilizó sus elementos para definir un conjunto de principios que las aplicaciones deben cumplir en una arquitectura de sistemas distribuidos hypermedia, asignando a las partes del protocolo HTTP un sentido lógico que encuadraba perfectamente en nuestras necesidades. En la siguiente sección describiremos la concepción informal que tomamos de la idea de servicios REST para definir el diseño que implementamos en esta etapa, que es como hasta hoy está definida la nube de servicios de la UNLP.

1.1.5.1 Arquitectura de servicios web basada en REST

Si bien en ese entonces no hicimos un análisis teórico profundo de la tesis doctoral de Fielding, el concepto de una API REST encuadraba naturalmente en la forma en que una aplicación web funciona y en el mecanismo que utiliza para hacer disponible para otras aplicaciones la información que posee.

Nuestra comprensión de una API RESTful consistía principalmente en tres elementos: *servicios*, *recursos* y *clientes*. Los *servicios* eran partes de una aplicación web (el servidor o proveedor de servicios) que al recibir peticiones respondían con representaciones textuales de los *recursos* (la información) utilizables para luego presentarlos al usuario. Los *clientes*, por su parte, eran los encargados de realizar esas peticiones al proveedor de servicios para obtener los datos que necesitaban y así presentarlos al usuario en el contexto adecuado.

El proveedor de servicios tenía una *Uniform Resource Locator* (URL) base que los clientes conocían, y a través de la cual accedían a sus servicios;

estos servicios se identificaban con una *Uniform Resource Identifier* (URI) relativa a la URL del proveedor de servicios (o endpoint) y ante una petición respondían con un documento *JavaScript Object Notation* (JSON) que representaba la estructura del recurso que el servicio abstraía; y los clientes, las otras aplicaciones web que hacían uso de la nube de servicios, utilizaban esa información en formato JSON para presentar al usuario de una forma amigable los recursos obtenidos del servicio.

Supongamos el siguiente escenario: un alumno de la UNLP accede al sistema web de Becas que se encuentra disponible en `http://becas.unlp.edu.ar`. Por brevedad, simplificaremos el alcance de la solicitud a una supuesta página de la aplicación que provee un listado de las becas disponibles para cada Unidad Académica de la UNLP. Para poder obtener las Unidades Académicas, la aplicación de Becas asumirá el rol de cliente de la nube de servicios de la UNLP (el proveedor de servicios) y realizará una petición al servicio de unidades académicas que la API ofrece. Al recibir esta petición, el servicio de la nube obtiene los datos necesarios para armar el listado de Unidades Académicas y responde al cliente con un documento JSON que representa todos los atributos de las distintas Unidades existentes. Luego, la aplicación de Becas toma esta respuesta del servicio, la trabaja internamente, organiza la información interna que posee de las becas acorde al criterio solicitado y termina por generar el *HyperText Markup Language* (HTML), ya que se trata de una aplicación web, del listado de becas disponibles para cada Unidad Académica en un formato amigable y entendible para el usuario. Visto este caso en un diagrama, las interacciones involucradas son presentadas en la Figura 1.

En la sección 2.5 realizaremos el desarrollo pertinente de la definición de la arquitectura de aplicaciones distribuidas REST que Fielding publicó, dando de esa forma un marco teórico adecuado a las definiciones de nuestra nueva arquitectura para la nube de servicios.

1.1.5.2 El diseño final: el Integrador

Luego de analizar las posibilidades que brindaba el uso de los principios REST en detalle, realizamos el primer diseño de cómo quedarían definidos los distintos servicios o endpoint y decidimos darle un nombre: el *Integrador*,

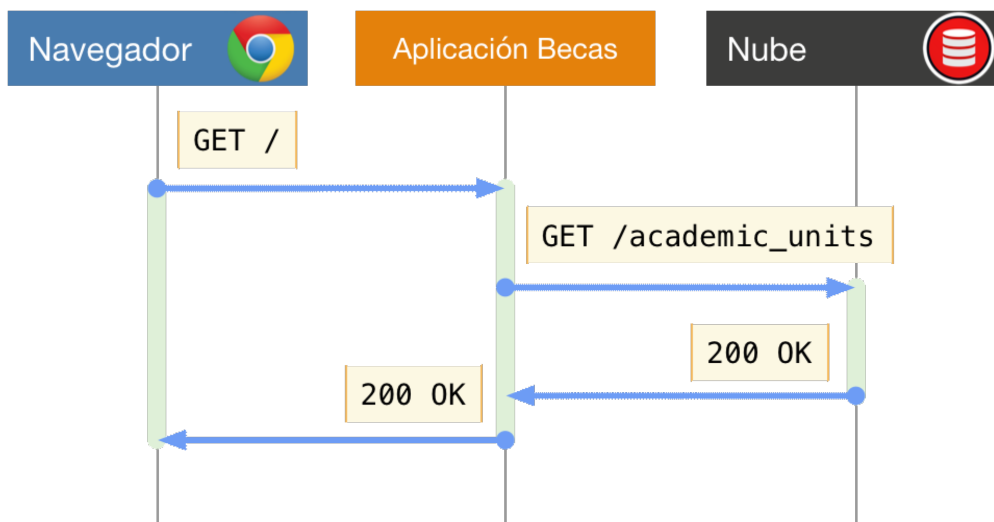


Figura 1: Interacciones involucradas en el listado de becas por Unidad Académica

una mezcla entre título de historieta[7] y *Terminator*, que nos acompaña hasta el día de hoy cuando hacemos referencia a la aplicación *detrás* de la nube de servicios.

Realizando un enfoque más pragmático que correcto para el diseño, definimos servicios que retornan documentos JSON para cada una de las entidades que se podían consultar sin realmente analizar si serían realmente útiles, o si siquiera se accederían alguna vez. Fue de esta forma que la API se compuso de los más de 100 endpoint que tiene hoy, que no siguen una misma línea en su diseño, no tienen una estructura estándar de respuesta, y de los cuales efectivamente se usa tan solo el 46 %[8]. En el Anexo II listamos todos los servicios provistos por la nube para utilizar de referencia.

Sin entrar en detalles sobre cada endpoint, el diseño existente muestra la falta de consistencia general en la definición de las URLs, la sobrecarga de niveles de anidamiento en servicios como el que devuelve las materias del plan de estudios de una carrera de una Unidad Académica[9] que anida

[7]En cierto modo, nos recuerda historietas como Castigador (*The Punisher*, de *Marvel Comics*) – `https://es.wikipedia.org/wiki/Punisher`

[8]Esta cifra surge de los datos que tenemos en nuestra herramienta de analíticas. De los 105 servicios que existen, 56 no registran accesos en el último año.

[9]`/api/academic_unit/:id/career/:career_id/career_programme/:`

esos 3 niveles para mostrar la información, aunque los identificadores únicos de cada elemento podrían usarse, en una estructura más plana, para acceder directamente al último nivel, algo así como `/api/career_programme/:career_programme_id/career_subject.json` y así simplificar las URLs.

Estos problemas sumados a la falta de documentación sobre cómo usar los diferentes servicios, qué parámetros reciben y qué estructura tiene la respuesta cada uno, hacen más complejo el uso de esta API tanto para desarrolladores que la venimos utilizando hace tiempo, como para nuevos integrantes del equipo a los que queremos sumar a alguna aplicación que haga uso de estos servicios.

1.1.5.3 El Integrador: implementación

La nube de servicios, como aplicación, fue un desarrollo más realizado en PHP 5 y el framework symfony 1.4, utilizando MySQL como motor para las 5 bases de datos que utiliza para proveer la información:

- Una base de datos donde se almacena información propia de la aplicación: tokens de acceso, estadísticas de acceso, información de clientes de la API.
- Una segunda base de datos para los datos de referencia de los que ya hemos hablado (tipos de documento, países, etcétera).
- Una tercera base en la que se mantiene actualizada la información que nos llega desde la oficina de Liquidaciones del CeSPI, la cual se transforma mediante procesos *Extract, Transform and Load* (ETL)[10][11] para normalizarla antes de guardarla en esta base de datos.
- Una cuarta para la información que proviene de los SIU Guarani de las distintas Unidades Académicas, la cual es actualizada directamente por el grupo de Sistemas Académicos del CeSPI.

`career_programme_id/career_subject.json`

[10]Técnica que se utiliza para tomar datos de una o múltiples fuentes de información, modificarla y cargarla en una o más almacenes de datos. En nuestro caso utilizamos la herramienta Kettle de la versión de comunidad de la suite Pentaho para transformar los datos fuente, que nos llegan en archivos de texto plano, en inserciones normalizadas en nuestra base de datos MySQL.

- Y una quinta base de datos donde se unifican y mezclan los datos de las últimas 3 bases datos mencionadas, de manera tal que se logre una trazabilidad de la persona como un todo a lo largo de su *vida en la UNLP*, ya sea como alumno, docente o no docente.

La aplicación se realizó sin considerar algunos aspectos claves para mejorar la *performance* del lado de los clientes de la API, como puede ser utilizar cabeceras de *caching*, compresión de respuestas, utilizar *caches* compartidas y otras estrategias que desarrollaremos más en detalle como parte de nuestro análisis para el futuro de la nube de servicios.

Para las aplicaciones satélite, se realizó un *plugin* de symfony que abstraía en clases y objetos la mayoría de los servicios existentes, implementando cierto mecanismo de *caching* local a cada aplicación. Este mecanismo ayudó drásticamente a mejorar los tiempos de respuesta de las aplicaciones, con casos en que presentar una página al usuario tomaba alrededor de 20 segundos sin *caching* y menos de 5 segundos con él. Pero por tratarse de decisiones realizadas meramente del lado de las aplicaciones cliente, existían situaciones en las que la información almacenada en esa *cache* local quedaba desactualizada (*stale*, en inglés) con respecto a lo que la API devolvía como el dato actual (*fresh*, en inglés), por lo que en algunos casos se debió implementar un mecanismo manual de vaciado de *cache* para subsanar estas situaciones.

En resumen: la implementación fue adecuada y funcional para las necesidades del momento, pero con el tiempo esas necesidades y, principalmente, la tecnología fueron cambiando, lo cual fue acrecentando gradualmente la necesidad de un nuevo análisis y replanteo para la solución actual.

1.1.6. Cuarta iteración: este trabajo

Mucha agua ha pasado bajo el puente desde que implementamos la versión actual de la nube de servicios de la UNLP, y varios han sido los cambios que el paso de estos 4 años nos ha dejado: pasamos de ser un equipo de alrededor de 8 personas en que todos nos dedicábamos a desarrollar aplicaciones web en PHP con el framework symfony, algunos toques de JavaScript para la interfaz de usuario y bases de datos MySQL, a ser un equipo de 17 personas que desarrolla aplicaciones en el lenguaje Ruby, con Ruby on Rails y Sinatra como *frameworks* web de cabecera, realizando algunas pequeñas aplicaciones meramente en JavaScript, que ha reescrito en Ruby varias de las aplicaciones realizadas en PHP y mantiene activamente aquellas desarrolladas en symfony que aún no se han migrado, utilizando bases de datos

MySQL mayoritariamente y en algunos casos combinándolas con bases de datos NoSQL y almacenes clave-valor en memoria, como Redis o Memcached. A modo de referencia, en el Anexo I detallamos las aplicaciones que actualmente estamos manteniendo y desarrollando.

Estos cambios han traído aparejada la implementación de un cliente desarrollado en Ruby para integrarlo, de la misma forma que lo hicimos en el caso de las aplicaciones PHP, en las aplicaciones basadas en Ruby on Rails y Sinatra. Este fue otro caso más donde las limitaciones del Integrador actual debieron ser sorteadas mediante la adición de lógica del lado del cliente:

- Las aplicaciones cliente desarrolladas en Ruby implementan una cache condicional basada en Redis o en archivos del filesystem (según disponibilidad, y en ese orden de prioridad), que almacena las respuestas a los requerimientos por un tiempo determinado[11] e intenta subsanar la falta de directivas de cabeceras de parte de la API de servicios.

- Al no tener un estándar para la estructura de las respuestas, la lógica de hidratación de estos documentos JSON para convertirlos en objetos del dominio de las aplicaciones es excesivamente costosa en términos de *performance* y tiene varios chequeos que podrían evitarse si se normalizaran y estandarizasen las respuestas.

- La falta de documentación nos ha obligado en ocasiones teniendo que hacer una suerte de ingeniería inversa de las respuestas para entender relaciones entre datos.

Otro gran problema es la dificultad para escalar que tiene la arquitectura actual. La aplicación web que atiende los pedidos a la nube y sus 5 bases de datos viven en una misma máquina virtual, lo cual puede ser hasta cierto punto conveniente para tener un mantenimiento y administración centralizados, pero esto acota en gran medida la posibilidad de poner rápidamente en funcionamiento nuevas instancias de la API que puedan balancear proporcionalmente la carga ante la creciente demanda que tiene por parte de las aplicaciones cliente. Este único punto de falla también es un potencial

[11] La posibilidad de especificar el tiempo por el cual se desea guardar la copia en cache (el *Time To Live* (TTL)) fue agregada recién en abril de 2015, es decir que antes se almacenaban indefinidamente las copias en *cache*, lo que para los casos en que se usaba el sistema de archivos como almacenamiento esto representaba un potencial problema de crecimiento sin tope de los archivos utilizados para la cache.

riesgo en caso de intrusiones o caídas de cualquier índole: fallas eléctricas, de red, de disco o errores humanos a la hora de realizar el mantenimiento de ese equipo virtual. Las tecnologías que utiliza ya no son mantenidas por sus desarrolladores: PHP 5.3 alcanzó su fin de mantenimiento (o *end of life*, como la comunidad de PHP lo denomina) el 14 de agosto de 2014[12] y symfony 1.4 dejó de ser mantenido en Noviembre de 2012[13], lo cual implica que en cierto modo la aplicación puede eventualmente ser víctima de nuevas vulnerabilidades que se descubran a esa rama del desarrollo y que ya no serán solucionadas.

Todos estos cambios y problemas motivan el presente trabajo, en el cual analizaremos las posibilidades que ofrece una API diseñada desde el comienzo con el nivel más alto[14] de adhesión a REST posible (hypermedia), basándonos en estándares ya establecidos en lugar de intentar reinventar la rueda y definir el nuestro propio para las respuestas JSON, pensando en aprovechar las posibilidades de caching que ofrecen las distintas capas del diseño, e intentando descentralizar la información de manera tal que se elimine el único punto de falla que existe en la actualidad y que nos permita escalar horizontalmente en cantidad de instancias de la nube de manera transparente y poco costosa.

---

[12]Cf. `http://php.net/eol.php`

[13]Cf. `http://symfony.com/blog/symfony-1-4-end-of-maintenance-what-does-it-mean`

[14]Así define Leonard Richardson el conjunto de requerimientos para alcanzar un servicio que cumpla realmente con todos los principios que Roy Fielding definió para una API REST, y define que *Hypertext As The Engine Of Application State* (HATEOAS) es el requerimiento de nivel 3 para alcanzar la pureza de REST. – `http://www.crummy.com/writing/speaking/2008-QCon/act3.html` y claramente analizado por Martin Fowler en `http://martinfowler.com/articles/richardsonMaturityModel.html`

2. Capítulo II: Marco teórico

Este capítulo presenta la base teórica sobre la que nos basaremos a lo largo del resto del informe para fundamentar las decisiones que hemos tomado en nuestro análisis de la arquitectura actual de la nube de servicios de la UNLP.

Estos conceptos son los cimientos de nuestra propuesta de rediseño, la cual trataremos en detalle en la Subsección 4.1, y son los conceptos claves para comprender hacia dónde intentamos llevar el diseño de los servicios brindados por la nube.

2.1. Historia del procesamiento distribuido

Los sistemas mainframe de las décadas de 1960 y 1970, como la serie IBM System/360, raramente se comunicaban entre sí, y cuando lo hacían el proceso de transferencia de información de un sistema a otro era realizado por medio de una cinta magnética. Con el tiempo y ante el creciente número de sistemas dentro de las organizaciones, el acceso en tiempo real entre éstos se hizo cada vez más necesario, tanto dentro de la misma organización como fuera de ella, tal es el caso de los mercados financieros que requerían realizar transacciones en tiempo real.

Inicialmente, el acceso en tiempo real se lograba vía comunicaciones de socket de bajo nivel, usualmente escritas en lenguaje assembler o C, cuya programación era compleja y requería amplios conocimientos en los protocolos de redes. Luego entraron en escena protocolos como *Network File System* (NFS) y *File Transfer Protocol* (FTP), que permitieron abstraerse de la complejidad de los sockets definiendo mecanismos de comunicación que facilitaron el intercambio de información. Estos protocolos dieron pie a abstracciones con mayores posibilidades como *Remote Procedure Call* (RPC), protocolo que permite realizar llamadas a funciones para que sean ejecutadas en un servidor remoto.

En la década de 1980 las computadoras personales habían entrado en escena y los desarrolladores estaban buscando formas más eficaces para aprovechar la potencia de cálculo de estos equipos. Asimismo, el número de servidores dentro de las organizaciones se incrementó exponencialmente debido a la disminución en el precio del hardware. Estas tendencias, junto con la creciente madurez de RPC, impulsaron dos importantes avances en la computación distribuida: *Common Object Broker Architecture* (CORBA) y *Distributed*

Computing Object Model (DCOM), tecnologías que ofrecían herramientas para desarrollar aplicaciones distribuidas en entornos heterogéneos. Estas comunicaciones entre las organizaciones eran caras y dependían de líneas alquiladas con propósitos específicos que formaban circuitos privados, lo cual resultaba práctico solamente para las grandes empresas.

A finales de la década de 1990, con la extendida adopción de Internet las compañías comenzaron a reconocer los beneficios de expandir sus plataformas digitales a socios y clientes, principalmente por la reducción de costos que este medio ofrecía. Desafortunadamente, la utilización de CORBA o DCOM para las comunicaciones en Internet resultó ser todo un reto, en parte debido a las restricciones impuestas por los *firewalls*, que sólo permitían el tráfico HTTP (necesario para los navegadores y comunicaciones con servidores web), y en parte porque ni CORBA, ni DCOM lograron dominar el mercado.

Cuando el protocolo SOAP apareció por primera vez en enero de 2000, fue promocionado como la panacea debido a su dependencia interoperable en XML. SOAP fue concebido principalmente como una alternativa a CORBA y DCOM para realizar llamadas remotas a procedimientos. En este sentido, vale la pena señalar que RPC SOAP era una mejora sobre las implementaciones RPC anteriores, ya que se basó en XML, lo que facilitó un mayor grado de interoperabilidad entre los lenguajes de programación.

Si bien el procesamiento distribuido basado en RPC, fue sin duda alguna una mejora sustancial sobre las comunicaciones basadas en sockets de bajo nivel, éste tenía varias limitaciones[4, p. 6]:

- La alta dependencia entre los sistemas locales y remotos requiere demandas de ancho de banda significativo, existiendo la posibilidad de que una excesiva cantidad de llamadas RPC de un cliente al servidor puedan generar una carga sustancial en la red.
- La naturaleza de grano fino de RPC requiere una red altamente predecible. En este sentido, la latencia impredecible, como es el caso de las comunicaciones sobre Internet, es inaceptable para las soluciones basadas en RPC.
- La compatibilidad de tipos de datos de RPC, que tiene como objetivo proporcionar un soporte completo para todos los tipos de datos nativos (`array`, `string`, `integer`, etc.), dificultó la compatibilización de lenguajes tales como C++ y Java.

Los mensajes RPC SOAP también sufrieron las mismas limitaciones inherentes como las mencionadas anteriormente. Afortunadamente, SOAP ofrece

estilos de mensajes alternativos que superan estas deficiencias.

2.2. Arquitecturas Orientadas a Servicios

La Arquitectura Orientada a Servicios (*Service-Oriented Architecture* (SOA)) establece un marco de diseño para la integración de aplicaciones distribuidas e independientes, permitiendo acceder desde la red a sus funcionalidades que se ofrecen como servicio. Habitualmente SOA es implementado mediante servicios web (*Web Services*), tecnología basada en estándares e independiente de la plataforma que provee los datos, de esta manera SOA puede descomponer las aplicaciones monolíticas en un conjunto de servicios[2].

Existen varias definiciones de SOA, muchas incluyen el término *Web Service*, pero estos conceptos no son lo mismo. SOA es un paradigma y *Web Service* es una forma posible de implementarlo.

Según Thomas Erl[5], SOA establece "un modelo arquitectónico que tiene como objetivo mejorar la eficiencia, agilidad y productividad de una organización mediante la colocación de los servicios como el principal medio a través del cual se realizan los objetivos estratégicos asociados a la comunicación orientada a servicios".

En un sentido similar, para el modelo de referencia *Organization for the Advancement of Structured Information Standards* (OASIS), SOA es un paradigma para organizar y utilizar, capacidades distribuidas que pueden estar bajo el control de dominios diferentes.

SOA incluye prácticas y procesos que se basan en el hecho de que los sistemas distribuidos no son controlados por los mismos propietarios. Diferentes equipos, departamentos o incluso diferentes organizaciones pueden gestionar estos sistemas distribuidos. Este concepto es clave para entender la idea de la Arquitectura Orientada a Servicios y los grandes sistemas distribuidos.

En el pasado se han propuesto una gran cantidad de métodos para resolver el problema de la integración de sistemas distribuidos mediante la eliminación de la heterogeneidad, pero la experiencia ha demostrado que estos enfoques no funcionan[12, p. 14]. Los sistemas distribuidos de mediana a gran escala, suelen tener diferentes propietarios y ser heterogéneos. El enfoque que sigue SOA *acepta* esta heterogeneidad, similarmente a como los métodos ágiles de desarrollo de software aceptan que los requisitos cambian en lugar de tratar de luchar contra esos cambios. Es una idea utópica considerar que se pueda introducir SOA diseñando todas las partes involucradas en los sistemas distribuidos desde cero: SOA se basa en el principio de que

hay que lidiar con el hecho de que la mayoría de los sistemas legados que se encuentran en producción, se mantendrán[12, p. 15].

A continuación enumeramos los 9 principios fundamentales de SOA:

- Contratos de servicio estandarizados: la interfaz de un servicio (su contrato con el cliente) tiene que estar explícitamente declarado. Los campos que forman parte de este interfaz deben estar correctamente tipados y ser conocidos. Con la ayuda de los estándares como WSDL y *XML Schema* (XSD), el contrato del servicio es autodescriptivo.

- Servicios con bajo acoplamiento (*Loose Coupling*): hace referencia al nivel de dependencia entre servicios. Cuanto menos acoplamiento, mayor independencia para el diseño del servicio y su posterior evolución.

- Abstracción de servicio: este principio pone el énfasis en ocultar los detalles internos del servicio. Indica que debe comportarse como una caja negra y estar únicamente definido por su contrato, que a su vez es el mínimo acoplamiento posible con el consumidor del mismo.

- Reusabilidad de servicio: la arquitectura SOA no busca la sustitución de las lógicas de negocio actuales sino que proporciona una forma de reaprovechar todos estos activos encapsulándolos en servicios para que a su vez puedan ser reutilizados por otros servicios.

- Autonomía de servicio: este principio indica que el servicio tiene un alto grado de control sobre su entorno de ejecución y sobre la lógica que encapsula.

- Servicio sin estado: el tratamiento de la información del estado afectaría gravemente a la escalabilidad del servicio, poniendo en riesgo su disponibilidad. Idealmente, todos los datos que necesita el servicio para trabajar provienen de los parámetros de entrada.

- Capacidad de descubrimiento de servicio: al servicio se lo dota de metadatos, gracias a los cuales puede ser descubierto de manera efectiva. Estos metadatos pueden ser interpretados de manera automática pudiendo ser reutilizados. Para ello es necesario disponer de un mecanismo de descubrimiento (llamado registro de servicios) como por ejemplo el *Universal Description Discovery and Integration* (UDDI)).

- Composición de servicios: define la capacidad de un servicio (servicio básico) para formar parte de un servicio más complejo. A medida de

que la arquitectura SOA se consolide, los nuevos servicios (de más alto nivel) podrán implementarse a partir de los servicios de más bajo nivel ya existentes.

- Interoperabilidad de servicios: cada uno de los principios anteriores contribuye a la interoperabilidad de alguna manera. En las arquitecturas SOA, el problema de la falta de esta cualidad es uno de los más importantes. Hay que tener en cuenta que muchos de los servicios que intervienen se implementan con una tecnología diferente, incluso con un sistema operativo distinto. Por ejemplo, se puede tener un servicio realizado en Java ejecutándose sobre Linux que invoca a otro implementado en .net corriendo en una máquina con Windows.

2.3. La evolución de SOA: Microservicios

Como vimos en el apartado anterior, SOA define qué principios debe cumplir un sistema distribuido para satisfacer los objetivos de la organización, que incluyen, facilidad y flexibilidad en la integración de sistemas legados, reducción en los costos de implementación y ágil adaptación a los cambios. Si bien esto ya establece precondiciones para un diseño, SOA en sí no es un patrón concreto. Por el contrario, y similarmente a otros diseños de arquitectura de sistemas distribuidos como REST (el cual será tratado en la Subsección 2.5), SOA es un conjunto de propiedades que dichos sistemas debieran cumplir y a partir de los cuales podemos observar características comunes entre aquellos que sigan ese tipo de arquitecturas.

Es sobre la base de características de un tipo de sistemas como SOA que se construyen luego los patrones de diseño. A modo de referencia se puede consultar el libro *"SOA Patterns"* donde se definen más de 20 patrones orientados a servicios, como *Service Host*[15, p. 19], *Composite Front End (Portal)*[15, p. 148] o *Service Bus*[15, p.162]. Todos estos (y otros) patrones obedecen a los principios de las Arquitecturas Orientadas a Servicios, pero ninguno *es* SOA en sí mismo, ni SOA *es* sólo uno de estos patrones. De hecho, si analizamos el caso particular de estos patrones podemos observar que inclusive pueden combinarse entre sí: el primero (*Service Host*) guía en cómo se puede manejar el ambiente en que corren los servicios y los clientes de los mismos para simplificar su gestión, el segundo (*Portal*) habla de cómo proveer una interfaz que combine más de un servicio, y el tercero (*Service Bus*) propone la inclusión de un canal de comunicación previo a los servicios para gestionar las peticiones y respuestas con mejoras sobre la autogestión de solicitudes por parte de los propios servicios.

En este contexto es que queremos centrarnos en un patrón de diseño de sistemas distribuidos emergente que se plantea como *la* alternativa al desarrollo de aplicaciones monolíticas: el patrón de arquitectura de *microservicios*. Quizás el concepto más importante para entender este patrón, es el de *service component*: la unidad básica de agrupamiento lógico de los servicios. En lugar de pensar en servicios para esta arquitectura, es mejor pensar en *service components*, que pueden variar su granularidad, desde un simple módulo a gran parte de una aplicación.

Las aplicaciones monolíticas típicamente consisten en componentes fuertemente acoplados, que son parte de una única unidad desplegable, resultando poco práctica y dificultosa la incorporación de cambios, *testing* y despliegue de la aplicación. Es por esto que las grandes organizaciones de IT con aplicaciones de estas características, suelen manejarse con ciclos *mensuales* de despliegue para sus productos.

El patrón de microservicios trata estas cuestiones, separando la aplicación en múltiples unidades desplegables (los *service components* antes mencionados), que pueden ser desarrolladas, testeadas y desplegadas independientemente de otros *service components*[8, p. 27].

Como puede observarse en la Figura 2, todas las solicitudes de los clientes son realizadas a través de una capa denominada *user interface layer*, para acceder finalmente a los *service components*.

Debido a que los componentes principales de una aplicación se dividen en partes más pequeñas y desplegables de manera individual, las aplicaciones construidas utilizando el patrón de arquitectura de microservicios son generalmente más robustas, proporcionan una mejor escalabilidad y pueden dar soporte a una entrega contínua (*continuous delivery*), permitiendo actualizar los ambientes de producción mediante despliegues en tiempo real, reemplazando la necesidad de hacerlos mensualmente o semanalmente. [8, p. 33] Todo esto es posible, dado que el cambio generalmente se encuentra aislado a un *service component*, y sólo las unidades que cambian tienen que ser actualizadas. Como se mencionó anteriormente, ésta es una notable mejora frente al desarrollo de aplicaciones monolíticas, donde el fuerte acoplamiento en sus componentes deriva en aplicaciones *frágiles* que suelen fallar con cada nuevo despliegue realizado.

Si bien existen decenas de formas para implementar el patrón de arquitectura de microservicios, las tres principales topologías son: *API REST-based*, *application REST-based*, y *centralized messaging*.[9, p. 29]

Según lo investigado, entendemos que será necesario implementar la to-

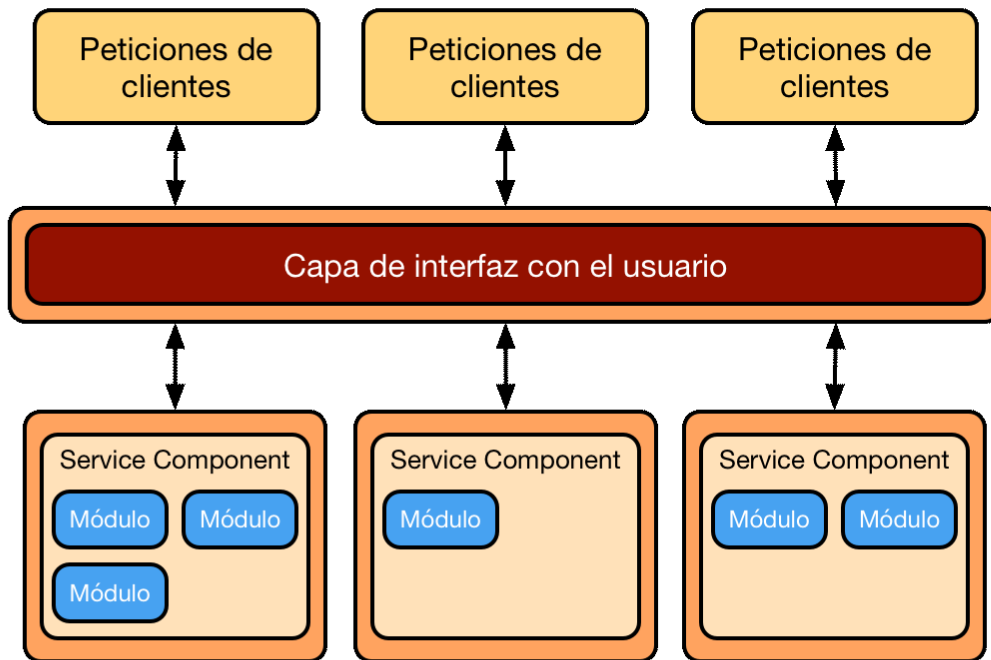


Figura 2: Arquitectura básica de microservicios

pología *centralized messaging* presentada en la Figura 3, la cual se adecúa a nuestra necesidades. Esta topología es similar a la topología *application REST-based*, sólo que en lugar de utilizar REST para acceder remotamente a un *service component*, utiliza un *message broker* centralizado y liviano. No hay que confundir esta topología con el patrón SOA o considerarlo como un "SOA-Lite". Este *message broker* no realiza orquestración, tranformación ni ruteo complejo, más bien, es sólo una capa de transporte ligera, para acceder a los *service components* remotos. Los beneficios obtenidos con esta topología son: mecanismos de colas, mensajería asíncrona, monitoreo, control de errores, *rate limit*, seguridad, balanceo de carga y escalabilidad.

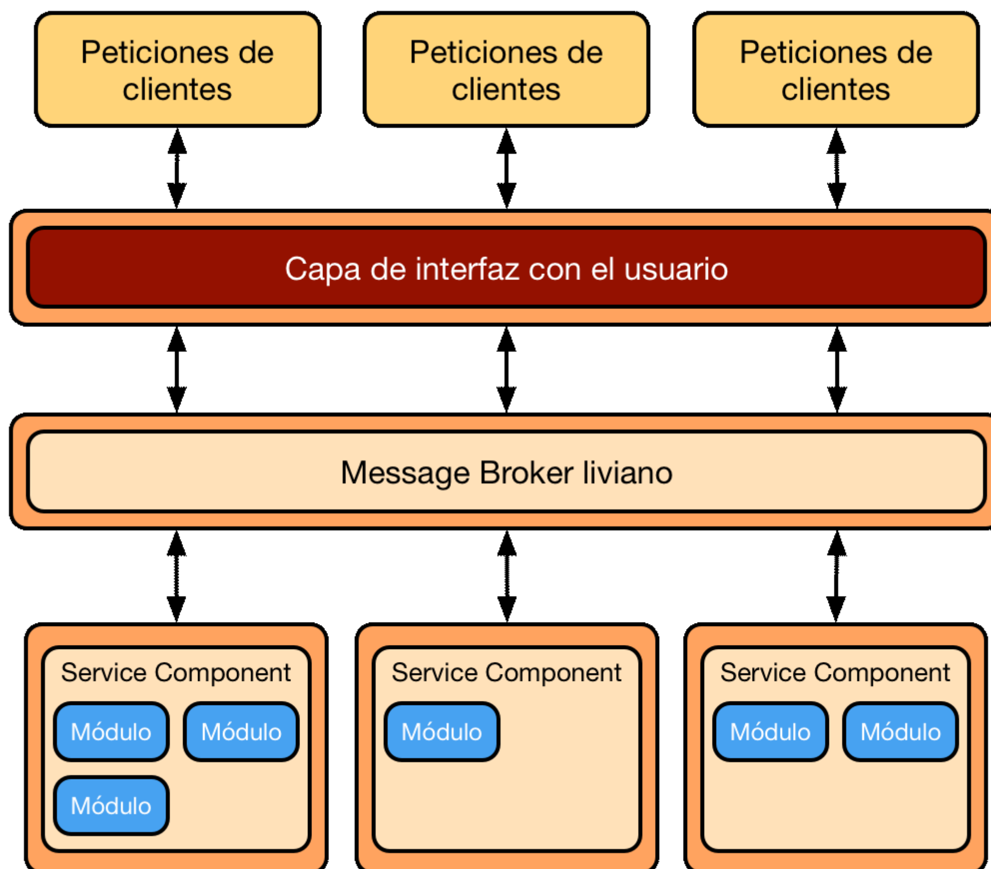


Figura 3: Arquitectura de microservicios con mensajería centralizada

Como se puede observar en la Figura 3, existe un punto único de falla y cuello de botella en esta arquitectura. El mismo puede ser tratado implementando un cluster de brokers, dividiendo el único broker en múltiples instancias y repartiendo la carga en cada una de éstas.[8, p. 31]

En [8, p. 34] se realiza un análisis de las siguientes características para el patrón de arquitectura de microservicios:

- **Respuesta al cambio:** el autor indica que este patrón, tiene un alto grado de *overall agility*, que es la habilidad de responder rápidamente a los constantes cambios en el entorno. Dado que es posible desplegar unidades de forma separada, los cambios se encuentran aislados a *service components* individuales, lo que permite despliegues sencillos y rápidos.
  Además, aplicaciones desarrolladas con este patrón, tienden a generarse con bajo acoplamiento, lo que ayuda en la incorporación de los cambios.

- ***Testability*:** dada la separación y aislamiento de las funcionalidad de negocio, las pruebas de regresión para un *service component* en particular, resultan más sencillas y factibles que las mismas pruebas realizadas a toda una aplicación monolítica. Además, ya que los *service components* en este patrón están débilmente acoplados, hay menos posibilidades de que un desarrollador "rompa" la aplicación completa debido a la incorporación de un cambio reciente.
- **Rendimiento:** en general este patrón no aporta en el desarrollo de aplicaciones de alto rendimiento, debido a la naturaleza distribuida de la arquitectura de microservicios y el overhead que inherentemente agregan esas comunicaciones entre las unidades distribuidas.
- **Escalabilidad:** las aplicaciones se encuentran separadas en unidades desplegables donde cada *service component* puede escalarse individualmente, permitiendo afinar la escalabilidad de acuerdo a las necesidades de la aplicación lo cual resulta en una gran versatilidad para escalar partes de o un sistema completo.
- **Facilidad de desarrollo:** como la funcionalidad se encuentra separada en diferentes *service components*, el desarrollo se vuelve más sencillo debido a un alcance más pequeño y aislado de cada unidad.

2.4. ESB

Uno de los 9 principios de diseño de SOA que mencionamos antes es el bajo acoplamiento (*loose coupling*) de los servicios, y una de las formas más comunes de implementarlo es mediante un *Enterprise Service Bus* (ESB): un medio de comunicación que conecta y abstrae a proveedores y consumidores de los servicios.

Un ESB no implementa en sí mismo una arquitectura orientada a servicios, sino que provee las características mediante las cuales puede implementarse. Es decir, proporciona una capa de abstracción para los *endpoints*, de esta manera se consigue flexibilidad y fácil conexión entre los servicios.

Existen diferentes opiniones acerca del rol exacto y responsabilidades de un ESB, ésto se debe principalmente a la existencia de diferentes aproximaciones técnicas para realizar un ESB[12, p. 47]. En función de los enfoques técnicos y de organización adoptados para la aplicación del ESB, éste puede implicar una o más de las siguientes tareas:

- Proveer conectividad.
- Transformar la información.
- Ruteo (*Routing*) inteligente.
- Manejo de aspectos de seguridad.
- Lidiar con la fiabilidad de los servicios.
- Manejo de los servicios.
- Monitoreo y registro de actividades (*logging*).

El rol principal de un ESB es proveer interoperabilidad. Debido a que integran diferentes plataformas y lenguajes de programación, una parte fundamental de esta función es la transformación de datos. Otra tarea fundamental de un ESB es el ruteo, ya que debe existir alguna manera de acceder desde un consumidor a un proveedor de servicios y luego se debe poder enviar la respuesta de regreso desde el proveedor hacia el consumidor. Dependiendo de la tecnología utilizada y el nivel de inteligencia proporcionado, esta tarea puede ser trivial o puede tornarse compleja.

Hay que tener en cuenta que no existe requerimiento alguno para que el dominio del ESB sea homogéneo. Aunque podría ser mejor usar una sola tecnología para la implementación de los servicios, raramente es el caso, y como se mencionó anteriormente, SOA por su propia naturaleza acepta la heterogeneidad. Eso incluye la heterogeneidad en middleware y protocolos. Incluso con un estándar como los *Web Services*, múltiples implementaciones pueden diferir entre sí. Tarde o temprano, se introducirá un nuevo estándar o una nueva versión del estándar que hace las cosas mejor y más fácil. Tan pronto como empiece a utilizar el nuevo estándar (junto a la antigua tecnología), el ESB se volverá heterogéneo[12, p. 49].

Idealmente, el cambio de tecnología en el ESB, no debe tener ningún impacto (siempre y cuando respete las interfaces existentes) en los proveedores y consumidores, quienes deben ser capaces de utilizar la misma API.

Con la aparición de estándares como SOAP, se establecieron bases para las aplicaciones altamente interoperables mediante *Web Services*. Este tipo de tecnologías abrieron muchas posibilidades pero también trajeron nuevos desafíos. Uno de ellos es la proliferación de comunicaciones punto a punto entre sistemas, que a menudo conduce a un modelo de integración llamado “plato de espaguetis” (ejemplificado de manera gráfico en la Figura 4), con

relaciones muchos a muchos entre diferentes aplicaciones. Si bien el problema de la interoperabilidad entre las aplicaciones se resolvía, se dificultaba su mantenimiento.[3, p. 4].

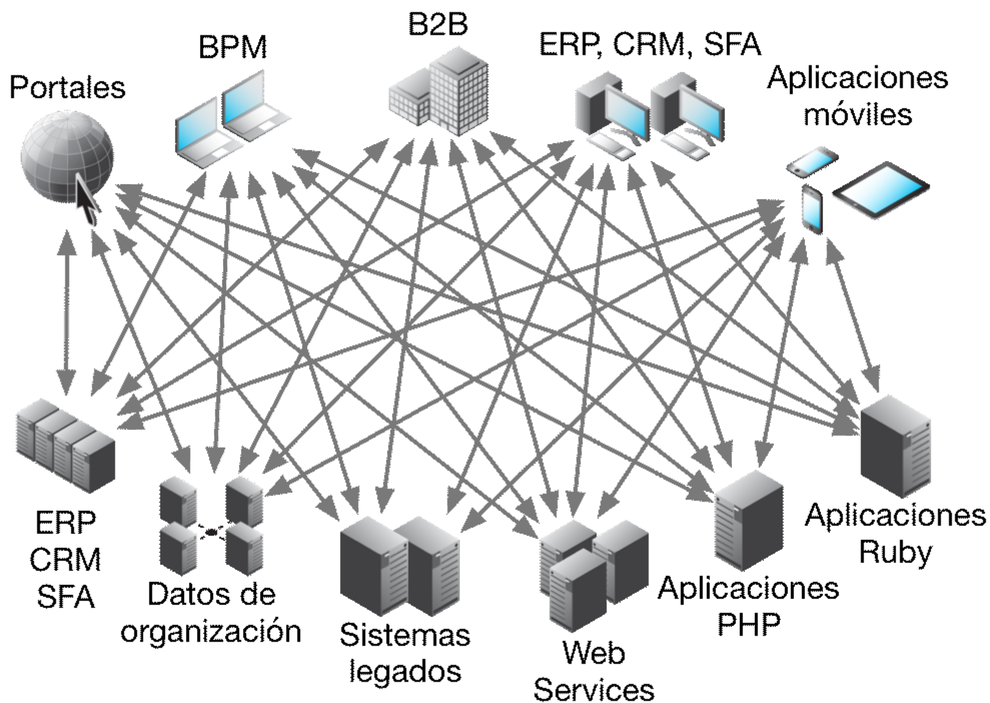


Figura 4: Diagrama ejemplificando la integración punto a punto

2.4.1. El ESB como mediador

En una arquitectura en la que se implemente un ESB, las aplicaciones se comunican a través de este bus central, que actúa como *message broker* entre ellas. Este esquema arquitectónico se llama “mediación” (*Mediation*). De esta manera se minimiza el número de conexiones punto a punto que se necesitan para permitir las comunicaciones entre aplicaciones. Al reducir el número de puntos de contacto entre las diferentes aplicaciones, se simplifica el proceso de mantenimiento y actulización de un sistema, decrementando el grado de dependencia directa que pueda existir entre cada instancia.

2.4.2. El ESB como interceptor

Otra manera en que un ESB basado en un protocolo de punto a punto puede proporcionar indirección a las llamadas de servicio, es con los llamados *interceptors* o *proxies*. Estos elementos, que forman parte del ESB, se ubican por delante de los servicios existentes y delegan en ellos las peticiones, pero sólo luego de procesarlas y decidir la mejor forma de tratarlas. Un enfoque sencillo es reemplazar el *endpoint* que ofrece un servicio, con un balanceador de carga. Los consumidores siguen utilizando el mismo *endpoint*, donde se delega la verdadera tarea, sólo que cuando los mensajes llegan, el interceptor los distribuye a las diferentes instancias que proveen ese servicio[12, p. 52]. Este mecanismo puede apreciarse en la Figura 5.

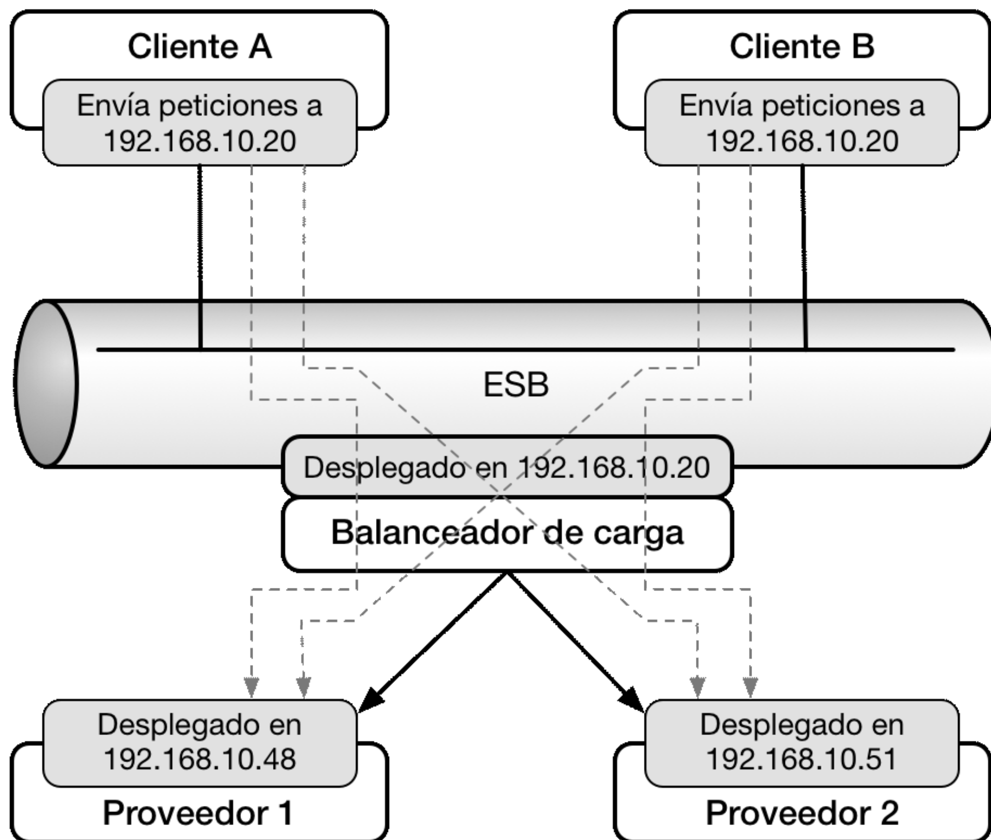


Figura 5: Un ESB como interceptor balancea la carga de los proveedores de servicios

2.5. REpresentational State Transfer (REST)

En el capítulo 5 de su tesis doctoral, Roy Fielding define el estilo de arquitectura de las aplicaciones web o, en términos más generales, de los sistemas distribuidos basados en hypermedia. Esta arquitectura, a la que denomina REST, está presentada como una derivación a partir de las restricciones de interacción que impone, partiendo de un estilo base sin restricciones, al que va agregando otras limitaciones hasta llegar a definir el modelo REST. Al realizar la caracterización de esta forma, Fielding logra de forma natural analizar qué propiedades surgen a partir de cada restricción aplicada.

En esta sección desarrollaremos sintéticamente los elementos de REST de manera tal que podamos utilizarlos como marco teórico en el desarrollo de nuestra propuesta para la nueva nube de servicios de la Universidad Nacional de La Plata, complementando lo que se ha expuesto en la Subsección 2.2.

2.5.1. Derivación de REST

En este apartado resumiremos el proceso de derivación que realiza Fielding para, a partir de un estilo base al que se le van agregando restricciones o propiedades, llegar a definir el estilo de aplicaciones distribuidas REST.

2.5.1.1 Estilo base

La derivación de REST parte de un estilo de sistemas distribuidos base en el que no existen restricciones ni límites entre los elementos que lo componen.

2.5.1.2 Cliente-Servidor

Las primeras restricciones aplicadas al estilo base son aquellas del estilo Cliente-Servidor[6, Sec. 3.4.1], en el que un componente *servidor* ofrece un conjunto de servicios y escucha peticiones a los mismos, mientras que un componente *cliente* realiza solicitudes indicando que desea se realicen esos servicios a través de un *conector*. Ante la recepción de la petición, el servidor rechaza o realiza lo solicitado y envía en retorno una respuesta al cliente. Estas adiciones al conjunto, inicialmente vacío, de limitaciones beneficia al principio de separación de responsabilidades y permite que los componentes cliente y servidor evolucionen independientemente, habilitando a una gran escalabilidad de esta arquitectura.

2.5.1.3 Sin estado

Tomando ahora como base el estilo Cliente-Servidor, se agrega la restricción de que el servidor no puede contener información del estado de la sesión[6, Sec. 3.4.3]: las peticiones realizadas al servidor deben incluir toda la información necesaria para reflejar el estado actual de la comunicación.

De esta forma, se obtienen nuevas propiedades positivas para el estilo en evolución: visibilidad, confiabilidad y escalabilidad. El sistema gana en visibilidad ya que dado un requerimiento, no se necesita ningún dato adicional para conocer su naturaleza; es más confiable ya que recuperarse, al menos en forma parcial, de un fallo implica simplemente retomar el requerimiento que falló; y la falta de información de estado del lado del servidor, simplifica su implementación y permite escalar fácilmente instancias del servidor que puedan atender esos requerimientos. Como contrapunto, este conjunto de restricciones tiende a ser penalizado con una performance disminuida sobre las comunicaciones de red, al necesitar incluir y repetir la información del estado en cada petición.

2.5.1.4 Cache

A fin de disminuir las penalidades en performance que trae aparejadas el modelo *Sin estado* definido en el apartado anterior mejorando la eficiencia en el uso de la red, se agrega la restricción del uso de *cache* de respuestas[6, Sec.3.4.4]. La cache funciona como intermediario entre el cliente y el servidor, encargándose de almacenar las respuestas *marcadas como cacheables* por el servidor de manera tal que peticiones posteriores equivalentes a una anterior con estas características puedan obtener la misma respuesta sin llegar a consultar al servidor.

Esta adición puede eliminar parcial o inclusive completamente algunas interacciones entre cliente y servidor, mejorando la eficiencia de uso de recursos de red y la performance perceptible desde el punto de vista del usuario.

Si bien este punto de la tesis de Fielding hace referencia a la cache únicamente del lado del cliente, nosotros tomaremos una postura más general, en la que la Cache puede ubicarse tanto del lado del cliente como del servidor, para poder incluir en este estilo a las cache compartidas o cache de *gateway*. Este tipo de caches funcionan más cercanas al servidor y permiten que, sin importar si los clientes implementan mecanismos de caching de respuestas, se reduzca la cantidad de requerimientos que efectivamente llegan a ser procesadas por el servidor, al almacenar en la cache las respuestas marcadas

como almacenables (o *cacheables*) e interceptar las peticiones que los clientes realizan, respondiendo con las respuestas cacheadas que dispongan en caso de aplicar. Las caches de gateway también permiten alivianar la carga del servidor cuando múltiples clientes realizan peticiones repetitivas.

El posible problema de agregar la cache al estilo de arquitectura anterior es que existe la posibilidad de que un recurso cacheado esté desactualizado (*stale*, como se lo referencia comúnmente en inglés) con respecto a la respuesta que el servidor proveería si esa petición le llegase.

2.5.1.5 Interfaz uniforme

Uno de los puntos centrales que aporta REST en su definición es la importancia que da a la necesidad de una interfaz uniforme entre los componentes de la arquitectura. Mediante la generalización del diseño del conector utilizado en las comunicaciones y la forma en que se transfiere la información, se simplifica la arquitectura general, a costo de no hacer el uso más eficiente posible de la forma de transmitir los datos, al hacerlo de una forma estandarizada y no una hecha a medida acorde a la información que se necesita intercambiar. La interfaz normalizada que REST define está pensada para lograr eficiencia al transferir datos hypermedia - como es el caso de la web - pero puede resultar subóptima para otro tipo de interacciones. La forma de lograr esta uniformidad es a partir de restricciones: un mecanismo unívoco para la identificación de recursos, la manipulación de estos recursos mediante representaciones, el uso de mensajes autodescriptivos y, principalmente, *Hypertext As The Engine Of Application State* (HATEOAS): el uso de hypermedia como el motor del estado de la aplicación.

2.5.1.6 Sistema en capas

La siguiente limitación que agrega la derivación en la que Fielding define REST es la de un sistema desarrollado en capas. Este estilo permite que la arquitectura se componga de capas jerárquicas que se conectan unas a otras de forma tal que un componente no pueda tener conocimiento más allá de la capa con la que está interactuando. Esta forma de concebir las interacciones permite que las capas abstraigan de la complejidad y la posible heterogeneidad de los sistemas que ocultan, y que su uso simplifique la escalabilidad de los sistemas al poder agregar capas que funcionen como balanceadores de carga.

2.5.1.7 Código bajo demanda

Por último, se agrega la restricción que identifica la forma de comunicación de la web: los clientes son entidades con una funcionalidad general, capaces de interpretar el código que el servidor le envía como respuesta a sus peticiones, a partir del cual obtienen el conocimiento específico. La forma en que se transmite este *know-how* al cliente es mediante *scripts*[15] que extienden esa funcionalidad básica, beneficiando a la extensibilidad del sistema general, pero reduciendo la visibilidad, por lo cual esta restricción es opcional.

2.5.2. Elementos de la arquitectura REST

2.5.2.1 Elementos de datos

REST basa las interacciones (la transmisión de la información) en el conocimiento compartido de los tipos de datos a través de la inclusión de metadatos. Los componentes se comunican transfiriendo los recursos utilizando formatos de representación estandarizados, elegidos acorde a lo solicitado por el receptor y la naturaleza del recurso. De esta forma, la información se compone de los siguientes elementos de datos:

- **Recursos e identificadores de recursos:** los recursos son la principal abstracción de REST, y hacen referencia a un conjunto de propiedades de una entidad en un momento dado del tiempo. Ésas propiedades pueden ser representaciones de recursos o identificadores de recursos, cuya semántica es lo que efectivamente diferencia un recurso de otro. REST utiliza los identificadores de recursos en la forma en que la web utiliza URLs y *Uniform Resource Name*s (URNs)[16] para referenciar recursos utilizando hypermedia.
- **Representaciones:** los componentes las utilizan para operar sobre el estado actual de un recurso y transmitirla entre sí. Una representación consiste de datos, metadatos para describirlos y, en algunos casos,

[15]En su especificación, Fielding además incluye los applets como medios, pero los obviaremos por tratarse de una tecnología en desuso en la actualdad, reemplazada y superada ampliamente por JavaScript.

[16]La definición formal de la sintáxis canónica de los URNs puede consultarse en la RFC 2141
`https://www.ietf.org/rfc/rfc2141`

metadatos para describir a los anteriores (también llamados datos de control), por ejemplo para validar la integridad de un mensaje. Los metadatos tienen la forma clave-valor, donde las claves obedecen al estándar que define la estructura y semántica del valor. La forma de describir la estructura de la información se llama *media type*.

2.5.2.2 Conectores

El estilo de arquitectura REST utiliza diferentes tipos de conectores para encapsular el acceso a los recursos y la transferencia de sus representaciones. Estos tipos se agrupan en:

- **Clientes:** junto a los servidores, son el tipo principal de conectores. Inician las comunicaciones al enviar una solicitud a un servidor.
- **Servidores:** esperan recibir ("escuchan") solicitudes ("conexiones") de los clientes, a las que responden a fin de dar acceso a sus servicios.
- **Caches:** pueden encontrarse en conexión a un cliente o un servidor. Su función es almacenar respuestas marcadas como cacheables para evitar realizar subsiguientes peticiones similares.
- **Resolvers:** traducen identificadores de recursos a la información efectiva de red necesaria para conectar los distintos componentes involucrados.
- **Túneles:** permiten realizar conexiones entre componentes que crucen alguno de los límites impuestos por la arquitectura en capas, generando comunicaciones directas entre componentes que de otra forma no podrían tener contacto.

Los conectores tienen una interfaz abstracta para establecer las comunicaciones entre los componentes, lo cual oculta los detalles del funcionamiento de conexión interno, separa las responsabilidades y permite que la implementación de cualquier parte del sistema sea remplazada sin afectar al resto de las entidades involucradas. En su funcionamiento, utilizan dos conjuntos de elementos en los mensajes, uno para el de la solicitud y otro para el de la respuesta. El primero consiste en datos de control de la solicitud (por ejemplo, para definir el formato en que se desea el recurso), un identificador de recurso (para indicar a qué recurso se desea acceder), y una representación opcional (por ejemplo, los datos con los que se quiere actualizar una entidad).

El conjunto de elementos utilizado en las respuestas se compone de datos de control para la respuesta (como información necesaria para alojar la respuesta en una cache), opcionalmente metadatos del recurso (como *links* relacionados al recurso), y una representación (por ejemplo, el recurso solicitado en sí), también opcional.

2.5.2.3 Componentes

Las partes que conforman la arquitectura de los sistemas REST se pueden clasificar acorde a su rol en las interacciones en:

- ***User agent*:** utiliza un conector cliente para enviar una solicitud, iniciando una comunicación, y luego espera una respuesta, la cual al llegar finalizará la interacción.
- ***Proxy*:** componente intermediario que funciona más cercano al lado del cliente, encapsulando la interfaz hacia otros servicios y, opcionalmente, brindando otras funcionalidades como traducción de información, mejoras de performance y medidas adicionales de seguridad. Funciona con conectores cliente y servidor para transmitir los mensajes (solicitudes y respuestas) que llegan a él.
- ***Gateway* o *Reverse proxy*:** otro componente intermediario, éste más cercano al servidor de origen, también encapsula servicios y puede brindar traducción de los datos que por él circulan, optimizaciones en la performance y realizar acciones para mitigar posibles problemas de seguridad. Al igual que un *proxy*, funciona como cliente y servidor a la vez, reenviando la información que por él pase.
- **Servidor de origen:** utiliza un conector de servidor para recibir solicitudes dirigidas a sus recursos. Las peticiones que le llegan obedecen a una estructura jerárquica que define cómo acceder a los servicios que provee.

3. Capítulo III: Análisis de tecnologías

En el proceso de elaboración de nuestra propuesta para el rediseño de la nube de servicios evaluamos diferentes tecnologías y herramientas para las distintas tareas en ella involucradas. En esta sección explicaremos, a partir de lo que hemos documentado de ese análisis, las principales alternativas que consideramos para cada caso.

Con el fin de agrupar lógicamente las herramientas a partir de su objetivo, las separamos en seis categorías e incluimos una conclusión por categoría en la que nos definimos por una de esas alternativas para utilizarla en nuestra propuesta final.

3.1. Para los servicios

Siendo la parte de la arquitectura que mayor desarrollo y mantenimiento de nuestra parte implicará, decidimos utilizar el lenguaje Ruby para implementar los servicios web. Basándonos en nuestra experiencia utilizando Ruby para el desarrollo de aplicaciones web, herramientas y *scripts*, seguimos eligiendo este lenguaje por los siguientes motivos:

- **Robustez sin sacrificio de la elegancia:** con varias aplicaciones en producción desarrolladas en este lenguaje[17], hemos comprobado que es robusto y potente, pero también elegante y *cómodo* de utilizar.
- **Agilidad:** implementar aplicaciones en Ruby, dada la gran oferta de librerías y frameworks que posee, es realmente ágil y práctico, principalmente si lo contrastamos con el desarrollo en lenguajes con procesos más largos de despliegue del código desde que se desarrolla hasta que se despliega en producción.
- **Sus bondades:** en estos más de 3 años realizando a diario desarrollos con Ruby, no hemos hecho más que apreciar cada vez más las bondades que tiene y los beneficios que nos trae, principalmente en comparación con experiencias anteriores utilizando otros lenguajes (principalmente PHP).

[17]La lista detallada de estas aplicaciones puede consultarse en el Anexo I

Complementariamente al lenguaje, hemos decidido utilizar un framework para desarrollo de aplicaciones web para sustentar nuestros servicios. El uso de un framework nos provee de una base probada, eficiente, estandarizada para nuestros proyectos, a la vez que nos asiste en cuestiones de seguridad fundamentales a la hora de exponer servicios en Internet. Es por eso que la elección del framework a usar no es un tema de menor importancia, ya que esperamos que éste nos asista y facilite las tareas básicas pertinentes al desarrollo de los servicios, permitiendo que el foco pueda mantenerse en la implementación de la lógica de los servicios y no verse desviada por la toma de decisiones triviales como, por citar un ejemplo, la organización del código del proyecto.

Partiendo de esa base, tomamos los dos frameworks para desarrollo de aplicaciones web más consolidados y utilizados[18] en el ámbito de Ruby como posibles opciones con las cuales implementar nuestros servicios web: Ruby on Rails y Sinatra.

En este apartado describiremos ambos frameworks comparativamente para concluir en la elección que hemos hecho para el presente trabajo.

3.1.1. Ruby on Rails

Ruby on Rails, o simplemente Rails, es *el* framework open source para desarrollo de aplicaciones web del lenguaje Ruby. Con más de 10 años de madurez y evolución, se ha convertido en el estándar *de facto* a la hora de considerar los cimientos sobre los cuales basar el desarrollo de una aplicación web, ya sea a pequeña, mediana o gran escala. Tan fuerte ha sido su influencia que frameworks de otros lenguajes lo han tomado como base en su diseño, siendo el caso más emblemático de estos el de la versión `1` del framework web para PHP Symfony.

3.1.1.1 Un framework, muchas librerías

Es un framework que sigue el patrón de diseño *Model-View-Controller* (MVC), que organiza en tres capas la lógica del dominio de una aplicación y ayuda a delimitar de manera clara las incumbencias y responsabilidades

[18]Para realizar esta apreciación nos basamos en las investigaciones que hemos hecho en el pasado como parte de nuestro trabajo en el CeSPI y en los datos que el sitio *The Ruby Toolbox* provee: `https://www.ruby-toolbox.com/categories/web_app_frameworks`.

de cada una de esas capas. Junto con esta organización siguiendo buenas prácticas probadas a lo largo de sus vastos años de existencia, Rails guía a los desarrolladores en el uso de técnicas beneficiosas como la realización de pruebas sobre el código de las aplicaciones (técnica conocida como *testing*), la organización clara y limpia del código, y el uso de librerías consolidadas. Esas librerías son lo que conforman internamente a Rails:

- Un conjunto de extensiones al lenguaje que simplifican y mejoran su usabilidad: `ActiveSupport`.
- Una capa de abstracción que permite manejar el modelo (la *M* de MVC, datos y lógica de negocios) con y sin bases de datos: `ActiveRecord` y `ActiveModel`.
- Los controladores (la *C* de MVC) que se encargan de aceptar y tratar los requerimientos entrantes a la aplicación, teniendo en cuenta aspectos importantes de seguridad, y de obtener la información necesaria para generar las respuestas: `ActionController` (parte de `ActionPack`).
- Un potente motor de generación de vistas (la *V* de MVC) que permite generar código HTML, JSON o XML entre otros, para ser enviado en las respuestas a los clientes: `ActionView` (parte de `ActionPack`).
- Una serie de componentes adicionales que permiten realizar tareas comunes en las aplicaciones web, como el envío de correos electrónicos, comunicaciones mediante *Web Sockets*, o el manejo de trabajos en segundo plano: `ActionMailer`, `ActionCable` y `ActiveJob`.

Rails integra estas librerías de manera tal que el desarrollo de la aplicación es ágil y se centra en la implementación de su lógica en sí, ya que el framework elimina decisiones triviales respecto a cómo organizar y gestionar elementos básicos comunes a cualquier aplicación web. Ese es otro gran punto a favor a la hora de ponderar Ruby on Rails y otras opciones.

3.1.1.2 El costo de Rails

Ruby on Rails es una herramienta altamente potente y estable, pero que posee una contra respecto a otras opciones: su uso puede traer aparejadas penalizaciones en el rendimiento de nuestra aplicación. Para casos muy sencillos, con dominios muy reducidos, el uso de Rails puede llegar a ser contraproducente debido al *overhead* que puede agregar la gran cantidad de componentes

que posee. Podemos tomar como ejemplo nuestra situación: necesitamos realizar aplicaciones web para proveer los servicios de nuestra nube, por lo cual no utilizaremos partes comunes a una aplicación web tradicional como pueden ser el manejo de sesiones de los clientes, la generación de vistas en otros formatos que no sean JSON, el envío de correos electrónicos[19], por nombrar algunos. Para situaciones como la nuestra, en general se suele optar por otros frameworks con funcionalidad más reducida, que eliminan ese *overhead* que agrega Rails, por lo que en nuestro caso sería contraproducente la utilización de éste. O al menos lo era, hasta la versión `4.2`.

3.1.1.3 `rails --api`

En su versión `5.0` (la versión más reciente al momento de escritura del presente trabajo), Ruby on Rails incorpora un nuevo *modo* de creación de aplicaciones web: el modo `--api`. Esto permite iniciar aplicaciones configuradas con un subconjunto reducido de componentes, obtenido a partir de la eliminación de aquellos que no son necesarios para el desarrollo de APIs web, y preparadas para servir contenido en formato JSON por defecto, aprovechando las técnicas de caching principales para este tipo de aplicaciones.

3.1.2. Sinatra

Sinatra, más que un framework, es un *Domain-Specific Language* (DSL) simple para el desarrollo rápido de aplicaciones web. Comparado con un framework completo como Ruby on Rails, Sinatra ofrece un conjunto reducido de funcionalidad, el estrictamente necesario para implementar la vista y el controlador del patrón MVC, pero no soporta oficialmente una forma concisa de organizar la lógica de negocios de las aplicaciones (la *M* de MVC).

Su mayor fortaleza radica en que permite rápidamente implementar aplicaciones web sencillas agregando un *overhead* mínimo. Podemos implementar una aplicación que reciba peticiones y responda con documentos HTML o JSON en poco tiempo y con muy pocas líneas de código, pero a medida que la complejidad crece es necesario complementar Sinatra con otras librerías de manera manual, lo cual no hace de esta herramienta un candidato fuerte para solucionar nuestras necesidades al implementar los servicios de la nube

---

[19] En general no será utilizado, aunque tenemos planes de implementar un servicio de notificaciones que sí lo haría.

de la Universidad Nacional de La Plata. Por ejemplo, si deseamos implementar modelos con conexión a una base de datos seguramente agreguemos `ActiveRecord`, la capa de abstracción de Ruby on Rails, y en el proceso deberemos manejar nosotros mismos la configuración y lógica de conexión a esa base de datos; similarmente ocurrirá si necesitamos utilizar una base de datos en memoria para el caching de nuestras aplicaciones (como podría ser Redis o Memcached).

3.1.2.1 Sencillez compleja

En el pasado hemos utilizado Sinatra para implementar algunas aplicaciones web sencillas, principalmente orientadas a proveer APIs que den soporte a clientes para mostrar datos a los usuarios. En esa experiencia comprobamos lo sencillo que es implementar aplicaciones pequeñas con esta herramienta, lo cual es altamente beneficioso para desarrolladores experimentados así como para aquellos que no lo son. Tiene un conjunto claro de directivas, una documentación extensa y detallada traducida al español, y promueve el desarrollo ágil y conciso.

También nos encontramos con los inconvenientes que trae aparejado Sinatra a la hora de salir de la *zona de comfort* de la herramienta: como mencionamos anteriormente, el trabajo adicional que requiere utilizar bases de datos, enviar correos electrónicos o implementar técnicas de caching, es un *overhead* no funcional que afecta en gran medida el desarrollo de las aplicaciones, quitando tiempo útil para destinarlo en tomar decisiones triviales que no afectan funcionalmente a las aplicaciones. Al ser tan sencillo, Sinatra tampoco promueve una forma estándar de organizar el código, lo cual es propenso a decisiones tendenciosas por parte de cada desarrollador a la hora de ubicar los componentes lógicos de la aplicación, resultando en muchos *pseudo-estándares* conviviendo en un mismo proyecto.

3.1.3. Conclusión

Al analizar estas dos opciones decidimos realizar una pequeña prueba de concepto implementando una parte de la API de datos de referencia en Ruby on Rails y en Sinatra. Esto no sólo nos permitió comparar el rendimiento de ambas versiones en términos de sus tiempos de respuesta, si no que además pudimos contrastar el esfuerzo necesario para desarrollar la misma solución en los dos contextos.

3.1.3.1 En términos de *performance*

Desde una visión estrictamente numérica ambas alternativas se comportaron de manera similar, obteniendo tiempos de respuesta promedio por debajo de los **150 milisegundos** en general. Al habilitar la cache local y las cabeceras de caching en el caso de la implementada con Ruby on Rails, las respuestas aprovechando esos niveles de cache bajaron al orden de los **10 ó 20 milisegundos**, lo cual fue un gran salto de velocidad de respuesta.

Si bien nos fue de gran utilidad implementar estas pruebas de concepto, desde el punto de vista de la *performance* estrictamente no existe una diferencia determinante que indique claramente qué herramienta utilizar. Es por esto que debemos considerar otro enfoque para la comparación: el de la productividad del desarrollador.

3.1.3.2 Desde el punto de vista del desarrollador

Además de su rendimiento, otro factor importante a la hora de elegir un framework para el desarrollo es tener una noción de en qué medida agiliza y facilita la implementación de la aplicación propiamente dicha. Algunos casos en que la herramienta podría impactar negativamente en el desarrollo son:

- Si el desarrollador se ve obligado a tomar decisiones no funcionales muy seguido,
- si el framework necesita ser complementado con un número alto de librerías de terceros para cubrir los requerimientos,
- si extender la funcionalidad del framework implica mucha investigación y trabajo manual, o
- si simplemente el conjunto de funcionalidad que éste ofrece no es suficiente.

Cualquiera de estas situaciones que enumeramos a modo de ejemplo funciona como indicador de que la herramienta elegida no está a la altura de las circunstancias, y eso fue exactamente lo que nos ocurrió al implementar la prueba de concepto con Sinatra. Nos encontramos con la necesidad de implementar y gestionar manualmente la conexión a la base de datos, con la falta de organización clara del código del proyecto, y con una complejidad innecesaria para implementar técnicas de caching.

Por el contrario, al implementar la prueba de concepto con Ruby on Rails no existieron estos retrasos en el desarrollo. El posible inconveniente que puede existir al adoptar Rails como framework para el desarrollo es la curva de aprendizaje del mismo, pero en nuestro caso no es un factor a considerar debido a que ya poseemos experiencia con él tanto nosotros como el resto de nuestro equipo de trabajo. El desarrollo de esa prueba de concepto fue mucho más ágil que en el caso de Sinatra y prácticamente no tuvimos que tomar decisiones triviales o considerar cómo organizar el código del proyecto, ya que el mismo framework nos facilitaba esas decisiones.

Por esas razones es que hemos optado por usar el framework Ruby on Rails como base para el desarrollo de los servicios de la nueva versión de la nube de la UNLP.

3.2. Para la estructura de los servicios

La consistencia, coherencia y claridad en la estructura de las respuestas de los servicios que una nube provee es crítica para su usabilidad y mantenibilidad. En el diseño del Integrador hemos optado por definir nuestra propia estructura basada simplemente en el esquema de nuestras bases de datos: publicamos todos los campos indiscriminadamente, lo cual resulta en algunos casos en la exposición innecesaria y excesiva de valores en las respuestas de la API. Esto, sumado a una cierta inconsistencia en los nombres de los campos y en la forma en que se conforman las URLs y sus parámetros, hacen difícil el uso de la nube sin consultas constantes a una documentación incompleta y desactualizada; consultas que en la myoría de los casos acaban por convertirse en una pérdida de tiempo y en la necesidad de leer el código fuente de las APIs para conocer exactamente qué parámetros admite y qué efecto tienen en la petición.

En nuestro análisis de la nube actual éste fue uno de los puntos más importantes a mejorar: el requerimiento de tener una estructura estandarizada, fundada en casos de éxito, y que se integre con alguna herramienta de automatización de la documentación (veremos más sobre eso en la Subsección 3.6) para que podamos tenerla actualizada sin necesidad de hacerlo manualmente.

Con respecto al formato de la respuestas, hemos decidido mantener JSON como nuestra opción. Se trata de un formato muy amistoso para el desarrollador por su claridad y su relativamente poco *overhead* para la codificación y decodificación, además de que se encuentra en uso en un creciente número de

APIs a nivel global[20], por lo que nos centramos en la búsqueda de estándares específicos al formato JSON.

En este apartado analizaremos diferentes estándares, relacionados a la estructura de las respuestas y parámetros en las consultas, para el diseño y la implementación de una API.

3.2.1. HAL

Este formato pensado para las respuestas de APIs hypermedia, es en sí un *media type* basado en los formatos JSON o XML que define cómo estructurar las respuestas de una API para que respete los principios del modelo REST que hemos tratado con anterioridad, principalmente el de hypermedia. HAL apunta a hacer las APIs explorables y su documentación accesible desde éstas mismas.

3.2.1.1 *Media type* dedicado

Para denotar los documentos JSON o XML que utilizan el formato HAL, se deben utilizar los *media types* dedicados a tal fin: `application/hal+json` y `application/hal+xml`, respectivamente. Debido a la orientación de nuestro trabajo, de aquí en más nos referiremos únicamente a la variante JSON de HAL.

3.2.1.2 Diseño sencillo y enfocado

El principal beneficio de este formato radica en su sencillez, ya que se limita únicamente a definir la estructura que las interrelaciones de las respuestas de nuestros servicios pudieran tener. El concepto que motiva su diseño es intentar que HAL funcione como HTML para máquinas en el sentido que sea el medio para establecer hipervínculos, como ocurre con las páginas web que se vinculan entre sí, que conecten de una manera genérica diferentes interacciones posibles en una o varias APIs.

[20] El interés en las APIs JSON ha crecido constantemente desde el año 2004 al presente según las tendencias de Google. Cf. `http://www.google.com/trends/explore?q=xml%20api%2C%20json%20api%2C%20html%20api`.

3.2.1.3 El modelo

HAL posee dos conceptos principales: el *recurso* y los *links* (vínculos). Los recursos HAL poseen links a URIs, otros recursos *embebidos* en ellos y un estado representado por sus atributos. Por otra parte, los links poseen un destino (una URI), un nombre para la relación que representa (o `rel`) y opcionalmente otras propiedades que funcionan de metadatos.

Esta estructura permite que los clientes de las APIs que utilicen HAL naveguen por éstas al seguir los links que los recursos poseen.

En el bloque de código 2 podemos distinguir los elementos antes mencionados, de la siguiente forma:

- El documento HAL en su totalidad es el recurso.
- El atributo de primer nivel `_links` es el contenedor de todos los links del recurso.
- Los atributos de primer nivel `name`, `surname`, `document_number` y `document_type_id` son los atributos del recurso principal.
- El atributo de primer nivel `_embedded` contiene los recursos embebidos al principal. En este caso, sólo uno: `document_type`.

3.2.1.4 Madurez y actualidad

HAL surgió a mediados de 2011 y fue actualizado en la segunda mitad de 2013, fecha desde la cual no ha tenido revisiones ni agregados para completarlo.

3.2.2. JSON API

JSON API es una especificación surgida en 2013 que intenta definir tanto cómo deben los clientes formular las peticiones, así como la forma en que los servidores deben responder a ellas, fomentando la eficiencia en el uso de recursos. Al momento de escritura del presente trabajo, esta especificación se encuentra en su versión `1.0` y con una versión `1.1` en proceso de definiciones.

En este análisis resumiremos de manera concisa los puntos sobresalientes de la especificación.

```
{
  "_links": {
    "self": {
      "href": "/people/00027855859"
    },
    "document_type": {
      "href": "/document_types/0"
    }
  },
  "name": "Miguel",
  "surname": "Carbone",
  "document_number": "27855859",
  "document_type_id": "0",
  "_embedded": {
    "document_type": {
      "_links": {
        "self": {
          "href": "/document_types/0"
        }
      },
      "abbreviation": "DNI",
      "code": 0,
      "name": "Documento Nacional de Identidad"
    }
  }
}
```

Bloque de código 2: Recurso HAL de ejemplo

3.2.2.1 *Media type* dedicado

Tanto los datos que el cliente envía en su petición como la respuesta del servidor deben indicar el *media type* dedicado de JSON API para cumplir con la especificación. El *media type* apropiado es: `application/vnd.api+json`[21].

Este tipo de contenidos es una definición de estructura basada en el formato JSON. Denominaremos "documentos JSON API" a aquellos documentos JSON que adhieran a este *media type*.

[21] Este tipo de contenido fue asignado por la IANA - http://www.iana.org/assignments/media-types/application/vnd.api+json

3.2.2.2 Estructura general de los documentos JSON API

Los documentos deben contener como elemento raiz un objeto, en el cual son admisibles las siguientes propiedades de primer nivel:

- `data`: la información principal del documento, puede ser un recurso o una colección de éstos.
- `errors`: indicadores de cualquier error que hubiera ocurrido.
- `meta`: especifica metadatos sobre la información.
- `jsonapi`: descripción de la implementación del servidor JSON API.
- `links`: conjunto de vínculos hypermedia relacionados a la información principal.
- `included`: recursos incluidos en el documento por estar relacionados al objeto principal.

3.2.2.3 Los recursos

Los elementos principales de información de un documento JSON API son los recursos. Cada recurso debe al menos tener dos propiedades: `id` y `type`, excepto cuando se trate de un *nuevo* elemento que esté siendo creado por el cliente del servicio en cuyo caso el `id` puede omitirse. Además de estas propiedades obligatorias un recurso puede tener las propiedades `attributes`, `relationships`, `links` y `meta`, donde las dos primeras son objetos JSON que indican los atributos del recurso y posibles elementos relacionados respectivamente, y las últimas dos tienen sentidos similares a las propiedades homónimas del elemento raiz mencionadas en el apartado anterior que describe la estructura general de un documento JSON API. En el mismo sentido, los recursos pueden identificarse mediante una versión reducida de sí mismos que sólo contenga las propiedades `id` y `type`, caso en el cual se los denomina "objetos identificadores de recursos".

En este punto, la especificación hace una recomendación respecto de los campos (dentro de los `attributes` de un recurso) que representan *asociaciones lógicas* o claves foráneas a otros recursos: los valores de clave foránea apuntando hacia otros recursos no deberían aparecer entre los atributos del recurso; en cambio debieran tener su entrada dedicada entre las *relationships* del mismo.

Las relaciones de un recurso son objetos JSONs que pueden tener las siguientes propiedades:

- `links`: un objeto JSONs que puede tener dos propiedades, `self` (URL mediante la cual se puede manipular la relación en sí) y `related` (URL del recurso asociado en sí, que no debe ser dependiente de la existencia de la relación).
- `data`: uno o varios (acorde a la cardinalidad de la relación) objetos identificadores de recursos para los recursos asociados.
- `meta`: un objeto JSONs con metadatos sobre el recurso.

La especificación, con miras a hacer un uso más eficiente de los recursos, define también los "documentos compuestos" que son aquellos que incluyen en la misma estructura la información relacionada a los recursos asociados a la información principal del documento, es decir, aquellos recursos referenciados en alguna de las relaciones. La información de estos recursos adicionales debe realizarse dentro de una clave `included` de primer nivel en la raiz del documento JSON API. Al incorporar esta información adicional al documento, se reducen la cantidad de peticiones necesarias para obtener un recurso y sus elementos relacionados, reduciendo la utilización de recursos necesaria para completar la tarea.

En el bloque de código 3 incluimos un ejemplo completo de un documento JSON API.

```
{
  "data": {
    "type": "document_types",
    "id": "0",
    "attributes": {
      "abbreviation": "DNI",
      "code": 0,
      "name": "Documento Nacional de Identidad"
    },
    "relationships": {
      "people": {
        "links": {
          "self": "/document_types/0/relationships/people",
          "related": "/document_types/0/people"
        },
```

```
        "data": [
          { "type": "people", "id": "00027855859" },
          { "type": "people", "id": "00031988189" }
        ]
      }
    }
  },
  "links": { "self": "/document_types/0" },
  "included": [
    {
      "type": "people",
      "id": "00027855859",
      "attributes": {
        "name": "Miguel",
        "surname": "Carbone",
        "document_number": "27855859"
      },
      "relationships": {
        "document_type": {
          "data": { "type": "document_types", "id": "0" }
        }
      },
      "links": { "self": "/people/00027855859" }
    },
    {
      "type": "people",
      "id": "00031988189",
      "attributes": {
        "name": "Jose Nahuel",
        "surname": "Cuesta Luengo",
        "document_number": "31988189"
      },
      "relationships": {
        "document_type": {
          "data": { "type": "document_types", "id": "0" }
        }
      },
      "links": { "self": "/people/00031988189" }
    }
  ]
}
```

Bloque de código 3: Documento JSON API representando un recurso

3.2.2.4 Obtención de recursos

Las peticiones de acceso de lectura a la información deben ser realizadas utilizando el método HTTP `GET` y utilizando alguna de las URLs provistas en los documentos JSON API como vínculos `self` o `related`. Por ejemplo, las peticiones presentadas a continuación son válidas para obtener información a partir del documento JSON API presentado anteriormente.

La petición del bloque de código 4 se puede utilizar para obtener la colección de tipos de documentos:

```
GET /document_types HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 4: Petición de una colección de recursos JSON API

De manera similar, la petición del bloque de código 5 puede usarse para obtener todas las personas relacionadas con el tipo de documento con `id` `"0"`:

```
GET /document_types/0/people HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 5: Petición de un recurso relacionado en JSON API

Y la del bloque de código 6 para obtener una persona en particular:

```
GET /people/00027855859 HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 6: Petición de un recurso en JSON API

Las respuestas exitosas a estas peticiones deben devolver un código de estado HTTP `200 OK` junto con el recurso (o un elemento nulo, en caso de no existir el recurso identificado y estar siendo solicitado a través de una relación) o la colección de recursos (o una colección vacía, en caso de no poseer elementos).

En el caso de las peticiones para obtener un único recurso inexistente que no sean realizadas a través de la URL de una relación, el servidor deberá responder con un código de estado HTTP `404 Not Found`.

El documento JSON API expuesto en el bloque de código 3 es un ejemplo del cuerpo de una respuesta exitosa que debiera responder con el encabezado HTTP del bloque de código 7:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json
```

Bloque de código 7: Encabezado HTTP de respuesta exitosa JSON API

Similarmente, el extracto presentado en el bloque de código 8 representa una respuesta exitosa a una petición sobre la colección de tipos de documento.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/document_types"
  },
  "data": [{
    "type": "document_types",
    "id": "0",
    "attributes": {
      "abbreviation": "DNI",
      "code": 0,
      "name": "Documento Nacional de Identidad"
    }
  },
  {
    "type": "document_types",
    "id": "1",
    "attributes": {
      "abbreviation": "DNT",
      "code": 1,
      "name": "DNI Temporario"
    }
  }]
}
```

Bloque de código 8: Respuesta exitosa a petición de una colección de recursos en JSON API

Si en cambio la petición fuese exitosa pero su respuesta no contuviese elementos, las respuestas serían las presentadas en los bloques de código 9 (para un recurso singular) y 10 para una colección.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/people/00027855859/relationships/document_type"
  },
  "data": null
}
```

Bloque de código 9: Respuesta JSON API para una petición exitosa a un recurso no encontrado

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/document_types"
  },
  "data": []
}
```

Bloque de código 10: Respuesta JSON API para una petición exitosa a una colección de recursos vacía

En los ejemplos anteriores se demuestra el uso del valor nulo (`null`) y la colección vacía (`[]`) para denotar la respuesta a una petición exitosa pero que no arroja coincidencias.

3.2.2.5 Obtención de relaciones

Como se mencionó anteriormente, las relaciones pueden tener un vínculo a sí mismas (`self`) que permita manipular la relación en sí y no a los recursos que ésta vincula. Esa URL consultada con el método HTTP `GET` nos permite consultar la información asociada a esa relación, obteniendo como dato principal el recurso o los recursos involucrados en ella en forma de identificadores de recursos (objetos JSON con las propiedades `id` y `type` únicamente).

En el bloque de código 11 se puede apreciar el documento JSON API que representa una relación.

El servidor responderá con un código de estado HTTP `200 OK` en caso de tratarse de una consulta exitosa, y con un `404 Not Found` si se tratase de una petición a una relación sobre un recurso principal inexistente. A modo de ejemplo, se presentan estos dos posibles casos en situaciones concretas:

- `GET /people/00027855859/relationships/document_type`
  Esta petición, realizada sobre el recurso principal de tipo `people` con identificador `00027855859`, pide el recurso de tipo `document_type` relacionado. Considerando que el recurso principal existe, se retornará invariablemente un código de estado HTTP `200 OK` y una respuesta cuyo atributo `data` contendrá o bien un recurso existente, o bien un valor nulo (`null`) en caso que dicho recurso no exista.

- `GET /people/inexistente/relationships/document_type`
  Esta petición, similar a la anterior pero realizada sobre un ficticio recurso principal inexistente, tendrá una respuesta con estado HTTP `404 Not Found` indicando que es el recurso principal el que no existe, y por ende descartando la posibilidad de que la relación solicitada pueda ser obtenida.

3.2.2.6 Inclusión de recursos relacionados

El estándar define una forma para reducir la cantidad de peticiones realizadas a los servicios mediante la inclusión de recursos secundarios o *relacionados* al recurso o conjunto de recursos principal. La forma de indicar que se desea utilizar esta funcionalidad es mediante la especificación del parámetro `include` con un valor que contenga una o más relaciones del recurso (o recursos asociados) separadas por coma. Por ejemplo, el valor presentado en el bloque 12 indicaría que se incluyan recursos relacionados.

3.2.2.7 Selección de campos

Otra forma incluida en la especificación orientada a la reducción del uso de recursos son los *sparse fieldsets*, recursos que son consultados indicando específicamente qué campos se deben incluir en la respuesta. Así como el apartado anterior explica una forma de reducir requerimientos, éste explica una forma de reducir el tamaño de cada respuesta, e inclusive ambos pueden

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/document_types/1/relationships/people",
    "related": "/document_types/1/people"
  },
  "data": [{
    "type": "people",
    "id": "00027855859"
  },
  {
    "type": "people",
    "id": "00031988189"
  }]
}
```

Bloque de código 11: Respuesta JSON API para una petición exitosa a una relación

```
GET /people/00027855859?include=document_type HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 12: Petición de un recurso y relaciones asociadas en JSON API

combinarse para reducir la cantidad de campos de diferentes recursos en una misma respuesta JSON API. Es justamente por este tipo de situaciones en que una misma respuesta podría incluir diferentes recursos de diferentes tipos que la forma de especificar qué campos se desean obtener en la respuesta es mediante el parámetro `fields` el cual es un arreglo de claves y valores, donde las claves son tipos de recursos y los valores los campos a incluir en la respuesta. A continuación, en el bloque de código 13, se presenta un ejemplo de petición utilizando esta posibilidad definida por JSON API.

3.2.2.8 Especificación de orden

Al realizar una petición de una colección, donde normalmente se retornará más de un elemento, la especificación reserva el parámetro `sort` para indicar diferentes criterios de ordenamiento de los recursos en la respuesta. El

```
GET /people/00027855859?include=document_type&
fields[people]=name,document_number&
fields[document_types]=abbreviation HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 13: Petición de un recurso utilizando *sparse fieldsets* e `include` en JSON API

formato para el valor del parámetro `sort` es una lista de campos, separados por coma, en la que se puede anteponer un símbolo menos (`-`) a cada campo en que se quiera indicar un orden descendente. En el bloque de código 14 se puede observar un ejemplo de esto.

```
GET /people?sort=surname,name HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 14: Petición indicando el orden de una colección de recursos en JSON API

3.2.2.9 Paginación de colecciones

JSON API define también una forma estandarizada de limitación de la cantidad de recursos incluidos en cada respuesta a las peticiones realizadas a las colecciones, la cual llama *paginación*. Esta técnica divide en *páginas* de una cierta cantidad de elementos (*tamaño de página*), accesibles mediante un índice o *número de página*, siendo `1` el número de la primera página.

A fin de que los clientes puedan especificar qué tamaño de página desean utilizar y qué número de página desean obtener, JSON API reserva el parámetro `page` y define que será un arreglo de clave-valor con dos claves posibles: `size` y `number`, para indicar el tamaño y el número de página, respectivamente. En el bloque de código 15 se muestra un ejemplo de una petición que indica estos parámetros. Si el cliente no especificase nada respecto de la paginación, el servidor puede implementar un comportamiento por defecto a su criterio.

```
GET /people?page[number]=2&page[size]=30 HTTP/1.1
Accept: application/vnd.api+json
```

Bloque de código 15: Petición indicando parámetros de paginación en JSON API

3.2.2.10 Filtrado de resultados

Adicionalmente a las posibilidades antes mencionadas, JSON API permite que se filtren los resultados de una petición mediante la palabra clave `filter`, cuyos valores serán los criterios de búsqueda que el cliente envíe al servidor para reducir el número de recursos únicamente a los que apliquen a dichos criterios. La forma concreta de interpretación de este parámetro se deja abierta a la implementación del servidor.

3.2.3. Conclusión

En nuestra búsqueda de formatos estándares para la estructura de la respuestas de nuestros servicios encontramos una gran variedad de opciones informales, propuestas desarrolladas por distintas empresas, pero sólo hallamos dos opciones que poseen la orientación adecuada para nuestras necesidades: HAL y JSON API.

El primer formato, HAL, es conciso y sencillo de implementar ya que no presenta grandes complicaciones en cuanto a sus requisitos funcionales. Si bien esto puede ser positivo, en esta situación nos resulta insuficiente para definir por completo el protocolo de acceso a nuestros servicios. En el mismo sentido, HAL no se encuentra en una versión definitiva, completa ni estable, lo cual lo elimina como candidato si consideramos que desde el año 2013 no recibe actualizaciones.

En el caso de JSON API, lo encontramos altamente beneficioso desde el punto de vista del desarrollo por su clara documentación, por tratarse de un estándar que define un protocolo de acceso a la información, por estar basado en las opiniones de varias empresas y equipos encabezados por arquitectos de renombre en la industria[22], por poseer una creciente adopción y por ser fácil de integrar con el framework que hemos elegido para implementar los servicios (Ruby on Rails), y es por estos motivos que lo utilizaremos para estructurar las respuestas de nuestra nube de servicios.

[22] Los principales editores de la especificación son Steve Klabnik (Ruby on Rails, lenguaje Rust), Yehuda Katz (Ruby on Rails, Ember.js, W3C TAG), entre otros.
`http://jsonapi.org/about`

3.3. Para el nodo central

3.3.1. Mulesoft ESB

Mule ESB es un ESB liviano basado en Java, una plataforma de integración que permite a los desarrolladores conectar aplicaciones rápido y fácilmente. Posibilita una integración sencilla de los sistemas existentes, independientemente de las tecnologías utilizadas para las aplicaciones, incluidas JMS, Web Services, JDBC y HTTP, entre otras.

La principal ventaja de un ESB es que permite que diferentes aplicaciones se comuniquen entre sí, actuando como un sistema de transporte para transmitir datos entre las aplicaciones dentro de la organización o a través de Internet. Mule ESB posee potentes capacidades, entre ellas:

- Creación y alojamiento de servicios.
- Mediación de servicios.
- Ruteo de mensajes.
- Transformación de datos.

3.3.1.1 Licencia

Licencia CPAL para *Community Edition*, propiedad de Enterprise Edition.

3.3.1.2 Aplicabilidad del proyecto

Mule ESB es un proyecto que no permite su libre distribución debido a su esquema de licenciamiento. Inicialmente incluimos este producto en nuestro análisis por poseer cierto renombre en la materia, pero al ahondar en sus detalles y conocer su modelo de negocios hemos decidido descartarlo automáticamente, al no ajustarse a nuestro propósito de utilizar tecnologías *Open Source*.

3.3.2. Kong

Es una herramienta desarrollada por la empresa Mashape que facilita administración de APIs, centralizando el acceso a las mismas desde uno o

más servidores que se encargan de llevar registro de qué servicios ofrecen, recibir los requerimientos para éstos y delegar la generación de las respuestas a los *backends* destinados a tal fin.

Kong funciona con una versión modificada del servidor web NGINX (Cf. Subsubsección 3.5.2), haciendo las veces de *proxy* reverso y proveyendo una estructura extensible mediante el uso de agregados (*plugins*) que permite brindar funcionalidad adicional a la instalación base, como ser autenticación, registro en logs, limitación de datos (*rate limiting*), *caching*, por nombrar algunos. Toda su información se almacena en una base de datos no relacional Apache Cassandra, altamente escalable por naturaleza.

Básicamente, Kong se ubica entre los clientes y las instancias vivas de las APIs, recibiendo los requerimientos para luego de aplicar las reglas que se le hayan especificado, enviarlos al servicio que corresponda.

3.3.2.1 Licencia

Kong se encuentra publicado bajo licencia Apache 2.0[23].

3.3.2.2 Extensiones disponibles (*plugins*)

Este producto permite extender su funcionalidad y comportamiento básico mediante la adición de *plugins* que brindan variadas funciones. Adicionalmente, en caso que necesitemos agregar comportamiento que no se encuentra disponible, podemos implementarlo nosotros mismos y contribuir con nuestra propia extensión a la comunidad.

A continuación presentamos un breve listado de las extensiones que Kong ofrece actualmente:

- **Autenticación:** Kong provee *plugins* para agregar autenticación a las APIs mediante distintas estrategias:
  - **Basic:** Autenticación mediante usuario y contraseña.
    `https://getkong.org/plugins/basic-authentication`
  - **Key:** Autenticación mediante una clave de API.
    `https://getkong.org/plugins/key-authentication`

[23]La misma puede ser consultada accediendo a `https://getkong.org/license/`

  - **OAuth 2.0:** Autenticación mediante el protocolo OAuth 2.0.
    `https://getkong.org/plugins/oauth2-authentication`
  - **HMAC:** Permite autenticar la identidad del cliente (*consumer*) mediante firmas HMAC en los mensajes.
    `https://getkong.org/plugins/hmac-authentication`
  - **JWT:** Provee autenticación mediante el uso del estándar JSON Web Tokens.
    `https://getkong.org/plugins/jwt`

- **Seguridad:** Las siguientes extensiones de Kong proveen capas adicionales de seguridad a los servicios:

  - ***Access Control List* (ACL):** Agrega listas de control de acceso para limiter qué clientes (*consumers*) pueden acceder a qué APIs.
    `https://getkong.org/plugins/acl`
  - ***Cross-Origin Resource Sharing* (CORS):** Permite que se realicen requerimientos desde distintos dominios con las limitaciones que impongamos, algo ideal para la implementación de clientes puramente desarrollados en JavaScript que corran por completo en el navegador de los usuarios.
    `https://getkong.org/plugins/cors`
  - ***Secure Sockets Layer* (SSL):** Agrega la posibilidad de establecer conexiones verificables mediante la inclusión de certificados SSL a los servicios que proveemos, tanto individual como globalmente, sin necesidad de hacerlo en cada uno de los puntos de acceso de las APIs.
    `https://getkong.org/plugins/ssl`
  - **Restricciones por dirección IP:** Posibilita manejar listas blancas y listas negras de direcciones IP que pueden realizar requerimientos a nuestros servicios.
    `https://getkong.org/plugins/ip-restriction`

- **Control de tráfico:** Estas extensiones habilitan reglas de control sobre el tráfico entrante y saliente de nuestras APIs:

  - **Límite de tasa de consulta (*Rate limiting*):** Permite limitar la cantidad de requerimientos que un cliente puede hacer a los servicios en un período de tiempo dado.
    `https://getkong.org/plugins/rate-limiting`

- **Límite de tasa de respuesta (*Response rate limiting*):** Hace posible limitar la cantidad de respuestas que los servicios envían en un período de tiempo dado (aplica límites sobre el tráfico saliente). `https://getkong.org/plugins/response-rate-limiting`
- **Límite de tamaño de solicitud (*Request size limiting*):** Bloquea aquellos requerimientos cuyo cuerpo exceda un límite que especifiquemos en su tamaño. `https://getkong.org/plugins/request-size-limiting`

- **Analíticas y monitoreo:** Tal vez el punto con menos opciones que provee Kong, ya que posee un único *plugin* que agregue funcionalidad en este aspecto:

  - **Galileo:** Integra el servicio de analíticas y monitoreo de APIs Galileo, una plataforma paga de *Business Inteligence* desarrollada por Mashape, los creadores de Kong. `https://getkong.org/plugins/galileo`

- **Transformaciones:** Mediante estas extensiones, podemos realizar modificaciones a las peticiones o las respuestas de un servicio cuando pasa por Kong:

  - **Transformación de peticiones (*Request transformer*):** Modifica la solicitud antes de enviarla al servicio correspondiente. `https://getkong.org/plugins/request-transformer`
  - **Transformación de respuestas (*Response tranformer*):** Altera la respuesta obtenida desde la API antes de enviarla al cliente del servicio. `https://getkong.org/plugins/response-transformer`

- **Registro de eventos (*Logging*):** Estos *plugins* permiten escribir a logs de registro las solicitudes y respuestas que pasan por Kong mediante distintas estrategias:

  - **TCP:** Envía la información al log mediante una conexión realizada con un servidor que habla el protocolo TCP. `https://getkong.org/plugins/tcp-log`
  - **UDP:** Ídem anterior, excepto que usando protocolo UDP. `https://getkong.org/plugins/udp-log`
  - **HTTP:** *Plugin* similar a los anteriores, excepto que lo hace contra un servidor que habla el protocolo HTTP. `https://getkong.org/plugins/http-log`

- **Archivo (*File*):** Escribe la información en archivo local al servidor de Kong.
`https://getkong.org/plugins/file-log`

3.3.2.3 Instalación y prueba

Basándonos en los pasos detallados en la guía oficial de Kong[24] para su instalación y ejecución mediante Docker[25], realizamos pruebas de concepto para analizar la factibilidad de implementación de esta herramienta, las cuales resultaron altamente satisfactorias. A continuación, reproduciremos las pruebas realizadas en el ambiente local[26], indicando los comandos ejecutados y las salidas obtenidas (en general acotadas a los datos relevantes para el caso). Para sintetizar los pasos, omitiremos la preparación del ambiente Docker, aunque se podría resumir simplemente en instalar dicha herramienta en el equipo donde se ejecutarán los contenedores.

Para iniciar los dos servicios requeridos por Kong para su funcionamiento (Apache Cassandra y Kong mismo), ejecutamos la siguiente secuencia de comandos:

---

[24] `https://github.com/Mashape/docker-kong`

[25] Desarrollar el potencial de Docker y las posibilidades que abre desde el punto de vista tanto de desarrollo de aplicaciones como de administración de las mismas en producción merecería un capítulo entero. Debido a que esto excede el alcance del presente trabajo, nos limitaremos a definir a Docker como una herramienta que permite ejecutar servicios (de cualquier tipo) en ambientes livianos aislados basados en GNU/Linux, los cuales ya empaquetan el servicio y cualquier dependencia que éste pudiera tener. Docker corre sobre un sistema operativo base que soporte la tecnología de contenedores (*containers*). Para mayor información, se puede consultar el sitio oficial de Docker: `https://www.docker.com/what-docker`

[26] De ahí que todas las referencias a servicios que en estos pasos se realizan son utilizando `http://localhost`

```
# Bajar imágenes de los contenedores (sólo la primera vez)
$ docker pull mashape/cassandra
$ docker pull mashape/kong
# Iniciar contenedores (Cassandra y Kong)
$ docker run -p 9042:9042 -d \
             --name cassandra mashape/cassandra
$ docker run -p 8000:8000 -p 8001:8001 -d \
             --name kong \
             --link cassandra:cassandra mashape/kong
# Probar con un requerimiento si está funcionando
$ curl http://127.0.0.1:8001
{
  "version":"0.5.3",
  "lua_version":"LuaJIT 2.1.0-alpha",
  "tagline":"Welcome to Kong",
  "hostname":"0b53705aaef9",
  ...
}
```

Bloque de código 16: Preparación y arranque de Kong

Con el último comando comprobamos que el servicio está funcionando y que responde con la información de la instancia en ejecución, en formato JSON. Ahora que Kong está correctamente instalado y ejecutándose, podemos pasar a configurarlo y probar su funcionamiento.

Por defecto Kong atiende requerimientos en dos puertos diferentes:

- **Puerto 8000:** Aquí escucha la porción pública de Kong que funciona como proxy reverso de los servicios que registremos en él, proveyendo acceso a sus APIs. A este puerto irán dirigidos todos los requerimientos de los clientes.

- **Puerto 8001:** En este puerto Kong provee una API administrativa, mediante la cual se realizan todas las tareas de configuración y personalización de la herramienta. Esto implica varias cosas: por un lado, no necesitamos tener acceso por consola a los servidores donde corre el servicio de Kong, ya que simplemente con peticiones HTTP basta para configurarlo; y por otro lado que necesitamos agregar políticas de seguridad adicionales para limitar el acceso a este puerto.

Teniendo esto en cuenta, procederemos a configurar una API de pruebas

en Kong, utilizando el servicio online Mockbin[27] que ofrece endpoints que imitan una API real para este tipo de situaciones.

```
# Damos de alta el servicio Mockbin en nuestra instancia de Kong
$ curl -i -X POST http://localhost:8001/apis/ \
       --data 'name=mockbin' \
       --data 'upstream_url=http://mockbin.com/' \
       --data 'request_host=mockbin.com'
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
...
Server: kong/0.5.3

{
  "upstream_url":"http:\/\/mockbin.com\/",
  "id":"c1550dc6-3482-4c8f-ccba-b978f81174ea",
  "name":"mockbin",
  ...
  "request_host":"mockbin.com"
}

# Probamos realizar un requerimiento a Mockbin a través de Kong
$ curl -i http://localhost:8000/ --header 'Host: mockbin.com'
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...
Via: kong/0.5.3

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mockbin by Mashape</title>
  ...
</html>
```

Bloque de código 17: Comandos para agregar un API a Kong

Una vez agregada una API, en nuestro caso lo más probable será que

[27] http://mockbin.com

querramos limitar su acceso mediante políticas de seguridad. A tal fin, agregaremos autenticación mediante clave con el *plugin* **Key** de Kong.

Inicialmente, es necesario habilitar el *plugin* antes mencionado sobre la API que hemos creado (`mockbin`) y probar que no se permitan peticiones sin datos de autenticación.

```
# Habilitar el plugin key-auth para Mockbin
$ curl -i -X POST http://localhost:8001/apis/mockbin/plugins/
 ↪  --data 'name=key-auth'
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
...
Server: kong/0.5.3
{
  "api_id":"c1550dc6-3482-4c8f-ccba-b978f81174ea",
  "id":"b68a61e8-0aa4-422b-c208-cf1a29032881",
  "enabled":true,
  "name":"key-auth",
  ...
}

# Probar que la autenticación funcione, un requerimiento
# sin clave no debería ser permitido:
$ curl -i http://localhost:8000/ --header 'Host: mockbin.com'
HTTP/1.1 401 Unauthorized
Content-Type: application/json; charset=utf-8
...
WWW-Authenticate: Key realm="kong"
Server: kong/0.5.3
{"message":"No API Key found in headers, body or querystring"}
```

Bloque de código 18: Comandos para habilitar autenticación mediante clave sobre una API

Una vez comprobado que es necesario incluir información de autenticación para poder acceder al servicio, procedemos a crear un *consumer* (cliente) para asignar a esta API y le asignamos una clave para que pueda identificarse ante Kong.

```
# Agregamos un consumer y le asignamos una clave propia
 ↪  ("t3st-K3Y")
$ curl -i -X POST http://localhost:8001/consumers/ --data
 ↪  "username=test_user"
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
...
Server: kong/0.5.3
{
  "username":"test_user",
  ...
  "id":"128a20ce-de29-4d11-cd51-ab005a137667"
}

$ curl -i -X POST
 ↪  http://localhost:8001/consumers/test_user/key-auth/ --data
 ↪  'key=t3st-K3Y'
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
...
Server: kong/0.5.3
{
  "consumer_id":"128a20ce-de29-4d11-cd51-ab005a137667",
  "key":"t3st-K3Y",
  ...
  "id":"27a02bbe-4a4c-4fd9-cfca-758acad7a724"
}
```

Bloque de código 19: Comandos para habilitar un *consumer* y su clave

Finalmente, probamos el requerimiento realizado anteriormente ahora utilizando la clave asignada al consumer creado, para corroborar que al incluir los datos de acceso correctos Kong nos provee la respuesta esperada por parte del servicio:

```
$ curl -i http://localhost:8000 --header "Host: mockbin.com"
 ↪  --header "apikey: t3st-K3Y"
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...
Via: kong/0.5.3
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mockbin by Mashape</title>
  ...
</html>
```

Bloque de código 20: Comandos para probar la autenticación mediante clave antes habilitada

Con esta breve prueba de concepto, hemos comprobado que tanto la instalación como el uso de Kong son sencillos, lo cual lo hace un gran candidato para su utilización como nodo central de la nueva arquitectura de la nube de servicios de la Universidad Nacional de La Plata.

3.3.2.4 Integración con nuestro diseño

En nuestra arquitectura tendríamos inicialmente un balanceador de carga delante de un cluster de instancias de Kong (junto con su correspondiente cluster de bases Cassandra con al menos dos instancias en réplica) como fachada para todas las peticiones a los servicios de la nube, y detrás de éste tendríamos las distintas aplicaciones que proveen los endpoints específicos. En caso de necesitar escalar, bastará con agregar más instancias al cluster de Kong y, de ser necesario, también al cluster de bases de datos Cassandra. La Figura 6 presenta en un diagrama la posible arquitectura propuesta incluyendo Kong como nodo central.

Al poder funcionar como *proxy* reverso, este producto nos será de gran utilidad al transicionar de la nube actual a la que estamos diseñando en este trabajo: nos permitiría enrutar todas las peticiones que lleguen al *hostname* `api2.dataintegration.unlp.edu.ar` a los servicios de la nube anterior y dirigir a las APIs de la nueva arquitectura aquellas que lleguen a `cloud.unlp.edu.ar`. De esta forma, la migración de las aplicaciones cliente a la nueva arquitectura sería gradual.

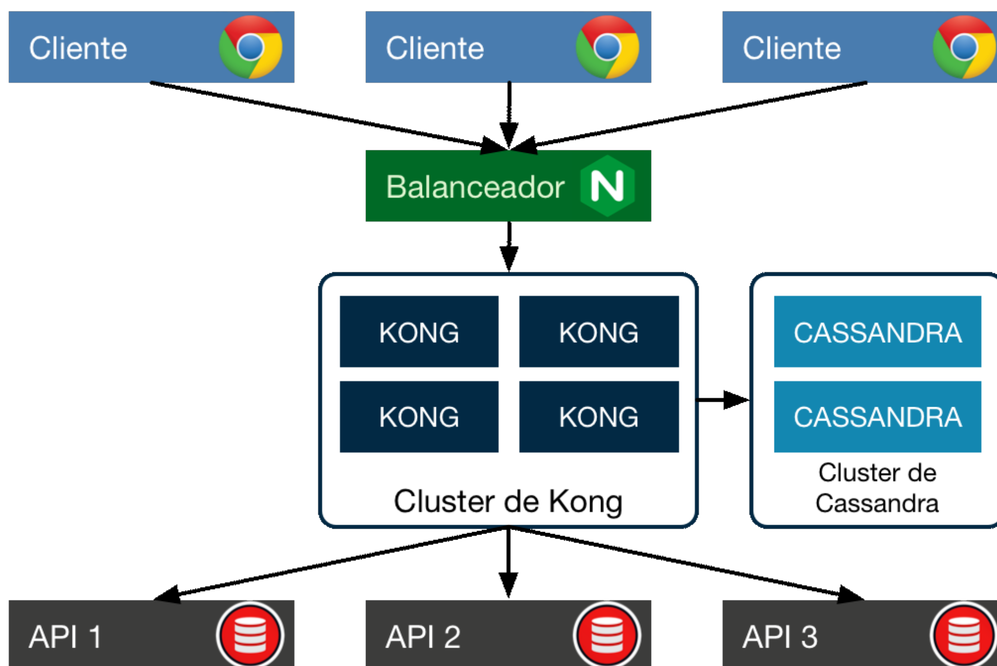


Figura 6: Esquema de integración de Kong en nuestra propuesta

3.3.3. API Umbrella

Es un producto open source desarrollado por la NREL (*National Renewable Energy Laboratory*) que actualmente está en uso por el Gobierno de Estados Unidos en la API su sitio de *open data*. Su función es la de un proxy que se ubica delante de las aplicaciones que proveen las APIs (los *API backends*) y procesa los requerimientos entrantes para esos servicios con un conjunto de reglas configurables, envía esos requerimientos a los proveedores correspondientes para finalmente enviar al cliente la respuesta. En la superficie es similar a Kong, pero su arquitectura interna es mucho más compleja, como veremos a continuación.

3.3.3.1 Licencia

Este producto está publicado bajo licencia MIT[28].

[28]https://github.com/NREL/api-umbrella/blob/master/LICENSE.txt

3.3.3.2 Arquitectura interna

Como se puede ver en la Figura 7: Arquitectura interna de API Umbrella, según la documentación oficial de la herramienta API Umbrella está compuesto de una serie de elementos que se enlazan secuencialmente, de más externos (cercanos al cliente) a más internos (cercanos a los *backends*):

- **Ruteo inicial con balanceo de carga:** Un servidor NGINX recibe las peticiones de los clientes y las deriva al *Gatekeeper* o al sitio para desarrolladores, según sea o no una petición a una API.
- ***Gatekeeper*:** Una aplicación hecha en Node.js[29] recibe peticiones para las APIs, aplica restricciones (autenticación, autorización y rate limiting) y transformaciones (reescritura de las peticiones originales), y registra datos de consumo de los servicios para poder utilizarlos con una herramienta de analíticas.
- **Cache HTTP:** Un servidor Varnish (Cf. Subsubsección 3.4.2) se ubica justo antes del enrutador de las APIs para evitar que éstas reciban accesos si la respuesta para el servicio solicitado ya se encuentra en esta cache compartida.
- **Ruteo de servicios con balanceo de carga:** Por último, otro NGINX funciona como enrutador de los endpoints reales que cada *backend* provee, pudiendo también balancear la carga entre distintas instancias de cada uno.

En particular, el *Gatekeeper* merece un gráfico propio describiendo su flujo de trabajo para aplicar restricciones, modificar las peticiones y registrar datos de analíticas. Dicha lógica se puede apreciar en detalle en la Figura 8: Lógica de decisión del *Gatekeeper* de API Umbrella. Del detalle del flujo de trabajo de este componente se desprenden el resto de los servicios que completan la arquitectura de API Umbrella:

- Bases de datos NoSQL MongoDB para almacenar configuración y claves de acceso (*API keys*).

---

[29] Node.js es un *runtime* para JavaScript basado en V8, el motor de ejecución de ese lenguaje del navegador Google Chrome, que permite ejecutar código escrito en JavaScript del lado del servidor. En la actualidad se encuentra en gran auge por las posibilidades de integración que provee, su velocidad y creciente comunidad de usuarios.
`https://nodejs.org/en/about`

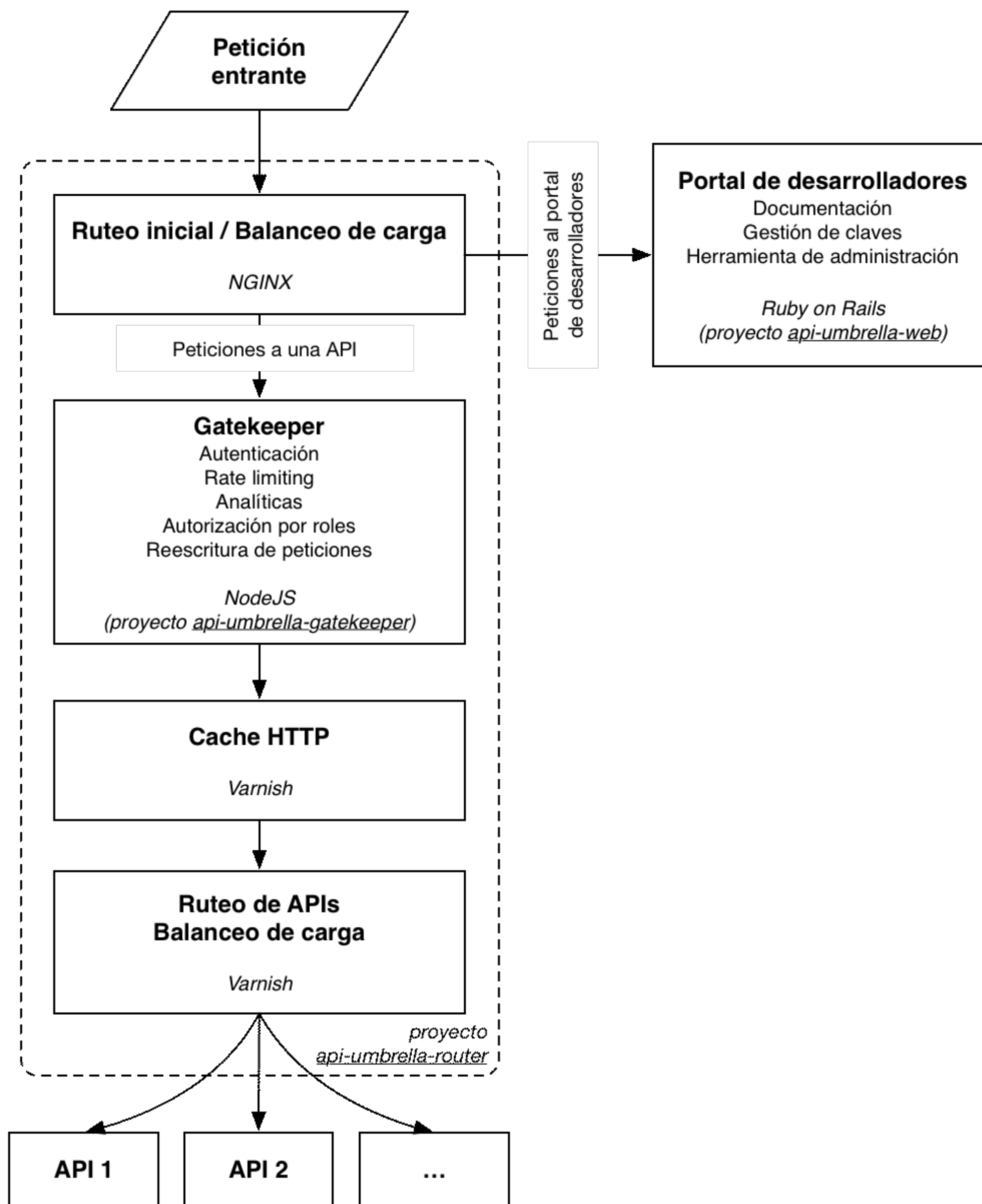


Figura 7: Arquitectura interna de API Umbrella

- Bases de datos clave-valor Redis para almacenar información volátil de los datos de las APIs y los clientes, como datos del *rate limiting*.
- Instancias de Elastic (anteriormente conocido como ElasticSearch), un almacen de datos para búsqueda en texto completo (*full text*) altamente eficiente, donde se almacenan los datos de acceso a los servicios para poder realizar análisis y monitoreo de los mismos.

3.3.3.3 Instalación y prueba

Al igual que en el caso de Kong, y ya que NREL provee una imagen oficial para ejecutar API Umbrella, utilizamos Docker para realizar una prueba de instalación del producto, evaluando sus características, facilidad de instalación y potencial. A continuación detallamos los pasos realizados.

```
# Bajar imagen del contenedor (sólo la primera vez)
$ docker pull nrel/api-umbrella
# Luego, en un directorio dedicado, crear el archivo de
↪  configuración para api umbrella
$ mkdir config
$ touch config/api-umbrella.yml
# Finalmente podemos ejecutar api umbrella usando docker
$ docker run -d -p 80:80 -p 443:443 --name api-umbrella -v
↪  $PWD/config:/etc/api-umbrella nrel/api-umbrella
```

Bloque de código 21: Preparación y arranque de API Umbrella

Una vez ejecutados los comandos anteriores, tendremos una instancia de API Umbrella en ejecución en el equipo local que podrá ser accedida desde los puertos `80` y `443` de `localhost`. Adicionalmente al planteo que Kong hace con respecto a la forma de administrar el producto mediante el uso de servicios de gestión, API Umbrella también utiliza un archivo de configuración en formato *Yet Another Markup Language* (YAML) que puede resultar más conveniente para poder utilizar un sistema de control de versiones para versionar su contenido, pero que también implica replicarlo en cada instancia del producto que querramos ejecutar.

En el archivo de configuración debemos mínimamente definir las direcciones de correo de los usuarios que podrán acceder a la interfaz web de administración que API Umbrella provee en `https://localhost/admin`. Una

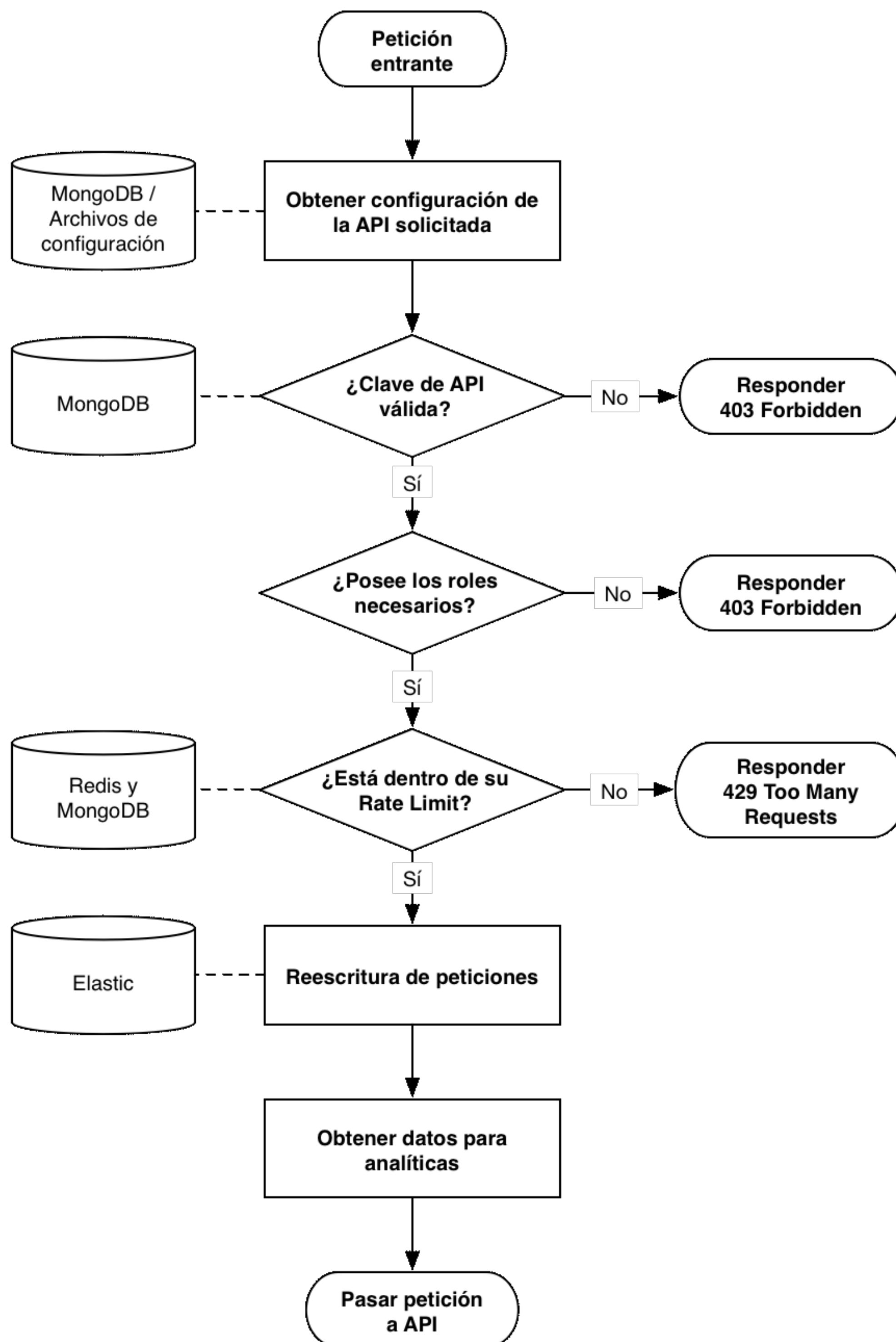


Figura 8: Lógica de decisión del *Gatekeeper* de API Umbrella

Figura 9: Interfaz web de carga de un *backend* de servicios de API Umbrella

vez identificados con alguna de esas cuentas de usuario, podremos proceder a utilizar la interfaz web para agregar una API para que API Umbrella haga de proxy de la misma. Siguiendo el ejemplo de la documentación del producto, agregamos la API de geocodificación de Google, indicando el *API backend* con la dirección del servicio provisto por Google y el prefijo de las URLs que nuestra instancia asociará a dicho servicio (utilizamos `/google/`), como puede observarse en la Figura 9. Luego de dar de alta esta información, publicamos los cambios mediante la interfaz web y así dejamos públicamente disponible el servicio de geocodificación en nuestra instancia de API Umbrella mediante el prefijo de URL `http://localhost/google/`.

Para poder utilizar este nuevo endpoint es necesaria una clave de acceso (*API key*), para lo cual API Umbrella provee una interfaz web dedicada. Completando los datos solicitados obtendremos nuestra clave de acceso para que nuestras peticiones sean autorizadas (en nuestro caso obtuvimos la clave `6dn6iL7xTuR1bvSWdRccpP7DCFp5X2OnhdYZzkrI`). Utilizando esta clave y la URL definida anteriormente, podemos realizar las peticiones detalladas en el bloque de código 22 para probar el correcto funcionamiento del producto, haciendo una petición sin credenciales primero y luego la misma petición incluyendo la información de autenticación. Al hacer esto, intentamos corro-

borar que la primer petición sea denegada mientras que la segunda obtenga la respuesta exitosa esperada.

```
# Primero probamos realizar una petición que no tenga una
# clave válida para corroborar que efectivamente falle
$ curl -i -k -G https://localhost/google/maps/api/geocode/json
 ↪  --data "address=La+Plata,+AR"
HTTP/1.1 403 Forbidden
Content-Type: application/json
...
X-Cache: MISS

{
  "error": {
    "code": "API_KEY_MISSING",
    "message": "No api_key was supplied. Get one at
     ↪  https://localhost"
  }
}

# Luego, intentamos con la clave obtenida desde la interfaz web
$ curl -i -k -G https://localhost/google/maps/api/geocode/json
 ↪  --data "address=La+Plata,+AR" --data
 ↪  "api_key=6dn6iL7xTuR1bvSWdRccpP7DCFp5X2OnhdYZzkrI"
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 996
Expires: Wed, 20 Jan 2016 15:56:56 GMT
Cache-Control: public, max-age=86400
Age: 201
...
X-Cache: HIT

{
  "results" : [ ... ]
}
```

Bloque de código 22: Prueba de uso de API Umbrella

Con estas pruebas pudimos comprobar varias situaciones:

- En primer lugar, que API Umbrella requiere por defecto el uso de cla-

ves de acceso para responder a los servicios para los que funciona de proxy. Este enfoque "seguro por defecto" es altamente deseable, ya que implica que para consumir la información que nuestros servicios proveen, se debe obtener una clave de acceso y por ende se debe identificar al cliente (*consumer*) de la información - en nuestro caso, serían las aplicaciones cliente las que identifiquemos, pero si llegásemos a proveer nuestra información a terceros, éstos deberían primero identificarse para obtener una clave de acceso y así permitirnos tener una noción de quién y cuánto utiliza nuestra información.

- En segundo lugar, que efectivamente hay un Varnish funcionando internamente a API Umbrella. Esto se hace evidente en las cabeceras que esta cache compartida agrega a las respuestas HTTP: `X-Cache` y `Age`. Adicionalmente, API Umbrella agrega las cabeceras correspondientes para indicar cuándo y por cuánto tiempo se pueden almacenar en caches intermedias las respuestas, mediante `Cache-Control` y `Expires`.

- También comprobamos que las respuestas tienen límites en la tasa de consultas que podemos hacer. Esto se hace evidente en las cabeceras `X-RateLimit-Limit` y `X-RateLimit-Remaining` que indican respectivamente el límite aplicable y la cantidad de peticiones que nos quedan disponibles hasta que se cumpla el período de tiempo del límite y esos contadores se reinicien.

- Y por último, aunque no por eso menos importante, que el producto funciona correctamente haciendo de proxy del servicio de geocodificación de Google, al devolvernos una respuesta correcta para nuestra consulta.

Luego de realizar algunas peticiones, podemos utilizar la potente herramienta de análisis de tráfico y respuestas que provee API Umbrella para tener una idea del uso que nuestra instancia está teniendo, junto con datos como quién, cuándo, cuánto, cómo y desde dónde está usando nuestras APIs. En la Figura 10 se puede apreciar, a modo de ejemplo, parte de la información que se puede obtener mediante esta herramienta.

3.3.3.4 Madurez

Al momento de comenzar a analizar los productos para el presente trabajo, API Umbrella se encontraba en activo desarrollo, definición y evolución. Su arquitectura interna era muy cambiante y aún le faltaban pruebas reales

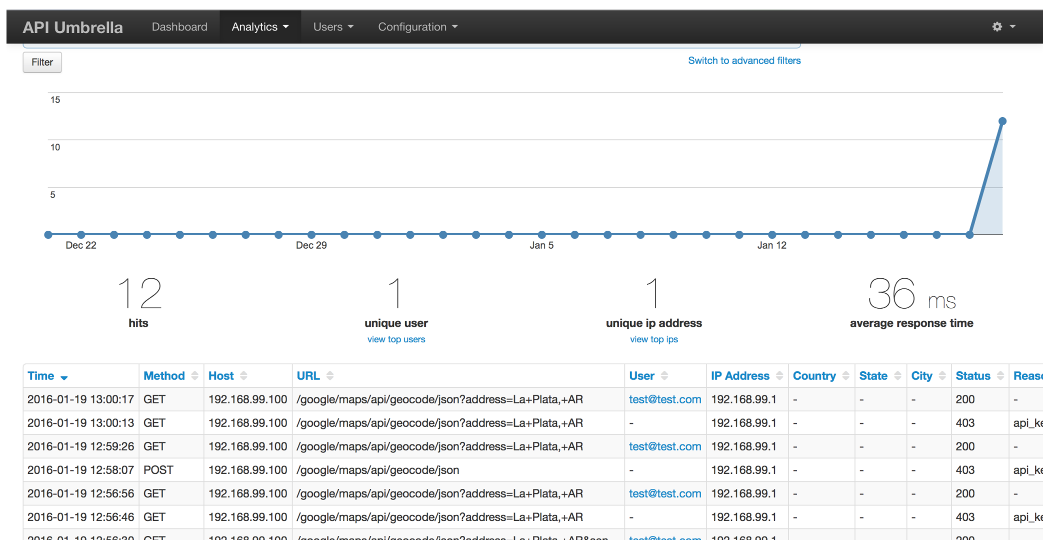


Figura 10: Interfaz web de análisis del uso de los servicios de API Umbrella

en ambientes productivos para poder cerrar las definiciones, por lo cual sus propios autores aconsejaban su uso únicamente para desarrollo[30].

Con la llegada de 2016 su desarrollo se desaceleró y la arquitectura quedó estabilizada, pero al momento de escritura del presente informe la herramienta carece de pruebas en producción que respalden las características que la documentación oficial describe y que nos den confianza que el producto se encuentre realmente maduro.

3.3.4. API Axle

Este es otro producto que comenzamos a analizar ya que parece tener características similares a Kong y API Umbrella:

- Funciona como un proxy reverso que se ubica delante de los servicios que queremos que provea.
- Utiliza NGINX como base para atender requerimientos entrantes.
- Implementa limitación de tasa de consultas, autenticación por clave de acceso, cache de respuestas, entre otros.

[30]Hasta Diciembre de 2015, la documentación oficial del producto mostraba la leyenda "*This version is for development only*" ("Esta versión es únicamente para desarrollo")

- Almacena en una base de datos Redis las estadísticas de uso.

3.3.4.1 Licencia

Este producto está licenciado bajo GPL v3[31].

3.3.4.2 Estado del proyecto

Lamentablemente, al indagar un poco más en profundidad sobre este producto notamos que el proyecto no tiene actividad desde mayo de 2015 y que su documentación es escasa e incompleta. La falta de ejemplos reales y de casos testigo que respalden el uso de la herramienta completan un panorama no muy alentador para considerar esta herramienta entre las opciones para nuestra nueva arquitectura.

3.3.5. Tyk

Tyk es un API Gateway liviano y plataforma de administración, todo en un mismo producto, que permite controlar quién tiene acceso a la API, cuándo y cómo. Está escrito en Go, es sencillo de configurar, y sus únicas dependencias son Redis y MongoDB (`v2.6` o superior, utilizado para el almacenamiento de los datos de acceso, de caracter opcional).

El API Gateway de Tyk se implementa delante de nuestras APIs para gestionar la autorización, el control de acceso y la limitación de acceso a nuestros servicios (*rate limiting*). De esta manera permite concentrarnos en el desarrollo de servicios para nuestras APIs, en lugar de implementar herramientas para la administración de la infraestructura. Esto nos permite enfocarnos en el desarrollo de los servicios, para luego integrarlos fácilmente al API Gateway.

Tyk posee un portal para desarrolladores que permite analizar quién y cómo utiliza nuestras APIs, limitar el acceso a los servicios (*rate limiting*), modificar parámetros de *caching*, y generar o revocar una *API key*, entre otras características. Todo esto de manera muy sencilla y centralizada, evitando trasladar parte de esta funcionalidad a los servicios.

---

[31] `https://github.com/apiaxle/apiaxle/blob/develop/GPL-3.0.txt`

3.3.5.1 Licencia

Tyk se encuentra publicado bajo licencia Mozilla Public License versión 2.0[32].

3.3.5.2 Características principales

A continuación presentamos un breve listado de las características principales que Tyk ofrece actualmente:

- RESTful API: provee una API RESTful para configurar Tyk mediante peticiones a ésta.
- Múltiples protocolos de acceso: Tyk soporta múltiples métodos de acceso a la API:
    - **Token-based:** autenticación mediante una clave de API. `https://tyk.io/v1.9/access-control/access-keys`
    - **HMAC:** permite autenticar la identidad del cliente (*consumer*) mediante firmas HMAC en los mensajes. `https://tyk.io/v1.9/access-control/hmac`
    - **Basic Auth:** autenticación mediante usuario y contraseña. `https://tyk.io/v1.9/access-control/basic-auth`
    - **OAuth2:** autenticación mediante el protocolo OAuth 2.0. `https://tyk.io/v1.9/access-control/oauth2`
    - **JSON Web Token:** provee autenticación mediante el uso del estándar JSON Web Tokens. `https://tyk.io/v1.9/access-control/json-web-tokens`
    - **Keyless:** provee acceso abierto, sin restricciones. `https://tyk.io/v1.9/access-control/keyless`
- Rate Limiting: permite configurar fácilmente el *rate limit* para cada *API Key*, definiendo la cantidad de peticiones por segundo y el tiempo de expiración (1 hora, 6 horas, 12 horas, 24 horas, 1 semana, 1 mes o que nunca expire). `https://tyk.io/v1.9/quotas-limits-security/access-control`

[32] La misma puede ser consultada en `https://www.mozilla.org/en-US/MPL/2.0`

- Policies: habilita a crear políticas para aplicar *quotas*, *rate limit* y *access rights* a un conjunto de claves de acceso.
  `https://tyk.io/v1.9/quotas-limits-security/limiting-access`

- Permisos por *endpoint* (*Path by path permissions*): permite configurar permisos de acceso para cada *endpoint*.

- Expiración de claves de acceso (*Key Expiry*): admite el control del tiempo de expiración de una clave de acceso.
  `https://tyk.io/v1.9/rest-api/api-key-management`

- Versionado de las APIs: permite versionar las APIs de tres maneras diferentes:

  - mediante una clave en el encabezado HTTP, por ejemplo `X-Api-Version`,
  - por URL o parámetro en el cuerpo de la petición,
  - por el primer elemento de una URL, ejemplo `/v1/resource/id` (donde `v1` es la versión).

  `https://tyk.io/v1.9/api-management/api-versioning`

- Analíticas: registro detallado de los accesos a las APIs.

- Zero downtime restarts: las configuraciones de Tyk pueden realizarse dinámicamente, reiniciar los servicios sin que esto afecte los requerimientos activos.

- Web Hooks: fácilmente se pueden integrar notificaciones y eventos que permitirán mejorar el monitoreo.

- Lista blanca de IPs: acceso autorizado únicamente a las direcciones IP indicadas en la lista blanca.

- Límite de tamaño: permite limitar el tamaño de las peticiones que llegan a las API, evitando que los sevicios sean saturados por peticiones.

- Health checks: provee verificación del estado de los nodos.

- Mock APIs: permite crear APIs de prueba, muy útil para el desarrollo de nuevos servicios.

- Soporte para API Blueprint: permite importar rápidamente una API en formato JSON.

- Soporte para Swagger: permite importar archivos que respeten *The OpenAPI Specification* (anteriormente conocida como *The Swagger specification*).
- Transformaciones de solicitudes y respuestas: permite utilizar templates para transformar datos y agregar o quitar cabeceras *al vuelo*.
- Caching: permite implementar una cache para las respuestas por *endpoint* o globalmente, incrementando la velocidad de respuesta, al mismo tiempo que se disminuye la carga de las API.
`https://tyk.io/v1.9/api-management/caching`
- Documentación de la API: el portal de Tyk soporta API Blueprint y Swagger.
- Endpoints virtuales: como AWS Lambda Functions[33], permite correr fragmentos de JavaScript en los *endpoints* para manejar interacciones complejas de servicios, tales como solicitud de procesamiento por lotes.
`https://tyk.io/v1.9/api-management/virtual-endpoints`
- Foco en microservicios: permite implementar el patrón *circuit breaker*, *hard timeouts* y *round robin*, para balancear la carga en el acceso a los servicios.
`https://tyk.io/v1.9/api-management/circuit-breakers`
`https://tyk.io/v1.9/api-management/load-balancing`
- Uptime Awareness: Tyk activamente monitorea los *endpoints* de las APIs y notifica cuando alguno de éstos se encuentra fuera de servicio.
`https://tyk.io/v1.9/uptime-tests/uptime-tests`

3.3.5.3 Instalación y prueba

A continuación se detallan los pasos necesarios para la instalación y ejecución de Tyk, basándonos en su documentación oficial [34].

Tyk puede instalarse de varias maneras:

- Instalar Tyk en Ubuntu desde paquetes.

[33]Cf. `https://aws.amazon.com/es/lambda/details`

[34]`https://tyk.io`

- Instalar Tyk en Redhat o CentOS usando paquetes RPM.
- Instalar Tyk desde una imagen Docker.

En nuestro caso particular se optó por instalar Tyk en Ubuntu 14.04 desde paquetes, a continuación desarrollaremos esta forma de instalación.

Inicialmente el equipo donde se instalará Tyk debe debe cumplir con los siguientes requisitos:

- Asegurarse que el puerto 8080 esté abierto: este puerto es utlizado por el API Gateway.
- Asegurarse que el puerto 3000 esté abierto: este puerto es utlizado por el dashboard, quien provee la GUI y el portal para desarrolladores.

Primero preparamos el ambiente del equipo para obtener los paquetes a instalar con los comandos descriptos en el bloque de código 23:

```
$ curl https://packagecloud.io/gpg.key | sudo apt-key add -
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
 ↪  7F0CEB10
$ sudo apt-get update
$ sudo apt-get install -y apt-transport-https
$ cd /etc/apt/sources.list.d/
$ echo "deb http://repo.mongodb.org/apt/ubuntu
 ↪  trusty/mongodb-org/3.0 multiverse" | sudo tee
 ↪  mongodb-org-3.0.list
$ echo "deb https://packagecloud.io/tyk/tyk-gateway/ubuntu/ trusty
 ↪  main" | sudo tee tyk_tyk-gateway.list
$ echo "deb-src https://packagecloud.io/tyk/tyk-gateway/ubuntu/
 ↪  trusty main" | sudo tee -a tyk_tyk-gateway.list
$ echo "deb https://packagecloud.io/tyk/tyk-dashboard/ubuntu/
 ↪  trusty main" | sudo tee tyk_tyk-dashboard.list
$ echo "deb-src https://packagecloud.io/tyk/tyk-dashboard/ubuntu/
 ↪  trusty main" | sudo tee -a tyk_tyk-dashboard.list
$ sudo apt-get update
```

Bloque de código 23: Preparación del servidor para instalar Tyk

Luego, procedemos a instalar Tyk y sus dependencias con los comandos del bloque de código 24:

```
# 1. Estamos listos para instalar Tyk Gateway y tyk Dashboard,
 ↪  junto con sus principales dependencias: Redis y MongoDB. Para
 ↪  instalar todo ejecutamos el siguiten comando:
$ sudo apt-get install -y mongodb-org redis-server tyk-gateway
 ↪  tyk-dashboard

# 2. Configuración Tyk Gateway
$ sudo /opt/tyk-gateway/install/setup.sh --dashboard=1
 ↪  --listenport=8080 --redishost=localhost --redisport=6379
 ↪  --domain="" --mongo=mongodb://localhost/tyk_analytics

# 3. Configuración Tyk Dashboard
$ sudo /opt/tyk-dashboard/install/setup.sh --listenport=3000
 ↪  --redishost=localhost --redisport=6379
 ↪  --mongo=mongodb://localhost/tyk_analytics
 ↪  --tyk_api_hostname=$HOSTNAME
 ↪  --tyk_node_hostname=http://localhost --tyk_node_port=8080
 ↪  --portal_root=/portal --domain="tesis.desarrollo.unlp.edu.ar"

# 4. Iniciar Tyk y Tyk dashboard:
$ sudo service tyk-gateway start
$ sudo service tyk-dashboard start
```

Bloque de código 24: Instalación y arranque de Tyk

Con los pasos anteriores, tuvimos una instancia de Tyk en ejecución para realizar pruebas.

3.3.5.4 Integración con nuestro diseño

Para el nuevo diseño de la arquitectura, se implementaría el API Gateway de Tyk como un nodo central, en el cual se enrutarían todas las peticiones realizadas desde los diferentes clientes. Detrás de Tyk tendríamos replicadas distintas instancias de las APIs, las cuales nos permitirían escalar horizontalmente de forma sencilla, ya que Tyk se encargaría de realizar el balanceo de la carga a cualquiera de estas instancias replicadas. En la Figura 11 se puede apreciar la posible arquitectura incluyendo a Tyk como nodo central.

La migración de la vieja nube a la nueva arquitectura se iría realizando de manera progresiva, cambiando la lógica de acceso antigua por la nueva en cada aplicación. De esta manera tendríamos conviviendo al mismo tiempo la vieja nube y la nueva arquitectura que en el presente trabajo estamos

definiendo.

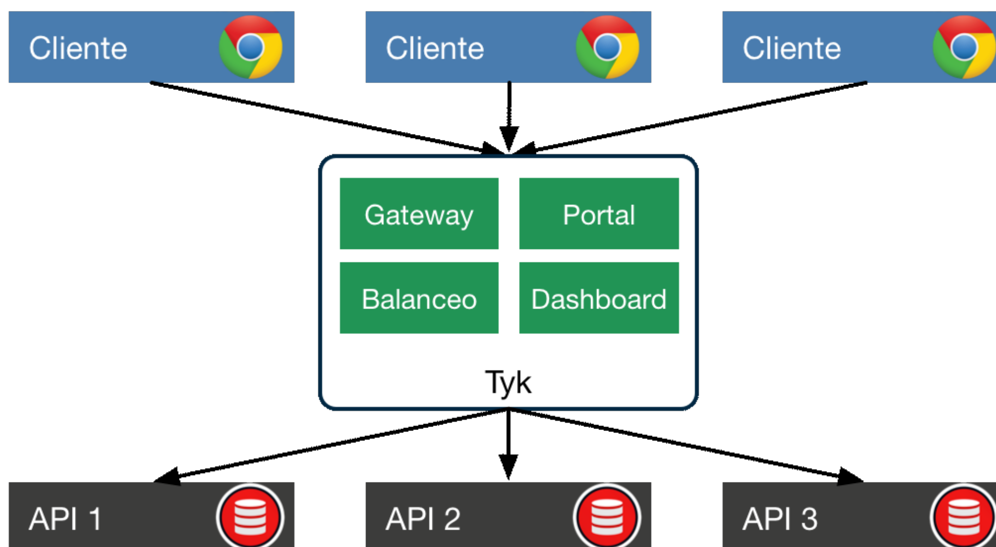


Figura 11: Esquema de integración de Tyk en nuestra propuesta

3.3.6. WSO2 ESB

WSO2 es una compañía de tecnología *open source*, que proporciona *middleware* de SOA. Es conocida por sus productos: *Enterprise Service Bus*, *API Management* y *Governance* por nombrar algunos, utilizados por eBay, Boeing y Experian entre otros.

WSO2 fue fundada por el Dr. Sanjiva Weerawarana y Paul Fremantle en Agosto de 2005, y ha sido respaldada por Intel Capital, Toba Capital y Pacific Controls. WSO2 ESB es rápido, ligero y versátil, 100 % *open source*, está basado en los proyectos Apache Synapse[35] y Apache Axis2[36], y ha demostrado interoperabilidad con la mayoría de los *stack* de Web Services, incluyendo Microsoft .NET WCF.

Utilizando WSO2 ESB se pueden implementar una gran variedad de patrones de integración empresarial (*Enterprise Integration Patterns*, EIPs), incluyendo filtrado, transformaciones y ruteo SOAP, binario, XML y de mensajes de texto, atravesando los sistemas de la organización sobre HTTP,

[35]`http://synapse.apache.org`

[36]`http://axis.apache.org/axis2/java/core`

HTTPS, JMS, mail, entre otros.

3.3.6.1 Licencia

WSO2 ESB se encuentra publicado bajo licencia Apache 2.0[37].

3.3.6.2 Características principales

- Adaptadores a servicios en la nube: posee una tienda con más de 100 conectores[38] como por ejemplo: CRM, ERP y redes sociales, entre otros.
- Soporte para diferentes tipos de transporte: HTTP, HTTPS, POP, IMAP, SMTP, JMS, AMQP, RabbitMQ, FIX, TCP, UDP, FTPS, SFTP, MLLP, SMS, MQTT y Apache Kafka.
- Formatos y protocolos: JSON, XML, SOAP 1.1, SOAP 1.2, WS-*, HTML, EDI, HL7, OAGIS, Hessian, Text, JPEG, MP4.
- Adaptadores para productos comerciales: SAP BAPI & IDoc, IBM WebSphere MQ, Oracle AQ y MSMQ.
- Ruteo: basado en headers, basado en contenido, basado en reglas y basado en prioridades.
- Transformaciones: XSLT 1.02.0, XPath, XQuery y Smooks.
- Alto rendimiento, alta disponibilidad, escalabilidad y estabilidad, soporta una cantidad de conexiones HTTP(S) concurrentes no bloqueantes en el orden de 1000 por servidor.
- Baja latencia en escenarios de alto rendimiento.
- Permite escalar horizontalmente.
- Estabilidad de ejecución a largo plazo con baja utilización de recursos.
- Permite balanceo de carga y failover para *endpoints* de alta disponibilidad.

[37] La misma puede ser consultada en `http://www.apache.org/licenses/LICENSE-2.0`

[38] `https://store.wso2.com/store/assets/esbconnector`

3.3.6.3 Instalación

A continuación se detallan los pasos necesarios para la instalación y ejecución de WSO2 ESB, basándonos en su documentación oficial [39]. Previo a la instalación se debe verificar que se cumpla con los siguientes requisitos previos[40]:

- Java SE Development Kit (JDK)
- Apache ActiveMQ JMS Provider
- Apache Ant
- Apache Maven

Paso 1. Obtener el Pack de instalación.
Descargar la última versión de WSO2 ESB (para la descarga seguir las instrucciones [41]).

```
$ wget http://dist.wso2.org/products/esb/java/4.0.3/wso2esb-4.0.3-
↪  src.zip
```

Bloque de código 25: Comando para descargar los fuentes de WSO2 ESB

Paso 2. Extraer el archivo.
Una vez descargados los fuentes, extraer los archivos de instalación en el directorio `home`:

```
$ unzip wso2esb-4.0.3-src.zip
```

Bloque de código 26: Comando para descomprimir zip

Paso 3. Construir el WSO2 Enterprise Service Bus.
Ejecutar el siguiente comando para construir WSO2 ESB en el directorio de instalación:

---

[39] https://docs.wso2.com/display/ESB403/Installing+ESB+on+Linux+and+Solaris+from+Source+Distribution

[40] https://docs.wso2.com/display/ESB403/ESB+Installation+Prerequisites

[41] Las mismas pueden ser consultadas en https://docs.wso2.com/display/ESB403/Obtaining+ESB

```
$ mvn clean install
```

Bloque de código 27: Comando para construir WSO2 ESB

Paso 4. Configuración de la variable `JAVA_HOME`.
Es necesaria la variable de entorno `JAVA_HOME` para poder ejecutar WSO2 ESB. La variable apunta al directorio donde se encuentra instalado *Java Development Kit (JDK).*

Editar el archivo `.bashrc` del directorio `home` y agregar la variable de entorno `JAVA_HOME`. Para configurar la variable `JAVA_HOME`, seguir los siguientes pasos:

1. Abrir el archivo `.bashrc` .
2. Agregar las siguientes líneas al final del archivo:

```
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH="$JAVA_HOME/bin:$PATH"
```

Bloque de código 28: Comandos para configurar variables de entorno

3. Guardar los cambios en el archivo y cargarlos en la sesión actual con el comando `source .bashrc`.
4. Para verficar la correcta configuración de la variable `JAVA_HOME`, ejecutar el siguiente comando:

```
$ echo $JAVA_HOME
```

Bloque de código 29: Verificamos la variable de entorno JAVA_HOME

Paso 5. Acceder a la consola de administración del ESB.
El ESB ya se encuentra instalado, ahora procedemos a iniciar el servicio, para lo cual debemos realizar los siguientes pasos:

1. Realizar una conección SSH al servidor.
2. Ir al directorio `<ESB_HOME>/bin`, donde `<ESB_HOME>` es el directorio donde se encuentran los archivos de WSO2 ESB.

3. Ejecutar el siguiente comando para iniciar el ESB:

```
$ sh ./wso2server.sh
```

Bloque de código 30: Comando para iniciar el servicio WSO2 ESB

3.3.6.4 Integración con nuestro diseño

Para el nuevo diseño de la arquitectura, se implementaría WSO2 ESB como nodo central, en el cual se enrutarían todas las peticiones realizadas desde los diferentes clientes. Detrás de WSO2 ESB tendríamos replicadas N instancias de las APIs, las cuales nos permitirían escalar horizontalmente de forma sencilla, dado que WSO2 ESB se encargaría de realizar el balanceo de la carga a cualquiera de estas instancias replicadas. Delante del WSO2 ESB se instalaría WSO2 API Manager para administrar las APIs, gestionar la autorización y el control de acceso a las mismas.

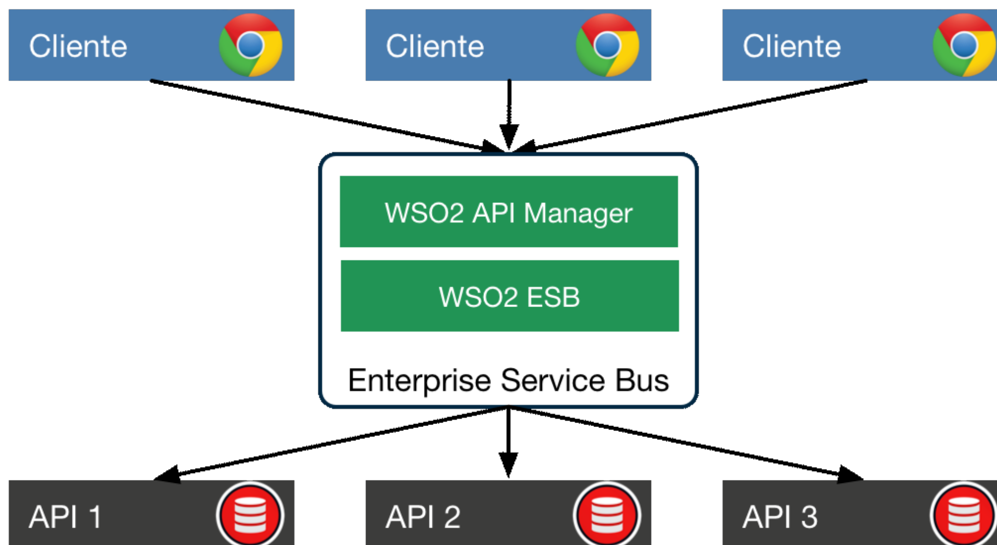


Figura 12: Esquema de integración de WSO2 ESB en nuestra propuesta

3.3.7. Conclusión

En esta sección hemos analizado variantes para abordar un mismo problema, la centralización de la distribución de conexiones entre los clientes y

los servicios sin generar nuevos cuellos de botella. En el análisis hemos considerado desde un ESB completo, con toda su estructura, hasta tecnologías que simplemente funcionaban de proxy reverso altamente performante con la posibilidad de administrarlo y algún agregado más.

Este amplio espectro de opciones nos ayudó a comprender mejor nuestras necesidades y, por ende, a tomar una decisión basada en las mismas y no simplemente en las tendencias de la industria. En un extremo del espectro de opciones nos encontramos con un ESB completo, lo cual nos resulta demasiado para el planteo que estamos haciendo, ya que incluirlo agregaría capas innecesarias de complejidad a nuestro nodo central. En el opuesto nos encontramos con productos en extremo simples (o tal vez incompletos) que nos generan la necesidad de implementar nosotros algunos elementos para nuestra arquitectura, como ser analíticas o monitoreo de *endpoints*.

En un punto relativamente intermedio del rango, encontramos opciones como API Umbrella y Tyk, que nos brindan un buen equilibrio entre prestaciones y complejidad, sin sacrificar nuestras necesidades para este nodo central. Entre estos productos, el aspecto determinante para nuestra elección se basa en su madurez: al momento de realizar este trabajo, API Umbrella no tiene un nivel de estabilidad adecuado para nuestras necesidades, mientras que Tyk sí, por lo cual elegimos este último para implementar el nodo central.

Al ahondar en SOA e investigar diferentes tecnologías y patrones de arquitectura de software (*layered architecture*, *event-driven architecture*, *microservices architecture*), observamos en el patrón de microservicios una mejora sustancial, en simplicidad y alcance, que se adecuaba a nuestras necesidades, por eso decidimos optar por un *API Gateway* que funcione como un "message broker liviano", en lugar de un ESB completo.

Si bien WSO2 ESB se presenta como un producto robusto y con amplias posibilidades de escalabilidad, se trata de una solución demasiado compleja para el alcance de este trabajo, cuyo foco es rediseñar la arquitectura de la nube de servicios de la UNLP.

En este sentido, Tyk cumple con todas las tareas que realiza un ESB (proveer conectividad, ruteo sencillo, manejo de seguridad, fiabilidad en los servicios, monitoreo y registro de actividades), menos en la transformación de datos y ruteo inteligente. Esto resulta suficiente para el alcance de este trabajo, el cual nos permitirá lograr experiencia en la materia, para luego decidir, en caso de ser necesario, cambiar el "message broker" por un ESB robusto, como podría ser Mule ESB, WSO2 ESB o Apache Synapse (en el

cual se basa WSO2 ESB), entre otros. Cabe aclarar, que este cambio no sería trascendental, ya que se trata de una capa de la arquitectura totalmente desacoplada que provee varios servicios (proveer conectividad, ruteo, manejo de seguridad, fiabilidad en los servicios, monitoreo y registro de actividades), los cuales se encuentran encapsulados o agrupados, lo que permitirá ser reemplazada sin afectar el resto de la arquitectura. Esto es posible debido a que desde el inicio, basamos el diseño de la arquitectura en los principios que propone SOA, logrando obtener los beneficios que este marco de diseño para la integración de aplicaciones propone.

3.4. Para la cache compartida

Un componente importante para la implementación de la arquitectura de la nueva nube de servicios es la memoria cache compartida, la cual evitará accesos innecesarios a los backends, obteniendo mejores tiempos de respuesta. Según la RFC 7234[42], una memoria cache almacena respuestas en pos de reducir el consumo del ancho de banda y el tiempo de respuesta. Asimismo, una memoria cache *compartida* es una cache que almacena respuestas para ser usadas por más de un cliente.

En este apartado nos centraremos en los tipos de memorias compartidas *proxy server* y *reverse proxy server*. Un *proxy server*, también conocido como *forward proxy server*, actúa como un proxy para los dispositivos que se conectan a él. Una implementación típica puede ser un *forward proxy* que provee acceso a Internet a un conjunto de clientes. En cambio, un *reverse proxy server* es un servidor proxy que recupera recursos desde uno o más servidores. Estos recursos se devuelven al cliente como si se hubieran originado desde el propio servidor, es decir, actúa como intermediario entre los clientes y servidores. Los *reverse proxies* se implementan en las proximidades de los *web servers*, a veces realizan tareas como balanceo de carga, autenticación o *caching*. En la Figura 13 se puede apreciar gráficamente la distinción entre estos tipos de memoria compartida.

[42] Este documento define las caches HTTP y las cabeceras de control de cache permitidas en la versión `1.1` de ese protocolo
`https://tools.ietf.org/html/rfc7234`

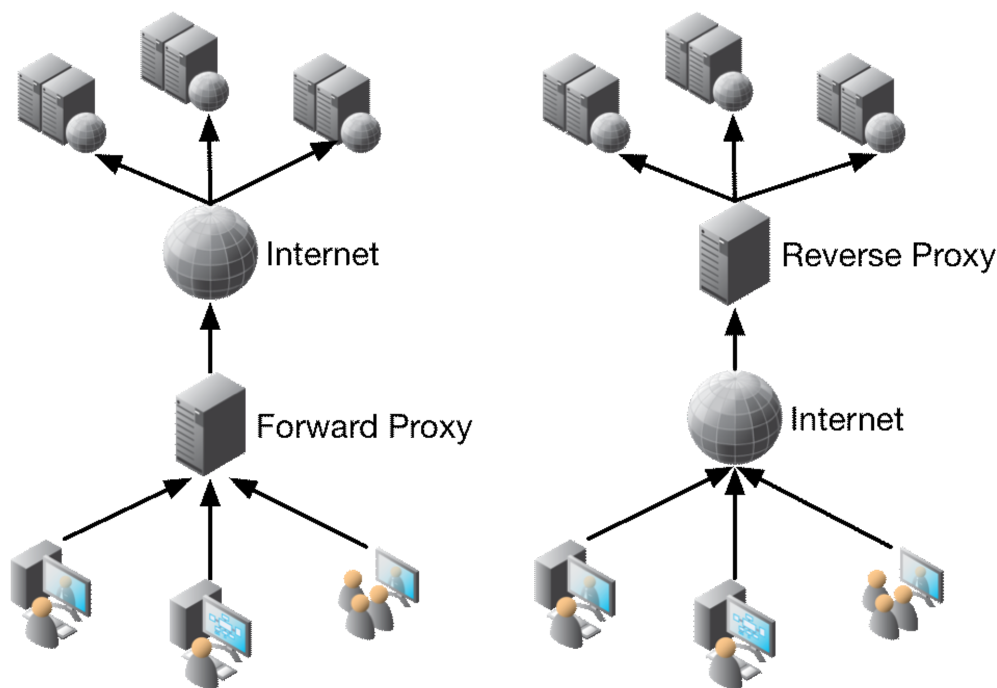


Figura 13: Forward proxy y reverse proxy

3.4.1. Squid

Squid fue originalmente desarrollado para el proyecto *Harvest* de la Universidad de Colorado Boulder. Duane Wessels bifurcó el proyecto de la "última versión pre-comercial de Harvest", el cual fue renombrado a Squid para evitar confusiones con la versión comercial Cached 2.0, que se convirtió luego en NetCache. La versión 1.0.0 de Squid fue liberada en Julio 1996.

Squid es un *proxy server* (*forward proxy server*) para la Web, soporta HTTP, *HyperText Transfer Protocol Secure* (HTTPS), FTP, entre otros. Reduce el ancho de banda y mejora los tiempos de respuesta por el almacenamiento en caché y la reutilización de las páginas web frecuentemente solicitadas. Puede implementarse en la mayoría de los sistemas operativos disponibles, incluyendo Windows y está disponible bajo licencia GNU GPL.

Squid optimiza el flujo de datos entre el cliente y el servidor, utilizando el contenido en caches frecuentemente usado, ahorrando de esta manera ancho de banda y obteniendo mejoras en el rendimiento de los clientes.

3.4.1.1 Instalación

A continuación se detallan los pasos para instalar Squid en un servidor con Ubuntu 14.04. Antes de comenzar con la instalación y configuración de Squid, debemos actualizar el software del servidor a su última versión, como se muestra en el bloque de código 31:

```
$ sudo apt-get update && sudo apt-get -y upgrade
```

Bloque de código 31: Actualización del sistema de base

Una vez que hemos terminado la actulización del equipo, estamos en condiciones de iniciar la instalación de Squid. Squid se encuentra disponible en los repositorios de Ubuntu, para instalarlo en el servidor debemos ejecutar el comando del bloque de código 32:

```
$ sudo apt-get install squid
```

Bloque de código 32: Instalación de Squid

La configuración principal de Squid se encuentra en /etc/squid3/squid.conf. Antes de realizar cualquier cambio en la configuración original, realizamos una copia del archivo con el comando del bloque de código 33.

```
$ sudo cp /etc/squid3/squid.conf /etc/squid3/squid.conf.orig
```

Bloque de código 33: Copia de respando de configuración de Squid

A continuación procedemos a modificar el archivo principal de configuración de Squid con el editor `nano` (bloque de código 34):

```
$ sudo nano /etc/squid3/squid.conf
```

Bloque de código 34: Configuración de Squid

Lo primero que debemos configurar es el puerto en el que Squid estará escuchando las peticiones, por defecto, Squid escucha las peticiones en el puerto `3128`. Para cambiar el puerto por defecto, debemos editar la directiva

`http_port`. En nuestro caso particular, configuramos el puerto `8888`, modificando la línea de esta directiva para que quede de la siguiente manera:
`http_port 8888`

Por defecto, el servidor proxy HTTP no permite el acceso a nadie. Para permitir el acceso al servidor desde cualquier IP, debemos editar la directiva `http_access` para que quede de la siguiente manera:
`http_access allow all`

Una vez que hemos realizado las configuraciones necesarias, debemos guardar los cambios y reiniciar el servicio de Squid, para que tome los cambios. Para reiniciar el servio ejecutamos el comando del bloque de código 35:

```
$ sudo service squid3 restart
```

Bloque de código 35: Renicio del servicio Squid

Para verificar el funcionamiento del servidor proxy, configuramos manualmente los datos de nuestro proxy en el navegador, ingresando la IP del servidor proxy y el puerto anteriomente configurado. En caso de que tengamos problemas, podemos ver log `access.log` para obtener más información como se indica en el bloque de código 36:

```
$ sudo tail -f /var/log/squid3/access.log
```

Bloque de código 36: Verificación del funcionamiento de Squid

3.4.2. Varnish

Varnish es un *proxy* reverso HTTP, a veces referenciado como acelerador de HTTP o a *web accelerator*, que almacena archivos y fragmentos de archivos en memoria, lo que reduce el tiempo de respuesta y el ancho de banda de red para las mismas solicitudes.[14, p. 20]

El proyecto Varnish fue iniciado por Verdens Gang en el 2005, contó con la gerencia, infraestructura y desarrollos adicionales aportados por la comunidad Noruega de Linux, que más tarde pasó a una empresa independiente, Varnish Software bajo licencia BSD.

Como mencionamos anteriormente, Varnish es un acelerador de aplicaciones web, que se instala delante de cualquier servidor HTTP y se configura

para almacenar en la caché del servidor una copia del recurso solicitado. Pensado para mejorar el rendimiento de aplicaciones web con contenidos pesados y APIs altamente consumidas, éste será nuestro caso.

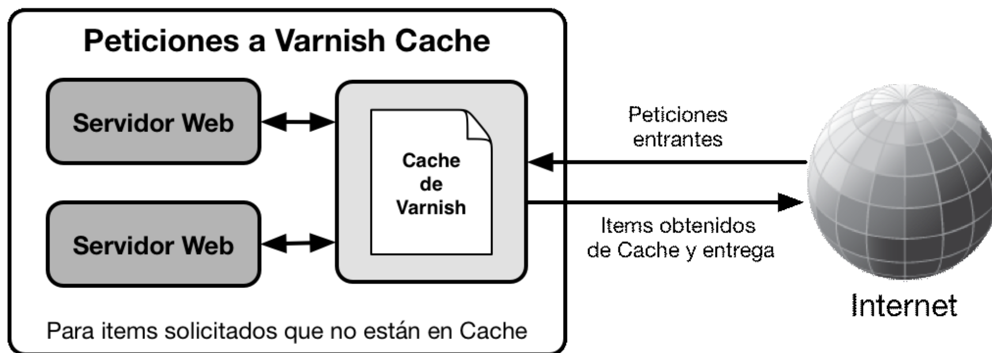


Figura 14: Varnish Reverse Proxy

3.4.2.1 Instalación

Varnish se distribuye en los respositorios de paquetes de Ubuntu, pero puede suceder que el paquete se encuentre desactualizado por lo tanto se recomienda usar los respositorios de paquestes de `varnish-cache.org`.

Para usar los respositorios de `varnish-cache.org` e instalar Varnish en Ubuntu 14.04, debemos ejecuatar la secuencia de comandos del bloque de código 37:

```
$ sudo apt-get install apt-transport-https
$ curl https://repo.varnish-cache.org/GPG-key.txt | sudo apt-key
↪  add -
$ sudo echo "deb https://repo.varnish-cache.org/ubuntu/ trusty
↪  varnish-4.1" >> /etc/apt/sources.list.d/varnish-cache.list
$ sudo apt-get update && sudo apt-get install varnish
```

Bloque de código 37: Instalación de Varnish

3.4.3. Conclusión

A continuación se detallan algunas de las ventajas de Varnish por sobre Squid:

- Poul-Henning Kamp explica en [7] que Varnish delega en el kernel la administración de la memoria virtual, mientras que Squid intenta mantener separados el almacenamiento en disco y la memoria cache, lo que genera conflictos acerca de lo que debe paginarse a disco.

- Mejor performance y escalabilidad, Squid corre como un único proceso en un núcleo de CPU, mientras que Varnish utiliza un hilo (*thread*) para cada conexión de cliente.

- Varnish utiliza VCL (*Varnish Configuration Language*), que es un potente lenguaje de configuración que permite definir políticas de *caching*, las mismas luego serán traducidas a código C, para más tarde ser compiladas. En el caso de Squid, algunas configuraciones resultan demasiado complejas, y otras incluso no pueden realizarse.

- Actualmente Varnish es utilizado por una gran cantidad de sitios web con alta demanda de tráfico como pueden ser: The New York Times, The Guardian, The Hindu, y sitios de redes sociales y contenidos como Wikipedia, Facebook, Twitter, Vimeo, Tumblr, entre otros.

- Squid no soporta ESI (*Edge-Side Includes*, un mecanismo de caching con invalidaciones parciales dentro de un mismo recurso) o invalidaciones explícitas.

- Squid es un *forward proxy* que puede ser configurado como *reverse proxy*, mientras que Varnish es un *reverse proxy* que adicionalmente puede funcionar como *forward proxy*.

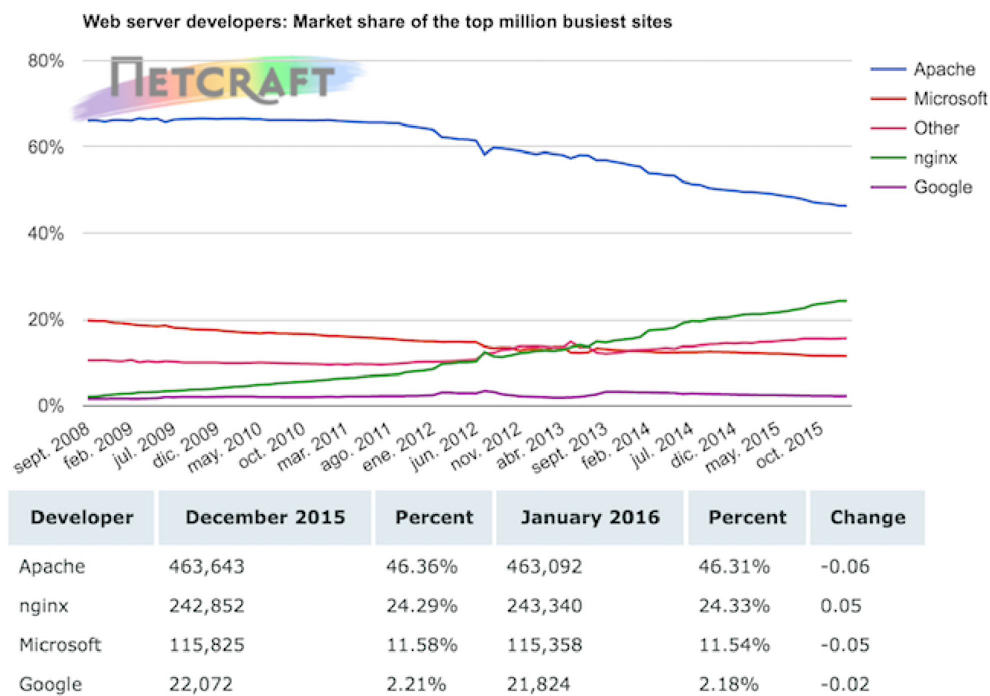
Además de estas ventajas, el equipo de desarrollo del CeSPI cuenta con la experiencia necesaria en la utlización de Varnish, ya que actualmente se encuentra implementado en producción para desarrollos realizados para la Universidad. Por lo tanto, para nuestra implementación se instalará Varnish delante del balanceador de carga y de las instancias replicadas en las que se encuentran las diferentes APIs, de esta manera, los accesos recurrentes a los mismos *endpoints*, serán obtenidos desde la cache de Varnish, en lugar de acceder a cualquiera de las APIs, logrando obtener mejores tiempos de respuesta.

3.5. Para balancear la carga

Al planificar una arquitectura escalable, necesitamos tener una forma de agregar o quitar nodos de manera dinámica de forma que se logre elasticidad

para escalar horizontalmente. La forma más común de lograr esto es utilizando un nodo dedicado a la distribución del trabajo entrante entre aquellos capaces de atender la tarea entrante, lo cual no sólo abstrae a las capas anteriores de la arquitectura sobre cuántos o qué nodos se encargan de realizar ese trabajo, si no que además permite distribuir más equitativamente esa carga entre ellos.

Esta forma de balanceo y escalabilidad horizontal es la que utilizaremos en nuestra propuesta, y a tal fin hemos considerado como opciones las dos herramientas de código abierto más populares y probadas de los últimos años. En los apartados presentados a continuación analizaremos brevemente cada una a fin de elegir una para utilizar en nuestra propuesta. En la Figura 15 se puede observar que del millón de sitios web más concurridos, los dos primeros servidores web que se utilizan son Apache y NGINX, con una tendencia en alza de este último, según datos del sitio *netcraft*[43].



Developer	December 2015	Percent	January 2016	Percent	Change
Apache	463,643	46.36%	463,092	46.31%	-0.06
nginx	242,852	24.29%	243,340	24.33%	0.05
Microsoft	115,825	11.58%	115,358	11.54%	-0.05
Google	22,072	2.21%	21,824	2.18%	-0.02

Figura 15: Resultados de la encuesta de uso de servidores web de netcraft, enero 2016

[43] http://news.netcraft.com/archives/2016/01/26/january-2016-web-server-survey.html

3.5.1. Apache

Apache es un servidor web que surgió en 1995 y desde 1999 es desarrollado por la *Apache Foundation*, fundación a la que dio origen. Desde su creación y durante muchos años, este producto fue el estándar *de facto* a la hora de instalar servidores web, la opción *indudable* cuando se buscaban tecnologías abiertas. Si bien en el presente existen otros grandes contendientes que se acercan lentamente para quitarle ese lugar, Apache ocupa el primer puesto entre los servidores web activos del mundo, según datos del sitio netcraft como puede verse en la Figura 15.

En su concepción más pura, Apache es un producto que funciona como servidor web, ya sea de sitios estáticos o dinámicos, lo cual lo dejaría fuera de nuestro análisis para este punto de nuestra arquitectura; pero esta funcionalidad básica puede ser extendida mediante la adición de *módulos* que tienen propósitos específicos, como para nuestro caso, existe el módulo `mod_proxy_balancer`[44], el cual dota al servidor de la lógica necesaria para que funcione como balanceador de carga.

A la hora de considerar Apache como opción, debemos saber su modelo de diseño y las consecuencias que éste puede tener en nuestra arquitectura. El modelo de funcionamiento que posee este producto se basa en la creación de procesos e hilos de sistema operativo para manejar las conexiones entrantes, donde el número máximo de procesos que pueda crear se configura de antemano y es un factor determinante en la degradación de performance que puede sufrir el equipo donde se ejecuta este servidor, ya que si se crean demasiados procesos de Apache puede ocurrir que el equipo se quede sin memoria principal disponible y deba comenzar a realizar *swapping* de la memoria al disco y viceversa. En el otro extremo, si se configura un límite bajo a esa cantidad de procesos disponibles, Apache rechazará las conexiones entrantes una vez que haya alcanzado ese número.

Esta limitación que presenta el modelo de Apache no es algo que podamos ignorar, ya que en momentos de alta carga puede llevar toda la arquitectura de la nueva nube de servicios a un colapso por un funcionamiento poco performante del nodo encargado de balancear la carga entrante.

---

[44] `http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html`

3.5.2. NGINX

Este servidor web, que surgió en 2002 y se liberó en el 2004 como proyecto *open source*, fue diseñado para solucionar las limitaciones que Apache tiene, principalmente en cuanto al excesivo uso de recursos y los problemas de escalabilidad que éstos acarrean.

La principal flexibilidad de este producto es resultado de su modelo de funcionamiento, el cual consiste en una arquitectura basada en eventos donde cada proceso *worker* (como se denomina a los que atienden conexiones entrantes) puede atender miles de conexiones entrantes en simultáneo, ya que los workers sólo reaccionan ante eventos, en lugar de bloquearse durante la atención de una conexión entrante hasta que se termina de enviar la respuesta. Esto, en contraste con el modelo bloqueante que Apache posee es una gran ventaja, tanto en la reducción de consumos que provoca como en la eliminación de restricciones fácilmente alcanzables con respecto a la cantidad de conexiones entrantes admisibles.

NGINX nace como un producto orientado a los nuevos patrones arquitectónicos y de diseño de las aplicaciones web modernas, como la empresa detrás del producto, NGINX Inc., lo define al comparar NGINX con Apache[45]. En esa publicación explican que, adicionalmente a poder ser productos complementarios, NGINX está diseñado para los nuevos modelos de aplicaciones haciendo especial hincapié en el modelo de microservicios, y para hacer las de proxy o balanceador de carga en una arquitectura que así lo requiera, soportando la mayor parte de la carga y distribuyéndola entre los *backends* de procesamiento (los servicios, en nuestro caso) para que la atiendan.

3.5.3. Conclusión

A partir del análisis realizado y de nuestra propia experiencia, en la que hemos utilizado ambos servidores, consideramos que NGINX es la mejor opción dadas las expectativas de crecimiento que tenemos para este nodo de la nube de servicios.

Su esquema de alta disponibilidad, con bajo consumo de recursos, poco mantenimiento requerido y su activa comunidad son factores determinantes en nuestra decisión.

[45]Cf. `https://www.nginx.com/blog/nginx-vs-apache-our-view`

3.6. Para documentar

Como desarrolladores conocemos la importancia de una buena documentación, que se encuentre actualizada y pensada en ayudar, que invite a utilizarla y que sea fácil de consultar. En la actualidad, la nube de servicios carece de una documentación que cumpla con estos requisitos, lo cual produce situaciones poco deseables para quienes utilizamos sus servicios, ya que esto, sumado a la falta de consistencia en las respuestas y los parámetros admitidos, hacen que debamos consultar el código fuente con demasiada frecuencia.

En este apartado analizaremos herramientas que proponen estándares de documentación para APIs e intentan asistir a mantener utilizable y al día la información de consulta sobre los servicios que éstas ofrecen.

3.6.1. RAML

Entre las alternativas sobresalientes para la documentación de APIs REST, encontramos en RAML un conjunto extenso de funcionalidades que no se limita únicamente a la generación de una documentación viviente para nuestros servicios, si no que además permite generar código y tests para realizar pruebas a unidad sobre nuestros endpoints mediante el uso de herramientas complementarias diseñadas para esta tecnología.

RAML, cuyo nombre es el acrónimo para *RESTful API Modeling Language*, es una especificación y un lenguaje de descripción de APIs REST basado en YAML, diseñado por un grupo de trabajo integrado por importantes empresas de la industria como Cisco, VMWare, Mulesoft e Intuit. Su premisa es ser una especificación abierta, sin ser dirigida a un proveedor específico, y que su estructura permita describir APIs REST de manera clara, correcta, precisa, consistente, legible, natural e intuitiva.

3.6.1.1 Licencia

RAML se encuentra liberado bajo la licencia Apache versión 2.0[46].

---

[46] `http://raml.org/about/legal`

3.6.1.2 Estructura

Al escribir la descripción de una API REST con RAML, debemos crear un archivo principal con extensión `.raml` que contendrá el nodo raiz de la especificación de los servicios. En ese nodo se define la versión de RAML a utilizar[47], el título que le damos a la API, su dirección base y algunos atributos opcionales como por ejemplo la versión.

Adicionalmente, se definen los recursos que los servicios pueden proveer, identificándolos por sus URIs, y anidándolos en caso que se trate de recursos que tengan esa organización. Bajo estas claves que indican los servicios (o sus URIs, más precisamente) se especifican qué métodos HTTP admite cada URI, junto con un detalle de los parámetros que admita (en caso que aplique). Además de describir los servicios y la forma de realizar las peticiones, RAML permite describir las respuestas mediante el uso de ejemplos o esquemas genéricos, agrupándolos por código de estado HTTP.

Al ver la forma en que se estructuran las especificaciones RAML, rápidamente notamos que es un formato orientado a la reutilización de recursos y definiciones, buscando en todo momento evitar la repetición innecesaria de información, lo cual es una consideración positiva a la hora de planificar una especificación donde es altamente probable que tanto datos como esquemas se repitan en distintos puntos. En el mismo sentido, permite incluir en cualquier punto de un archivo `.raml` el contenido de otro archivo para utilizar su contenido, sin que necesariamente se trate de otro archivo con el mismo formato. Esto también asiste a la simplificación del trabajo de documentación y la reutilización de los recursos definidos a lo largo de la especificación que se realiza.

3.6.1.3 Herramientas

El ecosistema de herramientas alrededor de RAML es variado, con implementaciones de su *Parser* en los lenguajes más populares, y con algunas librerías (principalmente implementadas en JavaScript mediante Node.js) para servir la documentación viviente generada a partir de los documentos RAML, y con otras para generar tests automatizados de las APIs REST descriptas mediante este lenguaje. Este último punto es una gran ventaja, consideran-

[47] Al momento de escritura del presente informe, la versión más reciente de RAML es la `1.0`

do la importancia que tienen las pruebas automatizadas sobre los servicios críticos que provee la nube de servicios de la UNLP.

3.6.2. *The OpenAPI Specification*

Esta especificación, que hasta enero de 2016 se conocía como *Swagger*, fue donada por la empresa *SmartBear* a la *Open API Initiave* (OAI) para fomentar su crecimiento en manos de esta última. La iniciativa OAI es un consorcio de empresas líderes en la industria[48] que forma parte de los proyectos colaborativos de la *Linux Foundation* y como tal mantiene un modelo de gobierno abierto entre sus miembros.

La OpenAPI-Spec apunta a ser un estándar, agnóstico tanto tecnológica como corporativamente, para describir las APIs REST que interconectan las aplicaciones relacionadas a la web moderna. Al momento de realización del presente trabajo de análisis, esta especificación se encuentra en su versión `2.0`, con cierto trabajo de preparación de la tercera versión en proceso por parte de la OAI.

3.6.2.1 Licencia

OpenAPI-Spec está liberado bajo la licencia Apache versión 2.0[49].

3.6.2.2 Organización

La especificación define una organización en archivos que describan la API que documentan utilizando el formato JSON o YAML, según sea preferencia de quien esté utilizando esta herramienta. Entre esos archivos se cuenta con uno principal y requerido que es `swagger.json` (o `swagger.yml`), a partir del cual se comienza la lectura de la información que describe la API.

La descripción de una API comienza con un `schema`, que es un objeto JSON que contiene información general como datos sobre el proveedor de los servicios, posibles `schemes` que utiliza para servir los recursos o tipos MIME utilizados, y luego información más técnica como descripciones detalladas

[48] Entre otros miembros destacados, podemos mencionar a Google, IBM, Microsoft, PayPal, apigee y Mashape - `https://openapis.org`

[49] `https://github.com/OAI/OpenAPI-Specification/blob/master/LICENSE`

de los endpoints que se exponen, parámetros y tipos de datos admitidos, protocolos de seguridad y metadatos como categorías para que al generar la documentación puedan agruparse lógicamente los servicios a partir de éstas.

A partir de estos elementos, se pueden generar descripciones altamente detalladas de una API y los servicios que ésta provea, lo cual podría alimentar tanto a clientes automatizados que respeten el estándar de documentación, como a generadores automáticos de documentación que la dejen disponible para ser utilizada por desarrolladores o interesados en utilizar estos servicios.

3.6.2.3 Herramientas

Las herramientas disponibles acorde a este estándar están orientadas a tecnologías como Node.js o Java, lo cual deja nuestra elección de lenguaje para el desarrollo de las APIs (Ruby) parcialmente marginada. Afortunadamente existen librerías que nos permiten integrar la documentación en nuestro mismo código Ruby mediante directivas de un DSL dedicado y a partir de esta información generan el archivo `swagger.json`. Es interesante destacar en este punto que si bien la especificación fue donada a la OAI y renombrada a OpenAPI-Spec, el conjunto de herramientas que provee siguen siendo las relacionadas a Swagger.

A partir de ese paso inicial, se puede utilizar la herramienta *Swagger UI* para generar sitios web con la documentación viviente.

3.6.3. Conclusión

En esta sección hemos analizado herramientas para documentar APIs REST que, en algunos casos, sirven para otros propósitos derivados del hecho de simplemente considerar la documentación de éstas como un texto descriptivo. Al tomar la documentación (o especificación) de un conjunto de servicios como un *contrato*, ésta recibe una entidad mayor y gana importancia en el proceso de implementación de una API, ya que a partir de lo que en la documentación se defina se pueden generar tests automatizados que aseguren el correcto funcionamiento de los servicios, y a su vez, generen la necesidad de mantener actualizada esa especificación a medida que los servicios evolucionan.

Si bien las opciones consideradas comparten muchas bondades, hemos encontrado en *The OpenAPI Specification* el modelo que más se ajusta a nuestras necesidades, y aquel que más se condice con nuestros principios de

trabajo, al ser un estándar abierto impulsado por la Linux Foundation y mantenido por un consorcio compuesto de las más importantes empresas del sector.

4. Capítulo IV: Propuesta de rediseño

Habiendo presentado el marco teórico de nuestro análisis en el capítulo II y a partir de las herramientas comparadas en el capítulo III, en este apartado nos centraremos en describir el diseño que proponemos para la nueva arquitectura de la nube de servicios de la Universidad Nacional de La Plata. Abordaremos la misma desde distintos aspectos que consideramos clave para su reimplementación futura, brindando una visión más orientada a las necesidades que vemos en el estado actual de la nube.

De aquí en más, para distinguir la nube actual (el Integrador) de la nueva arquitectura que estamos proponiendo, utilizaremos el nombre clave *Cloud* para hacer referencia a esta última.

4.1. Propuesta

Por lo analizado en capítulos anteriores, se propone una arquitectura más desacoplada a la planteada con el Integrador, permitiendo de esta manera minimizar el costo del mantenimiento, desarrollo y simplificando su despliegue en entornos de producción. Una solución basada en estándares que permita integrar sistemas heterogéneos, aceptando el hecho de que la mayoría de los sistemas legados que se encuentran en producción, se mantendrán y logrando de esta manera, que la infraestructura subyacente facilite la incorporación de cambios que puedan surgir como necesidades del CeSPI y la Universidad Nacional de La Plata.

El modelo de servicios facilita el acceso y consumo de la información a través de la red. Dado que los servicios son independientes y autónomos, pueden combinarse tantas veces como sea necesario de manera sencilla, generando nuevas aplicaciones que respondan a las necesidades en constante evolución de nuestra casa de estudios. Esta posibilidad de agregar y combinar servicios para resolver situaciones presentes y futuras, convierten en una opción altamente beneficiosa el uso de esta estrategia, con el fin de crear servicios y aplicaciones complejas con independencia de las tecnologías subyacentes[2].

El resultado final es una nube, con un conjunto de servicios y una creciente flota de aplicaciones dependientes de éstos, que se adapta fácilmente a los cambios.

A continuación profundizaremos la aplicación concreta de los conceptos vistos hasta aquí, explicando nuestra propuesta a través de los puntos mencionados en el objetivo al comienzo del presente trabajo.

4.1.1. Redundancia y escalabilidad

Cuando hablamos de escalabilidad, podemos basarnos en el modelo llamado *scale cube*[1] presentado visualmente en la Figura 16. Este modelo clasifica las distintas formas de escalar las aplicaciones en 3 sentidos, tomando como analogía las 3 dimensiones de un cubo:

- **Escalar sobre el eje *X*:** (*X-axis scaling*, en inglés) Técnica comúnmente utilizada para incrementar la disponibilidad de las aplicaciones. Consiste en el uso de instancias replicadas de la aplicación (idénticas a la original), ubicadas detrás de un balanceador de carga que reparte el trabajo entre ellas.
- **Escalar sobre el eje *Y*:** (*Y-axis scaling*, en inglés) Ésta es una técnica que se encuentra en auge con el *momentum* que están teniendo los microservicios. Se basa en la descomposición funcional de la aplicación en un conjunto de servicios colaborativos, donde cada uno implementa una parte específica de las funciones. En este caso, la disponibilidad se incrementa al separar en componentes más pequeñas la unidad funcional mayor que es la aplicación, dividiendo el trabajo independientemente entre ellas.
- **Escalar sobre el eje *Z*:** (*Z-axis scaling*, en inglés) En este enfoque se toma como base la replicación de instancias presente en *X-axis scaling* y a esto se agrega una capa superior de abstracción que separa el conjunto de datos que cada instancia o instancias atenderá, acorde a algún criterio lógico como puede ser el cliente que realice la petición (privilegiando unos clientes sobre otros).

Como vimos en el Capítulo I, el diseño de la arquitectura del Integrador presenta falencias que no permiten implementar redundancia ni escalar de manera eficiente, ya que para poder hacerlo, encontramos al menos dos inconvenientes: para lograr redundancia, sería necesario replicar su instancia virtual y al mismo tiempo implementar alguna solución que redirija las peticiones a cualquiera de sus réplicas. Además, debido a su diseño arquitectónico, sería necesario replicar cada API, pero como se mencionó anteriormente, éstas se encuentran fuertemente acopladas a sus aplicaciones, lo que fuerza a replicar no sólo la API, sino también la aplicación que genera los datos.

Debido a esto, planteamos una nueva arquitectura de la nube de servicios de la Universidad Nacional de La Plata que debe ser replicable y escalable. Para lograrlo, cada API correrá en una instancia virtual independiente, de

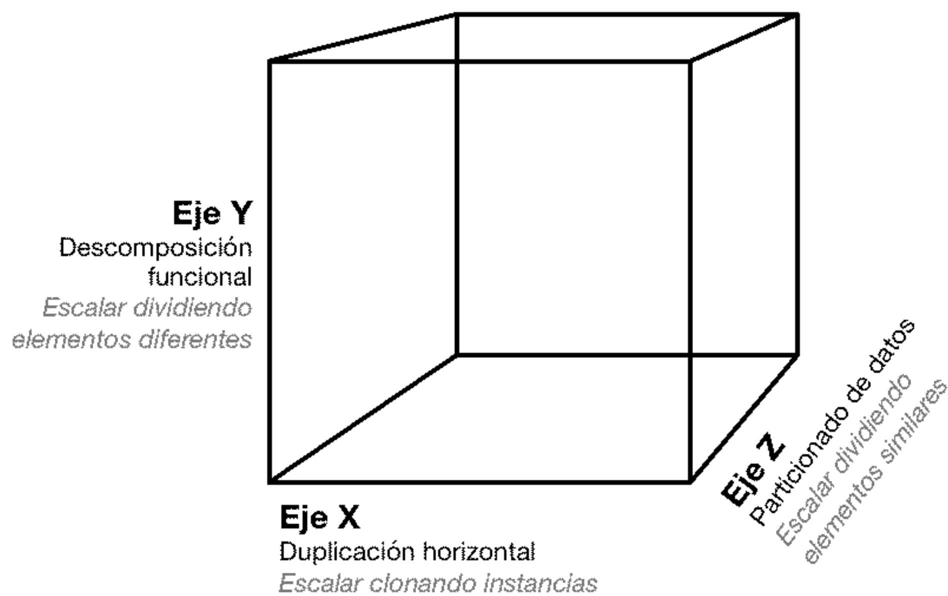


Figura 16: El modelo de escalabilidad *scale cube*

manera tal que la misma pueda ser replicada tantas veces como sea necesario (*X-axis scaling*). La capa de abstracción de estas intancias replicadas será un balanceador de carga, implementado con NGINX, que servirá tanto para balancear la carga de *failover*, dando continuidad a los servicios en caso de que alguna de las instancias replicadas falle.

La característica fundamental de un balanceador de carga es ser capaz de distribuir las peticiones entrantes a un grupo de servidores (*backends*) de acuerdo a un algoritmo de decisión y ponderación llamado *scheduler*. Este algoritmo seleccionará qué backend podría atender un requerimiento entrante, existiendo algunos simples (hacerlo de manera aleatoria o siguiendo un criterio ordenado por turnos como *Round Robin*) y otros más sofisticados (que consideran otros factores como por ejemplo la carga de cada backend, su tiempo de respuesta promedio, el número de conexiones activas, la ubicación geográfica, etc.). En su funcionamiento básico el balanceador de carga redirige las peticiones a alguno de los backends, el cual luego contesta al balanceador, para que finalmente sea el balanceador el que entregue la respuesta al cliente sin que este último sepa de la existencia de esta compleja arquitectura. Esta estructura transparente al cliente evita accesos directos entre éste y los servidores que actúan como backend.

Como se mencionó anteriormente, el balanceador puede también ser usado como *failover*, permitiendo que la falla de uno o más backends no afecten

la disponibilidad del servicio. Los backends son monitoreados continuamente por el balanceador, cuando uno de éstos falla, el balanceador deja de enviar tráfico al backend caído. Una vez que el backend vuelve a estar online, el balanceador detecta esta situación y comienza a enviarle tráfico nuevamente.

Para disminuir la cantidad de accesos que llegan a las instancias replicadas de los servicios, delante del balanceador se implementará una caché compartida utilizando Varnish, la cual evitará los accesos innecesarios al maneter una copia en memoria de las respuestas que alguna de las instancias replicadas de la Cloud haya generado, obteniendo mejores tiempos de respuesta así como tolerancia a fallos y reducción de carga en estas últimas.

A modo de ejemplo, podemos pensar en una aplicación (*cliente A*) que accede al servicio `/academic_units`. La petición es atendida por la API de referencia, donde el servicio devuelve todas la Unidades Académicas activas. Si más tarde otra aplicación (*cliente B*), consulta la API por las Unidades Académicas, accediendo también al servicio `/academic_units` de la misma API, terminará obteniendo el mismo listado de Unidades Académicas. Como se puede apreciar, tenemos dos accesos a la API de referencia con idénticas respuestas, ambas procesadas por completo por alguno de los backends. Esta situación puede evitarse con la inclusión de una cache compartida delante del balanceador de carga, logrando mejorar los tiempos de respuesta ya que la API recibirá únicamente un acceso, debiendo acceder y procesar los datos una única vez, ya que el resto de las peticiones serán obtenidas desde la cache compartida.

4.1.2. Desacoplamiento

Como se mencionó en la Subsección 2.4 (ESB), uno de los 9 principios de diseño de SOA es el bajo acoplamiento (*loose coupling*), y quizás la forma más común de implementarlo es mediante un ESB, el cual se encargará de proveer interoperabilidad entre las diferentes plataformas. En nuestra arquitectura proponemos utilizar Tyk como ESB delante del Varnish para agregar mecanismos de autenticación, *rate limiting*, monitoreo y un único punto de acceso a toda la Cloud, el cual estará restringido a las direcciones IP de las aplicaciones cliente. La inclusión del ESB evita la necesidad de desarrollar en las APIs funcionalidad extra, que este mismo provee.

Al mismo tiempo, en pos de trabajar en el desacoplamiento de las plataformas, se propone desarrollar una API dedicada a servir datos de referencia en la cual se implementarán los servicios necesarios que permitirán el acceso a esta información desde las distintas aplicaciones. Esta API comprenderá

los *endpoints* del Integrador que se encuentran en uso.

Adicionalmente, se deberán desacoplar de sus respectivas aplicaciones y reescribir todas las APIs que se encuentran actualmente en producción. Como hemos mencionado en la Subsección 3.1, se utilizará Ruby on Rails para desarrollar estas aplicaciones. Siguiendo el patrón de microservicios, este proceso dará origen a diferentes *service components*, que antes se encontraban implementados dentro de cada aplicación y acopladas a la misma, y ahora estarán implementados en una nueva aplicación independiente y dedicada. De esta manera desacoplamos la lógica de la aplicación, del acceso a los datos que se generan en la misma, permitiendo que esta solución escale horizontalmente de manera sencilla.

Para la solución planteada podemos observar las siguientes ventajas:

- **Escalabilidad:** permite escalar horizontalmente (*X-axis scaling*), es decir, en el hipotético caso que una de las APIs sea accedida por muchas aplicaciones, la misma podría replicarse en varias instancias, evitando la sobrecarga de cualquiera de ellas, al mismo tiempo que serviría como *failover*.
- **Administración centralizada:** se centraliza la administración de los datos de referencia, evitando que la duplicidad se propague en cada aplicación que necesite utilizarlos.
- **Mantenimiento centralizado:** cuando hablamos de mantenimiento nos referimos a mejoras en la API, como por ejemplo, implementación de nuevos servicios, arreglo de errores, etc. Tener una APIs totalmente desacopladas de las aplicaciones que generan los datos, permite actualizarlas de manera sencilla evitando parar momentáneamente otras aplicaciones.

También se debe tener en cuenta que esta solución permite independizarse del lenguaje utilizado para el desarrollo de la aplicación: podemos desarrollar las aplicaciones en Ruby, PHP, JavaScript, Java o cualquier otro lenguaje, distinto al que utilicemos para desarrollar las APIs. Esto genera una capa independiente de servicios, donde las aplicaciones delegarán en las APIs el acceso, la gestión y persistencia de los datos que ellas mismas generan.

Para ejemplificar la situación anterior, podemos pensar en la aplicación de registro de usuarios. En la actualidad, la aplicación accede a una base de datos compartida con la aplicación de administración de usuarios, lo cual resulta en una duplicación de lógica en ambas. Con el esquema que proponemos, se

creará una API de usuarios que provea la lógica común a ambas (como la creación de un nuevo usuario o su actualización) y se quitará la lógica afin de cada una de las aplicaciones existentes para transformarlas en clientes de esta nueva API. De esa forma, para crear un nuevo usuario desde la aplicación de registro o la de administración, obtendrá la información necesaria y luego la enviará al servicio de creación de usuarios de la nueva API, la cual será la responsable de persistir los datos recibidos.

4.1.3. Simplicidad

Como se desarrolló en la Subsección 2.3, las aplicaciones monolíticas consisten en componentes fuertemente acoplados, que son parte de una única unidad desplegable, resultando dificultosa la incorporación de cambios, *testing* y despliegue sin caídas del servicio. El patrón de microservicios trata estas cuestiones, separando la aplicación en múltiples unidades desplegables, que pueden ser desarrolladas, testeadas y desplegadas de forma independiente.

El hecho de dividir la aplicación en componentes pequeños y desplegables de manera independiente, facilita el desarrollo, *testing* y puesta en producción, esto se debe a que el cambio se encuentra aislado a un *service component*, permitiendo un mayor control. Es por esto que hemos optado por seguir el patrón de arquitectura de microservicios para el diseño de la nube de servicios de la Universidad Nacional de La Plata.

4.1.4. Tolerancia a fallos

Los sistemas distribuidos tienen potencialmente más fallos que los sistemas monolíticos, ya que en cada solicitud intervienen decenas (o cientos) de microservicios diferentes[10, p. 48]. Es por esto que no resulta suficiente descomponer un sistema en unidades independientes; también hay que evitar que un fallo en uno de éstas cause un fallo en cascada[10, p. 4], situación en que un error en una capa interna provoca un fallo en la capa más externa[13, p. 65]. Mike Nygard describe en su libro “Release It!”[13] varios patrones tolerantes a fallos, de los cuales el más popular es el *Circuit Breaker*, presentado en la Figura 17.

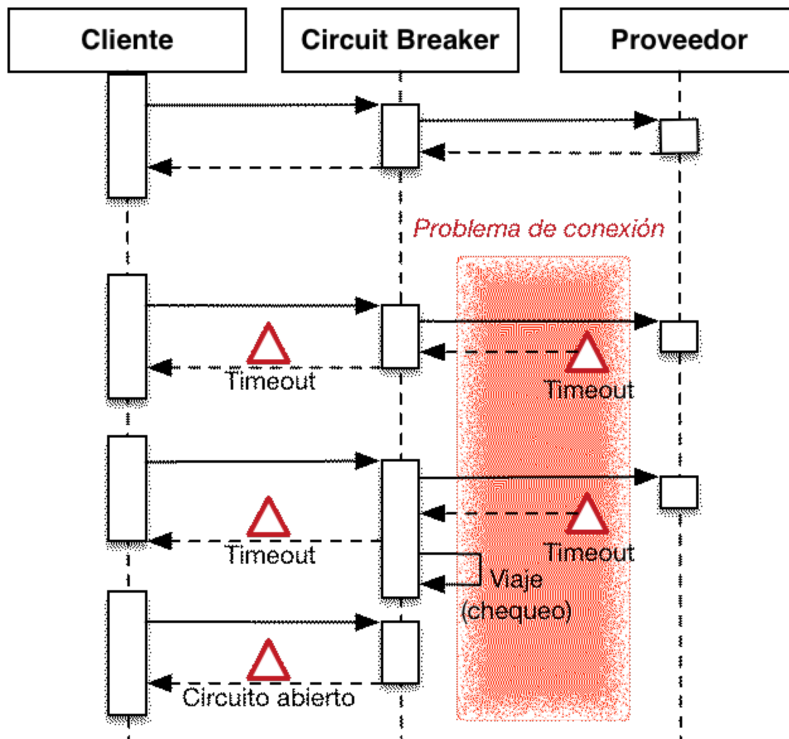


Figura 17: Patrón de tolerancia a fallos *Circuit Breaker*

Para implementar una infraestructura tolerante a fallos, los servicios deberán estar replicados en varias instancias virtuales y, como detallamos anteriomente, delante de éstas se implementará un balanceador de carga que permitirá distribuir las peticiones recibidas entre las instancias virtuales. Al mismo tiempo, se deberá trabajar en limitar el alcance del fallo a nivel de servicio. Cabe aclarar que el partón de microservicios, realiza un aporte importante en este orden, ya que al dividir la aplicación en *service components* aislamos el problema, facilitando luego su eventual solución y despliegues en producción.

4.1.5. Estandarización

Según el diccionario de la Real Academia Española, un estándar es lo que sirve como tipo, modelo, norma, patrón o referencia. En nuestro caso, la estandarización es el proceso por el cual se establecen normas comunmente aceptadas que permiten la comunicación de diferentes aplicaciones.

Como mencionamos anteriormente, para cada aplicación que lo requie-

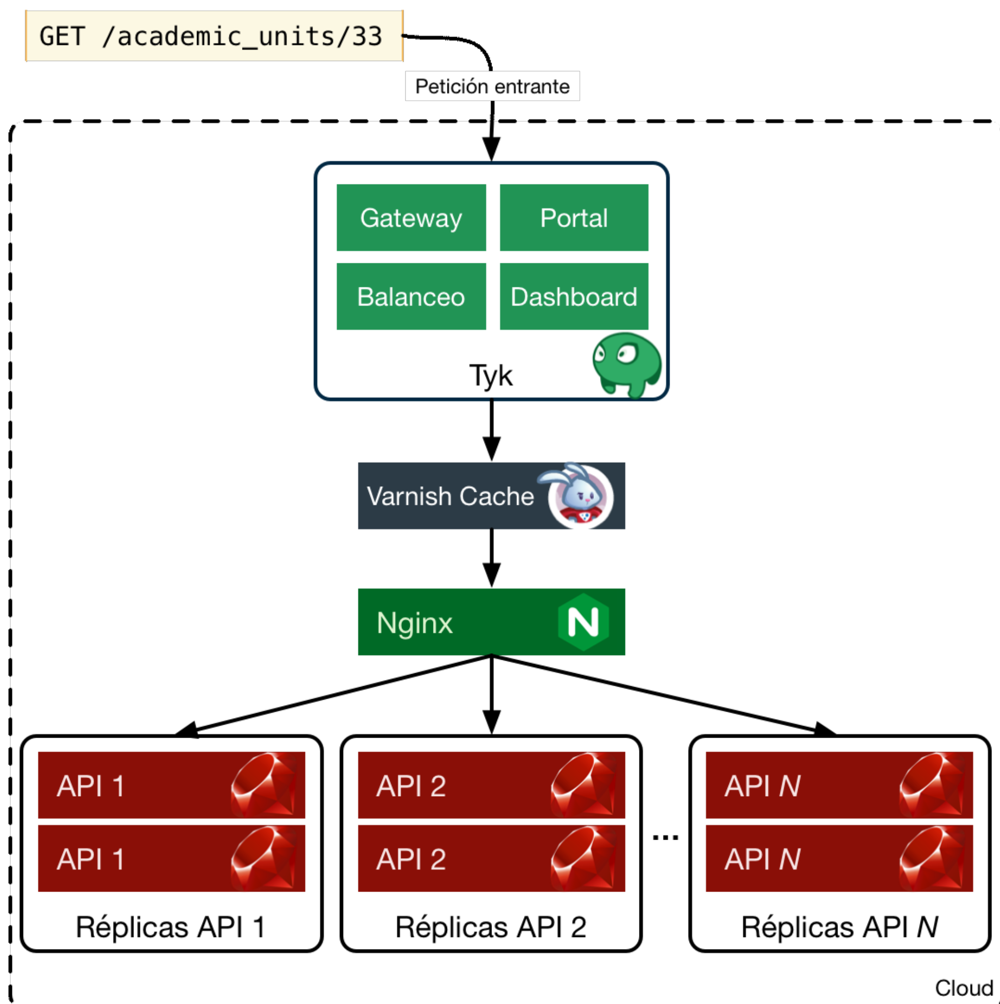


Figura 18: Arquitectura propuesta para la Cloud de la UNLP

ra se deberá implementar una API de acceso a los datos que ésta genere. Esta implementación se realizará basándose en la especificación JSON API detallada en la Subsección 3.2.

Esta estandarización facilitará el desarrollo de clientes que consuman información de los diferentes servicios de las APIs, ya que definen reglas que especifican cómo se podrá acceder a los datos, cuál será su estructura de respuesta, e incluso asisten en lograr independencia del lenguaje con el que se escriban estos clientes, quienes simplemente deben respetar las especificaciones para implementar el acceso a los servicios.

En la Figura 18 se puede observar la arquitectura final propuesta.

5. Capítulo V: Caso testigo

En este último capítulo presentaremos el desarrollo realizado como parte del presente trabajo, en el cual llevamos al plano práctico los conceptos que hemos analizado y explicado en los capítulos anteriores.

Tomando un caso testigo acotado documentaremos la experiencia de implementarlo siguiendo los principios que antes definimos, las cuales nos servirán de fundamento para el siguiente capítulo, en el cual enunciaremos las conclusiones obtenidas en el presente trabajo.

5.1. Caso testigo

Habiendo desarrollado la propuesta de rediseño que motiva el presente trabajo, creemos conveniente tomar un caso testigo que sirva de ejemplo práctico para hacer concretos los conceptos teóricos que hemos tenido en cuenta para el análisis, a la vez que sirva de disparador para una implementación preliminar acotada del nuevo diseño sobre los elementos de la nube de servicios.

5.1.1. Alcance

La situación que utilizaremos como caso testigo, es el proceso de registro de un nuevo usuario de *Single Sign-On* (SSO) de la nube de servicios de la Universidad Nacional de La Plata. Si bien *a priori* puede parecer un tanto sencillo para tomarlo de referencia, una vez que definamos en concreto todos los pasos involucrados en este proceso se hará evidente la elección realizada.

Aquellos agentes que forman parte del personal de la UNLP pueden tener un usuario único para acceder a las aplicaciones integradas en el SSO, entre ellas para consultar sus recibos de haberes, utilizar una aplicación como parte de sus tareas diarias, presentar proyectos de extensión cuando la convocatoria se encuentre abierta o para acceder a cualquier aplicación que a futuro pudiéramos publicar. Para registrar su usuario, el agente dispone de dos vías:

- La vía analógica: solicitando la creación del usuario, de manera presencial y por escrito, a la oficina de Personal de alguna de las Dependencias donde desempeña sus funciones. Por tratarse de un proceso manual y *offline*, no lo consideraremos para este caso testigo.

- La vía digital: utilizando la aplicación de autogestión que hemos desarrollado para realizar el registro en línea del nuevo usuario. *Ésta* es la vía que utilizaremos en nuestro planteo.

En el proceso de autogestión de un nuevo usuario, el agente debe completar los datos solicitados en una serie de pasos preestablecidos que lo guían hasta llegar a obtener el acceso a las aplicaciones que utilizan este esquema de SSO[50]. Podemos resumir los pasos para el registro en:

1. El agente ingresa a la aplicación de autogestión e indica que desea registrar un nuevo usuario. Para esto, debe ingresar una dirección de correo institucional propia para recibir por esa vía un vínculo de acceso para comenzar efectivamente el registro de su nuevo usuario.

2. Al ingresar al vínculo de acceso que le llega a su correo, la aplicación solicita al agente que se identifique mediante su tipo y número de documento de identidad y en respuesta a esto le indica si se encuentra entre los agentes de la Universidad que aún no tienen usuario de acceso único.

3. Una vez identificado el agente, se le sugiere un nombre de usuario siguiendo la política de nombres de usuario, el cual puede ser modificado como parte de la carga de información que se está realizando. Se valida que este usuario esté disponible y que respete dicha política.

4. Luego de elegido el nombre de usuario, se permite al agente ingresar una dirección de correo electrónico alternativa para tener un segundo medio de contacto.

5. Para cerrar la carga de datos personales, se le pide que indique en qué Dependencias de la UNLP presta servicios. Esto se realiza por medio de un captcha para evitar *bots* que intenten registrar usuarios en masa y personas malintencionadas que deseen registrar usuarios en nombre de agentes reales.

---

[50]En realidad, inicialmente obtiene acceso únicamente a ver sus recibos de haberes y cargar su *currículum de extensionista* en la aplicación de Proyectos de Extensión. Para ingresar al resto de las aplicaciones un usuario autorizado le debe asignar los roles necesarios para cada aplicación. Para más información sobre las aplicaciones integradas al SSO y la nube de servicios, referirse al Apéndice A.

6. Al validar correctamente esta información, se le presentan todos los datos ingresados para su confirmación y si el agente los acepta, se le envía un correo electrónico con un formulario para presentar firmado en la oficina de Personal de alguna de las Dependencias donde presta servicios, teniendo un paso de validación humana de la información[51].

7. Una vez presentado el formulario y debidamente corroborados los datos por algún empleado de la oficina de Personal correspondiente, el usuario es aprobado, evento que dispara el envío de un correo al agente, en el cual se le consignan su usuario y una clave provisoria para que ingrese por primera vez a los sistemas de la UNLP, y en ese momento la cambie por una clave de su propia elección. Con este paso, se finaliza el proceso de registro del nuevo usuario para el agente.

Desde el punto de vista de nuestra aplicación, este proceso se ve un tanto reducido ya que no es necesario que incluyamos los pasos que ocurren por fuera del *mundo virtual* o en la aplicación de gestión que utilizan los empleados de las oficinas de Personal de las Dependencias, ya que quedan fuera de su alcance. Formalizaremos los pasos que se realizan en este sistema con el diagrama de flujo mostrado en la Figura 19 para su mejor comprensión.

[51]Este paso adicional, si bien puede sonar contradictorio con el resto del proceso *online* planteado, fue requerido por las autoridades de la Universidad para fortalecer los chequeos realizados sobre los datos recibidos mediante este servicio público de registro.

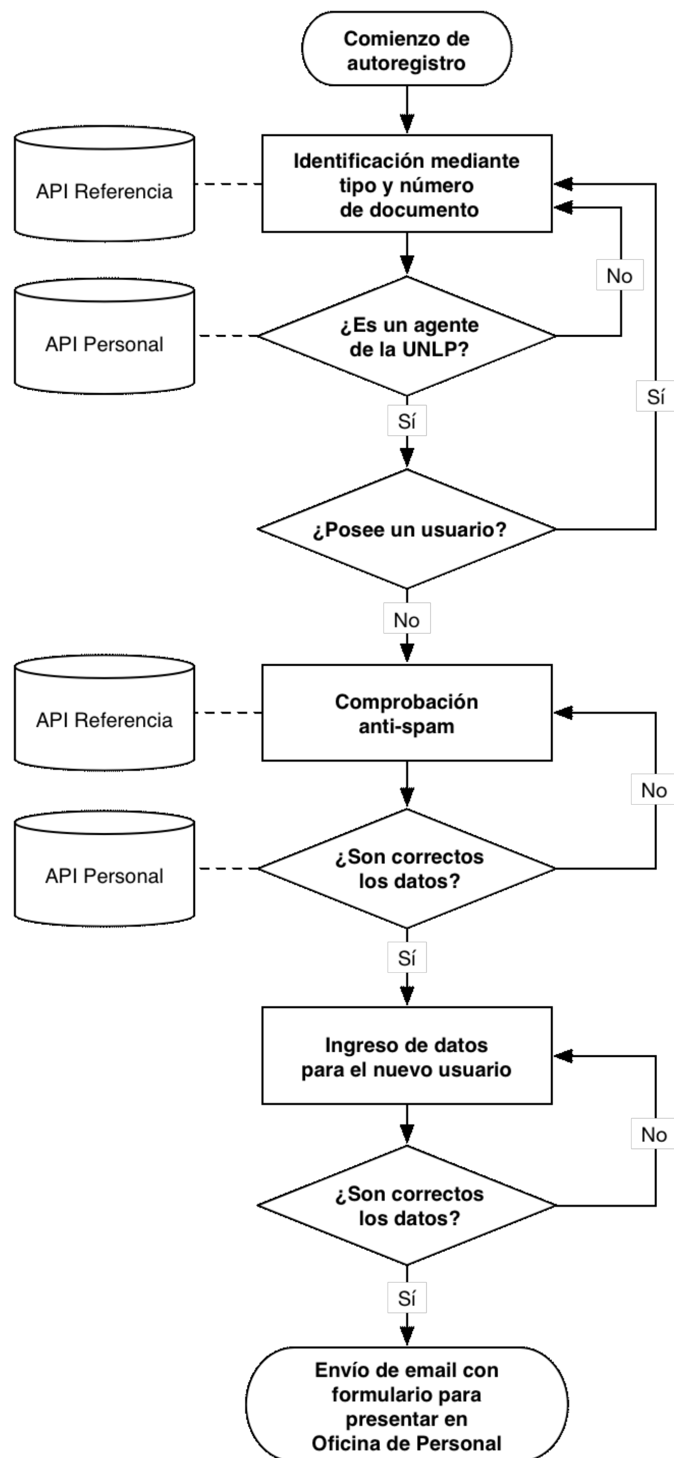


Figura 19: Proceso de autoregistro de un nuevo usuario de acceso único

Realizando una descomposición en servicios del proceso anterior, identificamos las siguientes dependencias con servicios de la Cloud:

- **Servicios de referencia:** para visualizar y marcar las Dependencias (o Unidades Académicas) y listados de tipos de documento de identidad.
- **Servicios de información sobre el personal:** en la identificación de la persona y posterior consulta de las Unidades Académicas en las cuales presta sus servicios.
- **Servicios de usuarios:** para la consulta de la existencia de un usuario de la persona seleccionada, sugerir nombres de usuario que respeten la política de nombres definida y en la creación del nuevo usuario.
- **Servicio de notificaciones:** en el envío de correos electrónicos. Destacamos en este punto que para limitar el alcance del caso testigo dejaremos la implementación de este servicio como un trabajo a futuro, ya que su existencia no es limitante para poder desarrollar los pasos de registro que se realizan en línea.

El desarrollo del prototipo funcional implica:

- Implementar desde cero los servicios de la Cloud antes mencionados (de referencia, de información sobre el personal y de usuarios), en los cuales incluiremos únicamente los *endpoints* necesarios para solventar la lógica del caso testigo.
- Implementar desde cero una nueva versión de la aplicación cliente de registro, respetando los pasos antes descriptos, que consuma toda la información de los servicios de la Cloud. Esta nueva versión difiere en su concepción de la versión actualmente implementada, en que no tendrá acceso a la base de datos de usuarios. De hecho, no tendrá acceso a *ninguna* base de datos, ya que realizará todas sus operaciones de manera volátil dependiendo completamente de los servicios de la Cloud para acceder a la capa de persistencia en base de datos.
- Desarrollar una *gema* (nombre que reciben las librerías reutilizables en el lenguaje Ruby) que encapsule la lógica de acceso a los servicios (cliente), desde la autenticación hasta la consulta y abstracción en objetos de las respuestas de sus APIs, basándose en el estándar elegido para la codificación de la información.

Hemos elegido este caso testigo principalmente porque se compone de una serie de interacciones entre distintos servicios que brindan una noción de la complejidad que pueden tener las operaciones a realizar utilizando la nube de servicios, más allá de las simples consultas de datos de referencia que habitualmente manejamos. El hecho de incorporar diferentes servicios, englobados en distintas áreas de acción, nos permite también mostrar cómo funcionan las instancias de las aplicaciones que proveen esos servicios, cómo interactúan con la aplicación cliente y con los elementos intermediarios de la comunicación (entiéndase *caches* compartidas, balanceadores de carga, *proxies* reversos y la capa de mediación ofrecida por Tyk).

En este informe intentaremos no entrar en detalles innecesarios sobre la implementación de la aplicación cliente ni de la capa de lógica del negocio de los servicios, para sí ahondar sobre temas relevantes desde el punto de vista de la comprensión de las entidades participantes en las comunicaciones. Para su referencia, se adjunta al presente informe el código fuente de los distintos apartados que hemos implementado en este capítulo.

5.1.2. Arquitectura

Partiendo de la Subsección 4.1, el primer paso dentro de la implementación del prototipo funcional para este caso testigo es acotar la arquitectura, diseñarla y preparar los nodos en ella involucrados. Para esto, pueden observarse en la Figura 20 los componentes lógicos del prototipo:

- **Aplicación de registro:** ésta es la implementación de una aplicación cliente de la Cloud. Es la cara visible hacia los usuarios públicos, y si bien en este caso particular hemos decidido implementarla utilizando Ruby on Rails por cuestiones prácticas, podría implementarse utilizando otras tecnologías como *frameworks* web JavaScript que se ejecuten meramente del lado del cliente, PHP o Java, por nombrar algunas. De todos los componentes de la arquitectura, éste es el único que se encuentra lógicamente por fuera de la Cloud. Esta separación del resto la hacemos evidente porque en este caso se trata de una aplicación desarrollada por nosotros mismos, pero en el futuro bien podría tratarse de una aplicación desarrollada por terceros, pertenecientes o no a la Universidad Nacional de La Plata.

- **API Gateway (ESB):** este nodo se encarga de ser la capa que aisla los clientes de la API, brindando el acceso a los mismos desde un único punto. Como hemos visto con anterioridad, es el encargado de autorizar

las peticiones entrantes y de chequear que se encuentren dentro de los límites para ellas establecidos, puede proveer una capa de caching intermedio, y también balancear la carga de los backends de servicios con un mecanismo de Round Robin que asigna una petición entrante *no cacheada* a cada backend por vez. Complementariamente a estas tareas, el producto elegido en nuestro análisis para este rol (Tyk) provee una interfaz web de gestión para los usuarios internos que administran el acceso a los servicios (en principio, nosotros) con gráficos estadísticos del uso de nuestras APIs que resulta de gran utilidad.

- **Cache de Gateway:** para obtener mejor rendimiento general de la Cloud, implementamos una Cache de Gateway ubicada por detrás del API Gateway. Esta capa adicional de caching nos permite evitar volver a procesar en los *service components* peticiones que estén frescas en esa cache, incrementando la performance general de los servicios. Se encuentra implementada con Varnish, cache de alto rendimiento antes analizada y que hemos elegido para este rol.

- **Balanceador de carga:** para brindar la versatilidad de agregar o quitar tantas instancias de los *service components* como se desee, incluimos un balanceador de carga antes de éstos para distribuir los requerimientos entrantes. Este se encuentra implementado con un NGINX que realiza la delegación de las peticiones mediante un Round Robin.

- **API de referencia:** esta, como el resto de las APIs implementadas, provee un conjunto de servicios relacionados a una única incumbencia lógica, siguiendo el principio de una sola responsabilidad[52] e implementando el patrón de microservicios tratado con anterioridad. En el caso de este *service component*, su incumbencia es brindar los servicios de consulta de datos de referencia acotados a los necesarios para el alcance de este caso testigo: tipos de documento y unidades acdémicas. Se encuentra implementada con el framework Ruby on Rails en su versión 5 en modo `--api`.

- **API de personal:** este nodo es otro *service component* que se encarga de brindar los servicios relacionados a la información de personal de la

[52] También conocido como *Single Responsibility Principle* en inglés. Este es uno de los cinco principios SOLID de diseño de software que apuntan a mejorar la calidad y facilidad del mantenimiento de un desarrollo.

Universidad Nacional de La Plata. Esta es otra aplicación Rails versión 5 en modo `--api`.

- **API de usuarios:** otro *service component* orientado a brindar acceso a las operaciones a realizar sobre los usuarios del esquema SSO: gestión de usuarios y políticas de nombres de usuario. Al igual que en las APIs anteriores, es una aplicación Rails 5 en modo `--api`.

- **API de notificaciones:** nodo que se dedica a la emisión de notificaciones (principalmente vía correo electrónico) que las aplicaciones cliente pudieran necesitar enviar. Es el *service component* que en una aplicación monolítica sería reemplazado por el *mailer* y el motor de generación de correos electrónicos. Similarmente a los casos anteriores, ésta es otra aplicación Rails 5 en modo `--api`.

- **Capa de persistencia:** esta unidad de la arquitectura planteada es la encargada estrictamente del almacenamiento persistente de la información. Se trata de una base de datos MySQL, que puede estar replicada utilizando un modelo maestro-esclavo para conseguir redundancia y escalabilidad en el eje X.

- **Cache compartida:** este componente de la arquitectura, de carácter opcional, beneficia en términos de escalabilidad y reducción de costos de procesamiento. Está implementado con un servidor de la base de datos clave-valor Memcached, que se utiliza como cache compartida entre las diferentes instancias de los *service components* antes mencionados. Su principal función es evitar volver a calcular las respuestas que cualquiera de los servicios debiera brindar, si antes ya se la generó (ya sea en la misma instancia del servicio u otra distinta) y los datos originales no han cambiado.

Como demostración de las posibilidades de escalabilidad de esta arquitectura, hemos replicado los nodos de servicios para correr dos instancias de cada uno, las cuales serán balanceadas de manera transparente por NGINX.

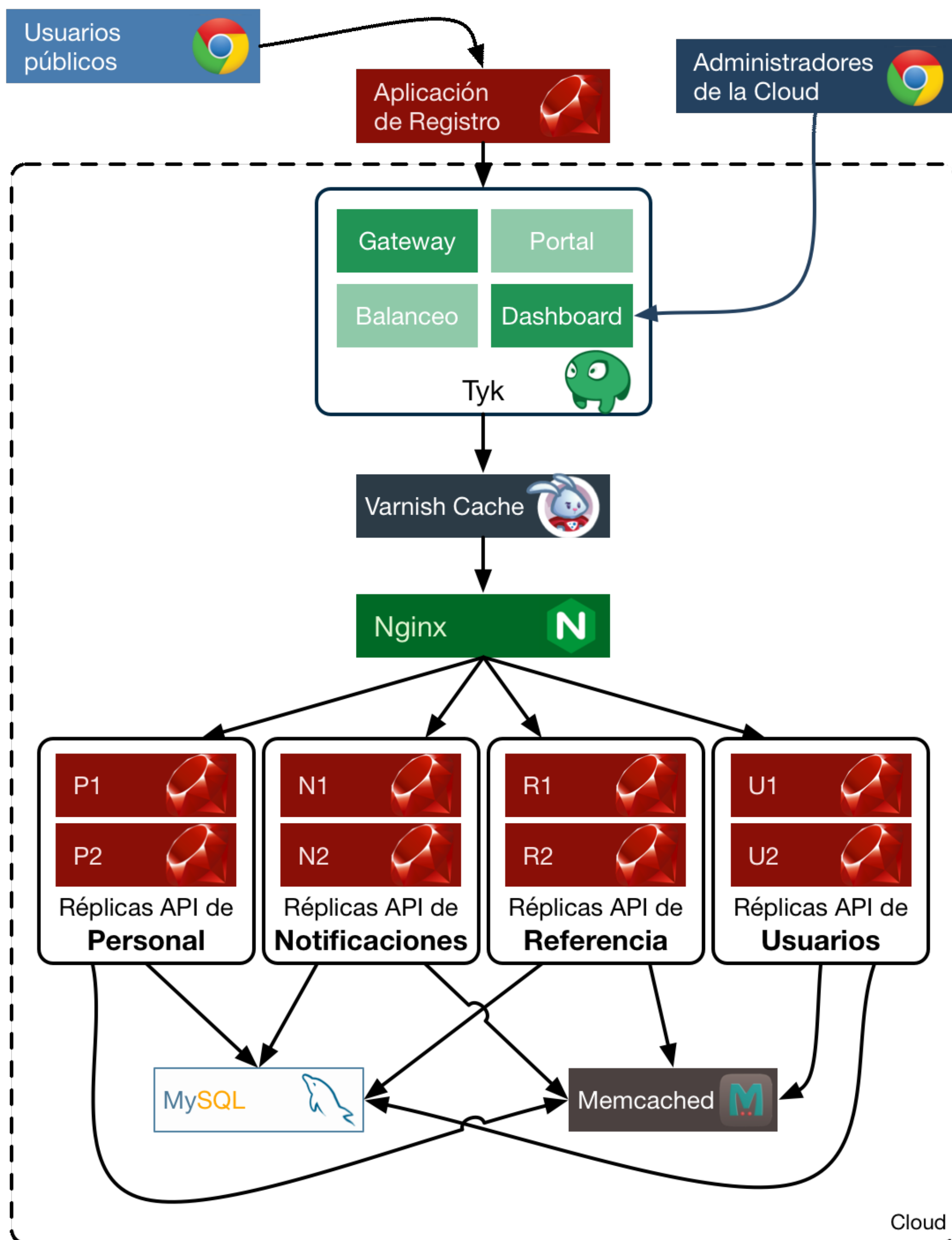


Figura 20: Visión lógica de la arquitectura del caso testigo

5.1.3. Desarrollo del prototipo funcional

En el desarrollo de la arquitectura de nuestro caso testigo hemos seguido los principios que utilizamos en nuestra oficina de trabajo, tanto en la metodología ágil de desarrollo empleada, como en las herramientas fundamentales que se basan en productos *open source*:

- Para el versionado de nuestro código utilizamos el sistema de control de versiones git, con la interfaz web que brinda el producto GitLab[53].
- En el desarrollo usamos ambientes locales en nuestras computadoras, replicando un entorno similar a aquel en que finalmente vivirán las aplicaciones en producción.
- Para armar la topología de producción, utilizamos máquinas virtuales con tecnología openvz, sobre un entorno de virtualización basado en Proxmox[54], en el cual creamos libremente tantas instancias virtuales como necesitamos.
- Como sistema base de cada instancia virtual utilizamos una versión reducida de la distribución de GNU/Linux Ubuntu 14.04, que se encuentra modificada y simplificada para correr en máquinas virtuales consumiendo menos recursos.
- Para la resolución de los nombres de las instancias virtuales instalamos un servidor de DNS BIND9[55] utilizando el subdominios para cada host perteneciente a la zona `tesis.desarrollo.unlp.edu.ar`.
- Para aislar la arquitectura y evitar problemas de seguridad, utilizamos un esquema de red privada local al servidor de Proxmox, al cual accedimos mediante modificaciones a las tablas de *routing* en nuestros equipos y la inclusión del servidor de DNS dedicado a la zona de nuestras instancias virtuales para la resolución de los nombres de cada equipo.
- Para los despliegues utilizamos Capistrano 3[56], una herramienta altamente personalizable que automatiza los pasos necesarios para realizar

[53]`https://about.gitlab.com`

[54]`http://www.proxmox.com/en/proxmox-ve`

[55]`https://kb.isc.org/article/AA-01031`

[56]`http://capistranorb.com`

la configuración, preparación, distribución y ejecución de cada una de nuestras aplicaciones (cliente y APIs).

- Para la configuración de Tyk utilizamos *Tyk Dashboard*, herramienta gratuita pero de código cerrado que distribuyen los creadores del API Gateway. En caso de no desear utilizar esa herramienta por no ser software libre podríamos utilizar la API REST que Tyk brinda para realizar esas configuraciones. En este caso utilizamos la interfaz web para poder familiarizarnos con la configuración y gestión del producto de manera más sencilla.

- Para la implementación de los servicios siguiendo el estándar JSON API utilizamos una *gema* que simplificó en varios órdenes de magnitud el desarrollo: `ActiveModelSerializers`[57]. Esta gema brinda la lógica de presentación necesaria para estructurar las respuestas acorde a la especificación JSON API que elegimos para dar forma a las respuestas de la Cloud.

- Para la implementación de la lógica de acceso a los servicios de la Cloud desarrollamos una gema propia que reutilizamos en toda aquella aplicación que funcionaba como cliente, inclusive en las aplicaciones Rails que funcionaban como API y debían consumir información de otros servicios. Un ejemplo de esto último es el servicio de personas (parte de la API de personal), en el cual se hace referencia al tipo de documento de la persona, que se encuentra efectivamente en el servicio de tipos de documento que brinda la API de referencia. Del lado del servicio de personas, el tipo de documento se identifica a partir de su clave identificatoria, y cuando se necesita acceder a los atributos del tipo de documento (descripción o abreviatura) se realiza una petición como cliente del servicio de tipos de documento, pasando por el API Gateway como lo haría cualquier otro cliente.

- En la visual de la aplicación de Registro (la que funciona como cliente de la Cloud) utilizamos el framework CSS Bulma[58], principalmente para diferenciar nuestro desarrollo como parte de este caso testigo de la estética institucional que mantienen los aplicativos desarrollados en el CeSPI, que poseen un patrón visual común y utilizan el framework

[57] `https://github.com/rails-api/active_model_serializers`

[58] `http://bulma.io`

Bootstrap 3[59].

En el proceso de diseño de esta arquitectura decidimos acotar algunos aspectos planteados en la propuesta, principalmente para simplificar la arquitectura en nuestras pruebas, ya que no revestía un cambio significativo en el proceso de desarrollo pero sí nos insumiría tiempo de preparación, configuración y pruebas sobre los nuevos nodos introducidos. Como se mencionó antes, fue en ese sentido que dejamos sin implementar la API de notificaciones.

Si bien se podría haber simplificado aún más la arquitectura, la inclusión de los nodos de Cache de Gateway y balanceo de carga se debió a la búsqueda de escalabilidad sin necesidad de replicar la capacidad de procesamiento (es decir, evitando el procesaminto innecesario) y redundancia de nodos. Al utilizar Varnish detrás del API Gateway, podemos proveer una capa de caching de mayor rendimiento en comparación a la más modesta que provee Tyk. Luego, al incluir Varnish también podríamos utilizarlo para balancear la carga de los *service components* ya que este producto lo admite, pero decidimos agregar un balanceador dedicado por detrás de la Cache de Gateway para tener un mecanismo de *fallback* por si hubiera una caída del servicio de caching, en cuyo caso podríamos *saltear* a Varnish hasta restablecer el servicio y utilizar directamente un NGINX como balanceador de carga, sin lógica de caching (la cual podría delegarse momentáneamente en Tyk).

5.1.4. Experiencia

A lo largo del desarrollo de las distintas partes involucradas en la implementación del caso testigo, pudimos apreciar cómo los conceptos clave enunciados hasta este punto en el informe se hacían presentes: desde la facilidad para realizar los despliegues virtualmente sin tiempo de caída del servicio, hasta la simplicidad introducida en el diseño y el desarrollo como producto de la clara separación de incumbencias por cada *service component*.

A su vez, la libertad de gestionar los nodos de la arquitectura nos permitió experimentar con la cantidad y distribución de los mismos. Esto es extremadamente enriquecedor a la hora de comenzar a hacer pruebas sobre las posibles variantes de una arquitectura, ya que permite toparnos con eventuales problemáticas de su implementación efectiva.

[59]`http://getbootstrap.com`

Como resultado de esta experiencia fue gratificante confirmar que las elecciones de tecnologías que hicimos habían sido atinadas y altamente beneficiosas, entre otras cuestiones, porque:

- Ruby on Rails fue un medio muy veloz para tener de manera sencilla y ágil las aplicaciones completamente funcionales, implementando técnicas de caching altamente favorables para la performance general de las aplicaciones. Nuestros conocimientos de este framework hicieron que podamos tener en poco tiempo las primeras versiones de cada uno de los *service components* y de la aplicación cliente.
- La adopción del estándar JSON API, tanto para la estructura de las respuestas de los servicios como para los mecanismos de realización de las peticiones, nos permitió utilizar gemas en ambos extremos de la comunicación que hicieron innecesaria la implementación de la lógica de conexión, transporte, serialización y des-serialización de la información.
- El uso de Tyk como API Gateway permitió que ignoremos detalles de autenticación y limitación por uso de recursos (*rate limiting*) al implementar nuestros servicios. Adicionalmente, obtuvimos beneficios agregados como poder realizar balanceo de carga de las instancias de las APIs con una configuración extremadamente sencilla, la gestión centralizada de claves de acceso, la recolección y visualización de datos estadísticos y analíticos del uso de los diferentes endpoints, y el chequeo de *uptime* (tiempo en que el servicio se encuentra activo y funcionando correctamente) de las APIs.
- Como parte del desarrollo, unificamos los criterios base para las aplicaciones que funcionarían como *service component*, creando una plantilla de aplicación en modo API como punto de partida al iniciar el desarrollo de este tipo de aplicaciones Rails.

5.1.5. Resultado

A continuación presentamos algunas capturas de pantalla en las que se pueden apreciar los distintos pasos implementados para la aplicación testigo de Registro de un nuevo usuario SSO de la Universidad Nacional de La Plata. Estos pasos pueden considerarse la implementación concreta del flujo expuesto anteriormente en la Figura 19 de este mismo capítulo.

En primer lugar, el usuario accede al sitio de registro, donde posibilita registrarse o acceder con su usuario y contraseña. Por ser el caso que nos

atañe, seguiremos el camino del registro en las próximas capturas. En la Figura 21 podemos apreciar una captura de esta página.

En el paso siguiente, se solicita al interesado en registrarse que ingrese su correo personal e institucional. Este correo debiera ser una dirección válida, perteneciente a un dominio terminado en `unlp.edu.ar`.

Una vez ingresado el correo electrónico, éste es validado y en caso de cumplirse con los requerimientos antes expuestos, se le envía un correo electrónico a la dirección ingresada con un vínculo para continuar con el proceso de registro. Este paso es necesario para confirmar la dirección ingresada por el solicitante, puede apreciarse en la Figura 22. En el momento en que se inicia el proceso de registro y justo antes de enviar el correo electrónico, se invoca al servicio de *tokens* de la API de usuarios, el cual retorna un *token* con un código único[60] que es utilizado para generar una URL única para cada inicio de registro. Esos *tokens* tienen un tiempo de vida de una hora y pasado este período, debe generarse uno nuevo comenzando desde el principio el proceso de registro.

Por los motivos expuestos con anterioridad hemos decidido no implementar el servicio de notificaciones, por lo cual en la aplicación cliente implementada el usuario es llevado directamente al paso siguiente, tal como si hubiera recibido el correo electrónico y accedido al vínculo que habría en él.

En el paso siguiente, se presenta al usuario un formulario de carga de información para que se identifique, ingresando su tipo y número de documento, lo cual dispara una consulta asíncrona mediante AJAX a la API de personas y muestra la información acorde a los datos ingresados (pudiendo mostrarse el nombre de la persona en caso exitoso o un indicador de que la persona no es un agente de la UNLP). Si los datos se corresponden con una persona que no posea un usuario, se iniciará una segunda consulta asíncrona al servicio de nombres de usuario de la misma API para que sugiera un usuario válido acorde a la política de nombres de usuario a partir del nombre y apellido de la persona identificada. Finalmente se solicita un correo electrónico alternativo y que se indiquen las Dependencias en las que el solicitante presta servicios, a modo de *captcha* para confirmar que se trate de un registro legítimo. Este paso puede observarse en la Figura 23.

Por último, se presentan los datos ingresados para que el solicitante los

[60] Más específicamente, utilizamos un Universally Unique Identifier (UUID) como código único.

confirme y genere efectivamente el nuevo usuario de SSO, como puede verse en la Figura 24. En caso de encontrar inconsistencias, puede optar por volver al paso anterior y corregirlas. Una vez confirmados los datos, se le enviará un correo mediante la API de notificaciones con los datos de la transacción y el formulario que deberá acercar personalmente a la oficina de Personal de su Dependencia. Adicionalmente, se hará un llamado al servicio de creación de usuarios de la API de usuarios para persistir la información, dejando el nuevo usuario en estado *pendiente de aprobación*, para que sea aprobado por los agentes de la oficina antes mencionada cuando reciban el formulario de solicitud firmado por el solicitante. De esta manera se concluiría el circuito de registro que hemos tomado como caso testigo.

A continuación se presentan las capturas de los pasos involucrados en el registro antes mencionados.

Registro

¡Bienvenid@!

Este es el sitio de registro para los servicios en línea de la Universidad Nacional de La Plata.

Comenzá tu registro

Para nuevos usuarios

Iniciá sesión con tu usuario

Para usuarios existentes

Figura 21: Paso 0: bienvenida a la aplicación de Registro

Registro

Registro para una nueva cuenta UNLP

Primer paso: ingresá tu correo institucional

1 2 3 4 5

Por favor, ingrese su dirección de correo electrónico institucional y personal

joaquin.v.gonzalez@dependencia.unlp.edu.ar

Esta dirección será utilizada como el medio de comunicación durante este proceso de registro y una vez que tu nueva cuenta haya sido activada, será la forma principal de contactarte.

Listo. Continuar

Figura 22: Paso 1: inicio del registro

Registro

Registro para una nueva cuenta UNLP

Segundo paso: completá tus datos

1 2 3 4 5

Este es tu correo institucional y personal

joaquin.v.gonzalez@unlp.edu.ar

Ingresá tu tipo y número de documento

Documento Nacional de Identidad | 31988189

Según la información ingresada, vos sos

Joaquín Víctor Gonzalez

Elegí un nombre de usuario para identificarte

joaquin.v.gonzalez

Este nombre de usuario está disponible.

› *Tu nombre de usuario será la forma en que te identifiques para iniciar sesión en los servicios en línea de la UNLP. Puede contener letras y puntos, y debe seguir el formato "nombre.apellidos".*

¿En qué otra dirección de correo electrónico podríamos ubicarte?

joaquin@example.org

› *Este dato es opcional y lo utilizaremos únicamente en caso de no poder contactarte en tu dirección institucional y personal.*

¿En qué Dependencias de la Universidad prestás tus servicios?

- [ ] Bachillerato de Bellas Artes "Americo D Santo"
- [ ] Ce.S.P.I
- [ ] Dirección de Deportes
- [ ] Dirección de Sanidad
- [ ] Esc. de Obstetricia FCM
- [ ] Escuela Práctica de Agric. y Ganad. "Inchausti"
- [ ] Facultad de Arquitectura y Urbanismo
- [ ] Facultad de Ciencias Agrarías y Forestales
- [ ] Facultad de Ciencias Económicas
- [ ] Facultad de Ciencias Jurídicas y Sociales
- [ ] Facultad de Ciencias Naturales
- [ ] Facultad de Humanidades y Ciencias de la Educación
- [ ] Facultad de Ingeniería
- [ ] Facultad de Periodismo y Comunición Social
- [ ] Facultad de Trabajo Social
- [ ] Liceo Victor Mercante
- [x] Presidencia - Autoridades y Doc.
- [ ] Radio Universidad
- [ ] Virtual Informática - Ingeniería
- [ ] Biblioteca Pública
- [ ] Colegio Nacional Rafael Hernandez
- [ ] Dirección de Obras y Planeamiento
- [ ] Dirección de Servicios Sociales
- [ ] Escuela Graduada Joaquín V. Gonzalez
- [ ] Escuela Universitaria de Recursos Humanos - Equipo de Salud
- [ ] Facultad de Bellas Artes
- [ ] Facultad de Ciencias Astronómicas y Geofísicas
- [ ] Facultad de Ciencias Exactas
- [ ] Facultad de Ciencias Médicas
- [ ] Facultad de Ciencias Veterinarias
- [ ] Facultad de Informática
- [ ] Facultad de Odontología
- [ ] Facultad de Psicología
- [ ] Hospital de Medicina
- [ ] Museo Samay Huasi
- [ ] Presidencia - No Docentes
- [ ] Rectorado (Función 5.20)

› *Seleccioná todas las Dependencias de la Universidad Nacional de La Plata en las cuales tengas un recibo de haberes vigente. Esta información es necesaria para corroborar que los datos que estás brindando sean correctos.*

Una vez que hayas completado los datos que te pedimos acá, vas a poder corroborar la información ingresada antes de confirmar finalmente tu solicitud de registro para tu nuevo usuario UNLP.

Listo. Continuar

Registro · 2016

Desarrollado por Miguel Carbone & José Nahuel Cuesta Luengo.

Figura 23: Paso 2: identificación del nuevo usuario

Registro

Registro para una nueva cuenta UNLP

1 2 3 4 5

Tercer paso: corroborá los datos ingresados

Te identificamos como

Joaquín Víctor Gonzalez DNI 31988189

Vas a iniciar sesión como

joaquin.v.gonzalez

Podemos contactarte en

joaquin.v.gonzalez@unlp.edu.ar Primario

No especificaste ningún email alternativo

Si la información es correcta, confirmala continuando al siguiente paso y te enviaremos a tu correo electrónico primario la información para finalizar el registro. Si considerás que algún dato es incorrecto, podés volver al paso anterior y corregirlo.

Volver al paso anterior | Listo. Continuar

Registro · 2016

Desarrollado por Miguel Carbone & José Nahuel Cuesta Luengo.

Figura 24: Paso 3: confirmación de los datos

6. Capítulo VI: Conclusión y trabajos futuros

En este capítulo final, escribiremos las conclusiones obtenidas del análisis y desarrollo que hemos realizado como parte de esta elaboración. Basándonos en lo investigado en el marco teórico sentado en el capítulo II, en las alternativas tecnológicas comparadas en el capítulo III, y en las decisiones prácticas y concretas tomadas en los capítulos IV y V, es en este apartado que recogemos los frutos producto de nuestro trabajo.

Luego de la conclusión, terminaremos por enumerar los trabajos futuros que dejamos planteados para continuar, ya en conjunto con el resto del equipo de trabajo de la Dirección de Desarrollo del CeSPI, hasta llegar a implementar y llevar a producción la nueva arquitectura para la nube de servicios de la UNLP, el resultado final de esta tesina.

6.1. Conclusión

Comenzamos esta tesina a partir de la existencia de una problemática que necesitaba ser abordada desde distintos aspectos: el teórico, el cual nos daría bases firmes sobre las cuales plantear un nuevo diseño de la arquitectura de la nube, y el práctico, en el cual aplicaríamos los nuevos conocimientos adquiridos y adoptaríamos técnicas y tecnologías modernas para plantear un caso testigo.

Durante el proceso de investigación del estado del arte en la materia, nuestro enfoque inicial fue modificándose a partir de la asimilación de nuevos conceptos. El mayor cambio que experimentó nuestra concepción de la nueva arquitectura fue al ahondar en los microservicios (Subsección 2.3), patrón que nos resultó natural y adecuado para la Cloud y su nuevo diseño. De manera similar, al conocer productos alternativos a los ESBs (Subsección 2.4 y Subsección 3.3) que preliminarmente habíamos relevado vimos la posibilidad de usar productos específicamente diseñados para la gestión de APIs de servicios, delegando en éstos tareas como la autenticación, limitación de uso de recursos, registro de eventos y accesos, y la administración de clientes y medios para su acceso a la información.

Al hacer la prueba de concepto, obtuvimos una noción palpable del costo aparejado a implementar los servicios siguiendo el patrón de microservicios y a llevar estos cambios a nuestras aplicaciones. Ese experimento le da un alto valor agregado a este trabajo, ya que al momento de estimar el tiempo y los recursos necesarios para plasmar este cambio en los desarrollos existentes y

futuros de nuestro equipo, podremos referirnos a esta experiencia. Ése es un punto clave dentro del camino que nos falta transitar para llegar al ambiente de producción con la Cloud, proceso que deberemos abordar, ya fuera del marco del presente trabajo, con el resto de nuestro equipo para planificar la transición ordenada de las aplicaciones existentes a la nueva arquitectura.

Con la implementación de una porción de la nueva arquitectura también pudimos experimentar los beneficios del uso del patrón de microservicios[8, p. 27]. Los puntos más fuertes derivados del uso de este patrón fueron la agilidad obtenida para el desarrollo, la facilidad para realizar despliegues independientes de los servicios en el ambiente de producción y, por sobre todas las cosas, el alto grado de desacoplamiento que acaban por tener los servicios entre sí y con respecto a las aplicaciones que los utilizan. Al separar los servicios por alcance, no presenta dificultades respetar el principio de una sola responsabilidad; y al separar las aplicaciones de los servicios, centralizando la lógica de negocios en estos últimos, se simplifican las aplicaciones clientes y se eliminan los posibles puntos de duplicación de código que pudieran existir en caso de tener la misma lógica de negocios presente en más de una aplicación[61]. Asimismo, la arquitectura se puede extender e incluir nuevos *service components* de manera transparente y sencilla, también debido al uso de este patrón y a la disposición de un *broker* que funciona de intermediario.

La arquitectura diseñada logra mejoras más allá de las que nos propusimos inicialmente, producto de la investigación y aplicación de conceptos que al momento de comenzar esta tesina desconocíamos, cuanto menos, formalmente.

Como hemos expuesto en el apartado *Experiencia* del caso testigo (Subsubsección 5.1.4), consideramos que la elección de tecnologías para llevar a la práctica el diseño teórico fue beneficiosa y acertada, y que el producto final palpable que hemos generado es un punto de partida concreto para transmitir lo aprendido en este trabajo.

Este trabajo nos deja una propuesta para la arquitectura de la nueva Cloud de la Universidad Nacional de La Plata que posee las características presentadas a continuación.

- **Redundante:** admite la inclusión de nodos replicados en los distintos

[61]Este tipo de situaciones no nos es ajena, ya que en la actualidad tenemos diferentes aplicaciones que tienen en común parte de su lógica de negocios y por eso acaban duplicando porciones de código, con el impacto negativo que esa repetición conlleva.

roles sin necesidad de modificar su diseño *lógico*.

- **Extensible:** permite de manera transparente y sencilla incluir nuevos servicios o quitar algunos existentes, todo sin más rodeos que configurar el *broker* acorde.
- **Escalable:** puede crecer en las distintas dimensiones del modelo *scale cube* (ver Subsubsección 4.1.1) según sea necesario.
- **Configurable:** su concepción está basada en la aceptación del cambio, y a tal fin debe poder ser modificada (o configurada) ante necesidades cambiantes.
- **Orientada a la *performance*:** la introducción de diversas capas de caching, el balanceo de la carga de procesamiento entre diferentes instancias redundantes de los nodos, y la reescritura de la librería cliente para adoptar los beneficios que los estándares y el HTTP caching están enfocados en reducir los tiempos de procesamiento y respuesta de los servicios, lo cual resulta en un mejor rendimiento general de la Cloud.
- **Basada en tecnologías Open Source:** siguiendo nuestros principios de trabajo, orientamos la elección de las herramientas utilizadas en todas las capas de la Cloud a productos Open Source, utilizadas por un número enorme de empresas y organismos, y que poseen comunidades sólidas que los respaldan y mantienen.
- **Basada en estándares abiertos:** al emplear estándares como JSON API y HTTP caching, resulta más sencilla la integración de nuestro *stack* de tecnologías con otros productos, e inclusive nos permite aprovechar herramientas desarrolladas por terceros para esos estándares sin necesidad de realizar implementaciones *ad hoc*.
- **Hecha a medida:** se origina y desarrolla con la Cloud en mente, lo cual la hace altamente específica para los casos de uso que tendrá. Así como se basa en análisis e investigaciones del estado del arte en la materia, también está planteada a partir del aprendizaje de nuestros propios errores.
- **Simple:** intentamos incluir la cantidad justa de nodos para cumplir los roles necesarios en el diseño, sin agregar capas innecesarias ni eliminar puntos posibles de expansión al quitar algún nodo que habilite la redundancia.

- **Desacoplada:** cada *service component* es una unidad lógica encargada de un conjunto concreto de tareas, cuyas interdependencias se manejan a través del *broker* y no de forma directa, lo cual acoplaría un servicio a otro.
- **Robusta:** gracias a la redundancia y el desacoplamiento antes mencionados, el uso del *broker* intermedio y las tecnologías empleadas, que han sido extensamente probadas en ambientes de producción.
- **Testeable:** su diseño descompuesto en capas y unidades independientes permite fácilmente aislar partes de la arquitectura para realizar pruebas sobre ellas.

En el siguiente apartado se presentan algunas tareas derivadas de la presente tesina para su posterior tratamiento.

6.2. Trabajos futuros

A partir del presente informe y las experiencias obtenidas, tanto a partir de la formalización teórica de los conceptos involucrados en el diseño de una arquitectura orientada a servicios, como al llevarlos a la práctica en un caso concreto, consideramos que los puntos presentados a continuación permitirían extender lo aquí desarrollado.

6.2.1. Automatización de la documentación

Si bien hemos analizado productos para documentar y mantener la documentación de las APIs desarrolladas, notamos que sería altamente beneficioso definir un proceso automático de actualización y publicación de la documentación de las mismas que se inicie cada vez que se publique una nueva versión del código de los servicios.

Esto mantendría siempre actualizada la documentación de los servicios, que deberían estar publicadas en un portal para desarrolladores que sirva para los integrantes de nuestra Dirección y para cualquier otro interesado en consumir la información pública de la Cloud de la Universidad Nacional de La Plata.

6.2.2. Automatización de la arquitectura

En nuestras pruebas, hemos manejado de manera manual la instalación y configuración de la arquitectura base que utilizamos. Si bien esto puede funcionar bien para un caso testigo como el que aquí hemos presentado, al momento de llevar una arquitectura completa a producción es conveniente tener una forma automatizada, documentada y replicable de poner en funcionamiento todos sus nodos, como pueden ser *scripts* de provisionamiento de los servidores involucrados.

En la actualidad, existe una cantidad considerable de herramientas que asisten en esta tarea, como por ejemplo Chef[62], Ansible[63] o Puppet[64], por nombrar algunos.

Se deja planteada para el futuro la necesidad de implementar esto con alguna herramienta afin, sea alguna de las aquí mencionadas u otra.

6.2.3. Extensión de la gema cliente desarrollada

El desarrollo que realizamos para dar soporte a las necesidades de la Cloud en nuestro caso testigo está limitado a aquellos puntos que se necesitó implementar. Aunque esto es un buen punto de partida y la gema resultante es completamente funcional, su alcance no sería suficiente para la implementación completa de la nueva nube de servicios.

Queda para etapas posteriores al presente trabajo la tarea de completar la lógica presente en la gema, agregando soporte para los nuevos endpoints que pudieran surgir, así como funcionalidad que asista al desarrollo de las aplicaciones cliente que no hayan sido necesarias para nuestro caso testigo.

6.2.4. Implementación de pruebas

Una buena práctica a la hora de desarrollar software es mantener una batería de pruebas a realizar sobre el producto para cerciorarnos que su ejecución genera los resultados esperados, y para garantizar que durante la evolución y el mentenimiento del mismo no se modifiquen su lógica de manera

[62] `https://www.chef.io`

[63] `https://www.ansible.com`

[64] `https://puppetlabs.com`

que impacte negativamente en dichos resultados. En el desarrollo de nuestro caso testigo obviamos la implementación de ese tipo de pruebas, dejando esto como una deuda técnica a saldar cuando este diseño comenzase a pasarse a desarrollo real. Recomendamos la realización de pruebas de unidad dentro de cada API, y de pruebas de integración entre las APIs y las aplicaciones cliente, para garantizar que ningún cambio realizado en cualquiera de las partes intervinentes en la arquitectura produzca efectos secundarios no deseados en el resto.

A. Anexo I

A.1. Aplicaciones cliente de la nube de servicios de la UNLP

Como ya hemos mencionado antes, la nube de servicios de la Universidad Nacional de La Plata es el almacén de datos de referencia y de dominio general de las aplicaciones de uso interno de la UNLP que desarrollamos en nuestra oficina. En esta sección daremos un marco más concreto en referencia a qué aplicaciones la utilizan de manera que el lector pueda comprender en detalle el alcance y las interacciones existentes que movilizan el rediseño motivo del presente trabajo.

Por último, detallaremos las dependencias existenes entre las distintas aplicaciones y la nube de servicios, indicando qué APIs utiliza cada una.

A.1.1. Albergue Universitario

Esta aplicación permite la gestión administrativa, de personal y de alumnos alojados en el Albergue Universitario de la Universidad Nacional de La Plata. Los usuarios del sistema son el personal administrativo, del comedor y de Guardia Edilicia que desempeñan sus tareas en esa institución, permitiéndoles manejar las entradas y salidas de los alumnos, planificar las comidas acorde a la cantidad de comensales, y llevar un registro detallado de las actividades que por reglamento deben realizar los alumnos como parte de la beca que les permite residir en el albergue.

Es una aplicación desarrollada en Ruby on Rails versión 3, que actualmente se encuentra en etapa de mantenimiento correctivo de errores.

A.1.2. Asociador

El asociador de documentos es una aplicación desarrollada para el uso conjunto entre el área de Digitalización de documentos del CeSPI y la Dirección de Salud de la Universidad Nacional de La Plata. Su objetivo es permitir que ése área suba en formato PDF las carpetas médicas históricas que digitaliza y luego las asocie a los agentes de la Universidad Nacional de La Plata, de modo tal que el personal de la Dirección de Salud pueda consultar en línea dichas carpetas sin necesidad de contar con una oficina llena de biblioratos con las carpetas médicas otorgadas en los últimos 20 años en soporte papel.

Esta aplicación está desarrollada en Ruby on Rails versión 3, y actualmente se encuentra en etapa de mantenimiento correctivo de errores.

A.1.3. Becas UNLP

La aplicación de becas de la Universidad Nacional de La Plata posee dos partes principales: una pública donde los alumnos (y futuros alumnos) de la Universidad pueden identificarse y solicitar algunas de las becas que la Universidad ofrece, y una privada accesible por personal de la Dirección de Becas que les permite realizar la asignación de las becas acorde a un órden de mérito que calcula el sistema, habilitar o deshabilitar becas de entre la oferta existente y agregar nuevas para que sean publicadas.

Está desarrollada en symfony versión 1.4, y actualmente se encuentra en etapa de mantenimiento correctivo de errores con soporte para nuevos requerimientos.

A.1.4. Libretas Sanitarias

Permite al personal de la Dirección de Salud la gestión de la información referente a las libretas sanitarias de los alumnos de la Universidad Nacional de La Plata, la obtención de turnos y la consulta del estado de los trámites relacionados.

Se encuentra implementada con el *framework* Ruby on Rails versión 3, y en este momento se encuentra bajo proceso de refactor y migración a Ruby on Rails versión 4.

A.1.5. Licencias Médicas

La Dirección de Salud de la UNLP utiliza este sistema para gestionar las solicitudes de carpetas médicas, el tratamiento de esas solicitudes y el seguimiento de las eventuales licencias otorgadas a partir de ellas o las juntas médicas que pudieran surgir.

Esta aplicación fue desarrollada en symfony versión 1.2, y se encuentra en etapa de mantenimiento correctivo de errores. Tenemos planificada para la segunda mitad del año 2016 una reescritura de la aplicación para actualizar su *stack* y extender su funcionalidad para brindar mejor soporte a situaciones no previstas en la versión actual, como por ejemplo permitir que los agentes

de la Universidad Nacional de La Plata soliciten en línea sus licencias, en lugar de tener que presentar en papel o telefónicamente los pedidos.

A.1.6. Programa "Mejor Aire"

Aplicación que gestiona las inscripciones de agentes de la UNLP al programa "Mejor Aire", una iniciativa de la Universidad para ayudar a dejar de fumar. Los inscriptos al programa participan semanalmente de reuniones de grupo y tienen controles periódicos para hacer un seguimiento de su evolución en el proceso. Mediante esta aplicación, los profesionales que llevan adelante el programa organizan los turnos, la asistencia a las reuniones y registran los chequeos realizados a cada inscripto.

Este sistema fue desarrollado utilizando symfony 1.2, y actualmente se encuentra en etapa de mantenimiento correctivo de errores.

A.1.7. Proyectos de Extensión

El sistema actual de gestión de Proyectos de Extensión de la Universidad Nacional de La Plata es el resultado de dos iteraciones de análisis de requerimientos e implementación, a partir de las cuales se depuraron las necesidades reales que la Secretaría de Extensión Universitario tenía para la versión *on line* del proceso de presentación, acreditación, evaluación y adjudicación de los proyectos postulados para el programa de subsidio de proyectos de extensión de la Universidad Nacional de La Plata. La aplicación es utilizada por los directores y miembros de los proyectos para presentar sus propuestas, por los Secretarios de Extensión para avalar los proyectos presentados desde su Unidad Académica, por los integrantes de las comisiones evaluadoras para analizar y calificar las presentaciones, y por el personal de la Secretaría de Extensión para el otorgamiento final y seguimiento posterior de los proyectos subsidiados.

Este sistema fue reescrito en el año 2015, pasando de un desarrollo en symfony 1.1 a una aplicación totalmente renovada y con mucha más funcionalidad implementada utilizando el *framework* Ruby on Rails versión 4. Se encuentra en desarrollo activo.

A.1.8. Recibos de sueldo

La aplicación de recibos de sueldo permite a los agentes de la Universidad Nacional de La Plata acceder de manera cómoda y en todo momento a sus recibos de sueldo, con la posibilidad de descargarlos para utilizarlos a su conveniencia, sin necesidad de acercarse a la oficina de Personal de su Dependencia para obtenerlos.

Esta aplicación también fue reescrita por completo en el año 2015, proceso mediante el cual pasó de ser un sistema symfony 1.4 a uno desarrollado con Ruby on Rails versión 4. Se encuentra en etapa de mantenimiento correctivo de errores e implementación de nuevos requerimientos, en caso que surjan.

A.1.9. Responsables

Este sistema centraliza la información que la oficina de Responsables de la Universidad Nacional de La Plata gestiona para el seguimiento y la rendición de cuentas sobre el destino dado a los fondos que la Universidad posee. Registra todos los gastos imputados a las distintas partidas presupuestarias, cerciorándose que no existan inconsistencias entre los montos otorgados, su destino y el uso efectivo de los mismos.

Fue desarrollada con Ruby on Rails versión 3, y en la actualidad se encuentra en etapa de mantenimiento correctivo de errores.

A.1.10. Acceso Único (SSO)

En el año 2015 se implementó de manera global la centralización de cuentas de usuario de los agentes de la Universidad Nacional de La Plata para las aplicaciones que nuestra Dirección de Desarrollo del CeSPI realiza, haciendo que cada persona disponga de una única cuenta que le permita acceder a aquellas aplicaciones que utilice para su labor diaria, así como también a sus datos personales y recibos de sueldo. Este proceso comprendió la creación de más de 5000 cuentas de usuario, la implementación de un proceso de autoregistro para los agentes y capacitaciones para los agentes de las oficinas de Personal de cada Dependencia en su uso.

Es una aplicación desarrollada en tres partes, dos de gestión (una para autogestión de cada usuario) y otra de administración general de los datos de usuarios, permisos y aplicaciones utilizando Ruby on Rails versión 4 y otra que implementa el estándar *Security Assertion Markup Language* (SAML)

para proveer autenticación y autorización al resto de las aplicaciones. Se encuentra en mantenimiento correctivo de errores, con una planificación para extender su funcionalidad en el año 2016.

A.1.11. Sueldos

La aplicación de Sueldos de la UNLP es una reimplementación completa del sistema que actualmente se encuentra en uso para liquidar los sueldos de la Universidad. Como aplicación, además de realizar el cálculo de la liquidación de sueldos, es el puntapié inicial para un sistema integral de gestión de recursos humanos de la Universidad Nacional de La Plata. Este proyecto lleva alrededor de dos años en desarrollo y está planificado para ser puesto en producción en la primer mitad del año 2016.

Esta aplicación está desarrollada en Ruby on Rails versión 4, y se encuentra en desarrollo activo.

A.1.12. Títulos

Aplicación desarrollada para la Oficina de Títulos que digitaliza el proceso de solicitud de los títulos otorgados por la Universidad Nacional de La Plata a sus alumnos, comprendiendo todas las etapas involucradas en el mismo desde la solicitud inicial hasta la entrega final del título en papel.

Este sistema fue desarrollado utilizando Ruby on Rails 3, se encuentra en mantenimiento correctivo de errores. Se tiene planificada su reescritura para el año 2016 debido a recientes cambios en el alcance inicialmente definido.

A.1.13. Dependencias con la nube de servicios

A continuación se presenta un cuadro que describe qué grupos de datos de los que provee la nube de servicios utilizan las aplicaciones enumeradas en este anexo.

En este cuadro se indica con un tilde (✓) en qué grupos de APIs depende cada aplicación, siendo los posibles grupos los siguientes:

- **Ref** comprende las APIs de datos de referencia.
- **Alumnos** abarca las APIs de datos de alumnos de la UNLP.

- **Personal** referencia las APIs de datos de agentes que trabajan en la Universidad Nacional de La Plata.
- **Usuarios** indica que la aplicación usa las APIs de datos de cuentas de usuarios registrados en el Sistema de Acceso Único.

Aplicación	**Ref**	**Alumnos**	**Personal**	**Usuarios**
Albergue Universitario	✓	✓		
Asociador	✓		✓	
Becas UNLP	✓	✓		
Libretas Sanitarias	✓	✓		
Licencias Médicas	✓		✓	
Programa “Mejor Aire”	✓		✓	
Proyectos de Extensión	✓	✓	✓	✓
Recibos de sueldo	✓		✓	
Responsables	✓			
Acceso Único (SSO)	✓	✓	✓	✓
Sueldos	✓		✓	
Títulos	✓	✓		

Cuadro 1: Dependencias de las aplicaciones con los servicios de la nube

B. Anexo II

B.1. *Endpoints* de la nube de servicios actual

Aquí presentamos una lista de todos los endpoints de la nube de servicios de la UNLP tal como se encuentra al momento de escritura del presente trabajo. El propósito de este listado es mostrar las inconsistencias existentes en la forma de organizar las APIs y la alta densidad de puntos de acción que provee la nube, de los cuales una gran parte no se encuentra en uso. Identificaremos los endpoints por los patrones de sus URLs (omitiremos el dominio y el protocolo para simplificar la lista):

- `/api/academic_data.json/:id`
- `/api/academic_degree_type.json`
- `/api/academic_degree_type.json/:id`
- `/api/academic_degree_type.json/count`
- `/api/academic_unit.json`
- `/api/academic_unit.json/:id`
- `/api/academic_unit.json/count`
- `/api/academic_unit/:id/academic_unit_college_degree.json`
- `/api/academic_unit/:id/academic_unit_college_degree.json/count`
- `/api/academic_unit/:id/career.json`
- `/api/academic_unit/:id/career.json/count`
- `/api/academic_unit/:id/career/:career_id/career_programme.json`
- `/api/academic_unit/:id/career/:career_id/career_programme.json/count`
- `/api/academic_unit/:id/career/:career_id/career_programme/:career_programme_id/career_programme_degree.json`
- `/api/academic_unit/:id/career/:career_id/career_programme/:career_programme_id/career_programme_degree.json/count`

- /api/academic_unit/:id/career/:career_id/career_programme/:career_programme_id /career_subject.json
- /api/academic_unit/:id/career/:career_id/career_programme/:career_programme_id /career_subject.json/count
- /api/academic_unit/:id/degree.json
- /api/academic_unit/:id/degree.json/count
- /api/academic_unit/:id/degree/:degree_id/career_programme_degree .json
- /api/academic_unit/:id/degree/:degree_id/career_programme_degree .json/count
- /api/academic_unit/:id/paycheck.json
- /api/academic_unit/:id/paycheck.json/count
- /api/academic_unit/:id/paycheck.json/search/:query
- /api/academic_unit/:id/paycheck.json/search/:query/count
- /api/academic_unit/:id/person.json
- /api/academic_unit/:id/person.json/count
- /api/academic_unit/:id/person.json/search/:query
- /api/academic_unit/:id/person.json/search/:query/count
- /api/academic_unit_college_degree.json/:id
- /api/career.json/:id
- /api/career_programme.json/:id
- /api/career_programme_degree.json/:id
- /api/career_subject.json/:id
- /api/census_data.json/:id
- /api/city.json/:id
- /api/country.json

- /api/country.json/:id
- /api/country.json/count
- /api/country/:id/state.json
- /api/country/:id/state.json/count
- /api/country/:id/state/:state_id/department.json
- /api/country/:id/state/:state_id/department.json/count
- /api/country/:id/state/:state_id/department/:department_id/city .json
- /api/country/:id/state/:state_id/department/:department_id/city .json/count
- /api/degree.json/:id
- /api/department.json/:id
- /api/document_type.json
- /api/document_type.json/:id
- /api/document_type.json/count
- /api/gender.json
- /api/gender.json/:id
- /api/gender.json/count
- /api/marital_status.json
- /api/marital_status.json/:id
- /api/marital_status.json/count
- /api/official_scale.json
- /api/official_scale.json/:id
- /api/official_scale.json/count
- /api/official_scale/:id/personal_scale.json

- /api/official_scale/:id/personal_scale.json/count
- /api/paycheck.json/:id
- /api/person.json
- /api/person.json/:id
- /api/person.json/count
- /api/person.json/search/:id
- /api/person/:id/academic_data.json
- /api/person/:id/academic_data.json/count
- /api/person/:id/academic_unit/:academic_unit_id/paycheck.json
- /api/person/:id/academic_unit/:academic_unit_id/paycheck.json/count
- /api/person/:id/academic_unit/:academic_unit_id/paycheck.json/search /:query
- /api/person/:id/academic_unit/:academic_unit_id/paycheck.json/search /:query/count
- /api/person/:id/census_data.json
- /api/person/:id/census_data.json/count
- /api/person/:id/is_graduated.json/:query_1/:query_2/:query_3
- /api/person/:id/paycheck.json
- /api/person/:id/paycheck.json/count
- /api/person/:id/paycheck.json/search/:query
- /api/person/:id/paycheck.json/search/:query/count
- /api/person/:id/person_email.json
- /api/person/:id/person_email.json/count
- /api/person/:id/person_high_study.json
- /api/person/:id/person_high_study.json/count

- /api/person/:id/person_role.json
- /api/person/:id/person_role.json/count
- /api/person/:id/personal_age.json
- /api/person/:id/personal_age.json/count
- /api/person/:id/personal_charge.json
- /api/person/:id/personal_charge.json/count
- /api/person_email.json/:id
- /api/person_high_study.json/:id
- /api/person_role.json/:id
- /api/personal_age.json/:id
- /api/personal_charge.json/:id
- /api/personal_scale.json/:id
- /api/state.json/:id
- /api/student_status.json
- /api/student_status.json/:id
- /api/student_status.json/count
- /api/study_category.json
- /api/study_category.json/:id
- /api/study_category.json/count
- /api/study_category/:id/study_type.json
- /api/study_category/:id/study_type.json/count
- /api/study_type.json/:id

Referencias

[1] akfpartners.com. Splitting applications or services for scale, Mayo 2008.

[2] Microsoft Corporation. La arquitectura orientada a servicios (soa) de microsoft aplicada al mundo real. pages 2–3, Diciembre 2006.

[3] D'Emic David, Dossot; John. *Mule in Action.* Manning, Febrero 2010.

[4] Jeff Davis. *Open Source SOA.* Manning Publications Co, softcover edition, 2009.

[5] Thomas Erl. Soa: Principles of service design, 2007.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000. AAI9980887.

[7] Poul Henning Kamp. Notes from the architect, Agosto 2008.

[8] Richards Mark. Software architecture patterns - understanding common architecture patterns and when to use them. 2015.

[9] Richards Mark. *Software Architecture Patterns - Understanding Common Architecture Patterns and When to Use Them.* O'Reilly, Febrero 2015.

[10] Stine Matt. Migrating to cloud-native application architectures. page 48, Febrero 2015.

[11] L. Mufioz, J.N. Mazon, and J. Trujillo. Etl process modeling conceptual for data warehouses: A systematic mapping study. *Latin America Transactions, IEEE (Revista IEEE America Latina)*, 9(3):358–363, June 2011.

[12] Josuttis Nicolai M. *SOA in Practice: The Art of Distributing System Design.* O'Reilly, paperback edition, 2007.

[13] Michael T. Nygard. *Release It!* The Pragmatic Programmers, Febrero 2007.

[14] Francisco Velázquez; Kristian Lyngstøl; Tollef Fog Heen; Jérôme Renard. *The Varnish Book.* Manning, 2016.

[15] Arnon Rotem-Gal-Oz. *SOA Patterns.* Manning Publications Co, softcover edition, 2012.

Glosario

ACL

Access Control List. 61

antipatrón de diseño de software

es un patrón de diseño que encamina el software a una situación desfavorable o problemática. 10

API

Application Programming Interface. 14–20, 30, 42, 43, 45–47, 59–62, 64–66, 68–70, 72, 74, 76, 78–81, 83, 88, 95, 99–102, 105, 107–109, 111, 116–119, 122–126, 130, 133, 135, 136, 140–142

aplicaciones satélite

término que hemos acuñado en nuestro trabajo diario para calificar a aquellas aplicaciones que utilizan la nube de servicios de la UNLP. 12, 18

bloating

exceso de código y funcionalidad a causa de la inclusión de lógica que al momento del diseño o desarrollo se considera que *podría ser útil*, pero que en la práctica no se utiliza. 2

captcha

prueba automatizada utilizada para determinar cuándo el usuario es o no humano. 113

CORBA

Common Object Broker Architecture. 21, 22

CORS

Cross-Origin Resource Sharing. 61

DCOM

Distributed Computing Object Model. 21, 22

documentación viviente

versión en línea de una documentación que puede utilizarse para probar aquello que ella describe. En el caso específico de las APIs web, permite consumir endpoints de ejemplo para conocer qué parámetros admiten y qué respuestas ofrecen. 99, 100, 102

DSL

Domain-Specific Language. 42, 102

endpoint

punto de conexión o acceso a un servicio. 15, 16, 65, 68, 70, 74, 99, 102, 134, 142

ESB

Enterprise Service Bus. 29–32, 59, 89, 107, 130

ETL

Extract, Transform and Load. 17

FTP

File Transfer Protocol. 21, 91

git

herramienta de control de versiones *open source* con esquema distribuido, ágil y simple de utilizar.
`https://git-scm.com`. 8, 121

HATEOAS

Hypertext As The Engine Of Application State. 20, 35

HMAC

Hash Message Authentication Code, protocolo criptográfico para la firma de mensajes, definido en la RFC 2104.
`https://tools.ietf.org/html/rfc2104`. 61, 79

HTML

HyperText Markup Language. 15, 41, 42

HTTP

HyperText Transfer Protocol. 13, 14, 22, 52–55, 64, 76, 80, 90, 91, 93, 100, 132

HTTPS

HyperText Transfer Protocol Secure. 91

hypermedia

forma básica de conexión de contenidos en la web, estableciendo vínculos lógicos a distintos niveles o hipervínculos entre los distintos medios que la componen. 14, 20, 35, 36

JSON

JavaScript Object Notation. 15, 16, 19, 20, 41, 42, 45, 46, 48–50, 54, 64, 80, 101

JWT

JSON Web Tokens, estándar de intercambio de mensajes utilizando tokens de control definido en la RFC 7519.
`https://tools.ietf.org/html/rfc7519`. 61

Memcached

es un almacen de datos en memoria distribuido *open source*, organizado en pares clave-valor que es habitualmente utilizado como cache compartida para evitar accesos a recursos más costosos del lado del servidor.
`http://memcached.org`. 19, 43, 119

MVC

Model-View-Controller. 40–42

NFS

Network File System. 21

NoSQL

nueva tendencia en motores de bases de datos que están orientadas a mejorar problemas posibles del paradigma tradicional de las bases de datos relacionales. Como principio, son no relacionales, distribuidas y escalables horizontalmente. Algunos ejemplos son Hadoop, Cassandra, CouchDB y MongoDB. 19, 70

OAI

Open API Initiave. 101, 102

OASIS

Organization for the Advancement of Structured Information Standards. 23

Redis

es un almacen en memoria de datos de código abierto, que suele ser utilizado como *cache*, base de datos de alto rendimiento (para conjuntos reducidos de datos) o cola de mensajes entre procesos distribuidos. `http://redis.io`. 19, 43, 72, 78

refactor

técnica utilizada en el desarrollo de software en la que, luego de identificar puntos posibles de mejora para una pieza de software, se modifica su implementación interna manteniendo su interfaz externa y comportamiento. 12, 137

REST

es un estilo de arquitectura de sistemas distribuidos definida por Roy T. Fielding en su tesis Doctoral. Referirse a la Subsección 2.5 para mayores detalles.. 46

REST

REpresentational State Transfer. 14, 15, 20, 33, 35–38, 99–102

RESTful

término que se utiliza para denotar aquellas APIs que cumplen, al menos parcialmente, con las propiedades de la arquitectura de sistemas distribuidos REST.. 14

RPC

Remote Procedure Call. 21, 22

Ruby on Rails

es un *framework* hecho en Ruby para el desarrollo ágil de aplicaciones web. Es, de hecho, *el framework* web por excelencia del lenguaje. `http://rubyonrails.org`. 18, 19, 117, 136–140

SAML

Security Assertion Markup Language. 139

Sinatra

es un DSL para el desarrollo ágil de aplicaciones web implementado en el lenguaje Ruby.
`http://sinatrarb.com`. 18, 19

SOA

Service-Oriented Architecture. 23–25, 27, 29, 30, 84, 90, 107

SOAP

Simple Object Access Protocol. 13, 22, 30

SPOF

Single Point Of Failure. 2

SSL

Secure Sockets Layer. 61

SSO

Single Sign-On. 112, 113, 119, 124, 125

subversion

herramienta de control de versiones centralizada, en la actualidad mantenida por la *Apache Software Foundation*.
`http://subversion.apache.org`. 8

symfony

es un *framework* web desarrollado en el lenguaje PHP, que al momento de elaboración del presente trabajo se encuentra en su segunda versión mayor. A modo de referencia, en nuestros desarrollos siempre utilizamos la versión 1.x.
`http://symfony.com`. 8, 10, 17, 18, 20, 137–139

TTL

Time To Live. 19

UDDI

Universal Description Discovery and Integration. 24

URI

Uniform Resource Identifier. 15, 47, 100

URL

Uniform Resource Locator. 14–17, 36, 45, 50, 52, 54, 74, 80

URN

Uniform Resource Name. 36

Universally Unique Identifier (UUID)

Es un número de 16 bytes que suele expresarse como 32 caracteres hexadecimales y 4 guiones, siguiendo el formato `0000000-0000-0000-0000-000000000000`. Tienen la característica de tener chances extremadamente bajas de generar colisiones cuando se los genera aleatoriamente.. 125

Web Service

estándar desarrollado por la W3C que engloba diferentes tecnologías utilizadas para realizar comunicaciones máquina-máquina sobre el medio web.
`http://www.w3.org/TR/ws-arch/#whatis`
`http://www.w3.org/TR/ws-arch/#technology`. 13, 14

WSDL

Web Services Description Language. 13, 24

XML

eXtensible Markup Language. 13, 22, 41, 46

XSD

XML Schema. 24

YAML

Yet Another Markup Language. 72, 99, 101

Índice de figuras

1. Interacciones involucradas en el listado de becas por Unidad Académica . . . 16
2. Arquitectura básica de microservicios . . . 27
3. Arquitectura de microservicios con mensajería centralizada . . 28
4. Diagrama ejemplificando la integración punto a punto . . . 31
5. Un ESB como interceptor balancea la carga de los proveedores de servicios . . . 32
6. Esquema de integración de Kong en nuestra propuesta . . . 69
7. Arquitectura interna de API Umbrella . . . 71
8. Lógica de decisión del *Gatekeeper* de API Umbrella . . . 73
9. Interfaz web de carga de un *backend* de servicios de API Umbrella . . . 74
10. Interfaz web de análisis del uso de los servicios de API Umbrella 77
11. Esquema de integración de Tyk en nuestra propuesta . . . 84
12. Esquema de integración de WSO2 ESB en nuestra propuesta . 88
13. Forward proxy y reverse proxy . . . 91
14. Varnish Reverse Proxy . . . 94
15. Resultados de la encuesta de uso de servidores web de netcraft, enero 2016 . . . 96
16. El modelo de escalabilidad *scale cube* . . . 106
17. Patrón de tolerancia a fallos *Circuit Breaker* . . . 110
18. Arquitectura propuesta para la Cloud de la UNLP . . . 111
19. Proceso de autoregistro de un nuevo usuario de acceso único . 115
20. Visión lógica de la arquitectura del caso testigo . . . 120
21. Paso 0: bienvenida a la aplicación de Registro . . . 127
22. Paso 1: inicio del registro . . . 127
23. Paso 2: identificación del nuevo usuario . . . 128
24. Paso 3: confirmación de los datos . . . 129

Listado de bloques de código

1. Ejemplo de clase PHP de referencia de la etapa 1 de la nube de servicios . . . 9
2. Recurso HAL de ejemplo . . . 48
3. Documento JSON API representando un recurso . . . 51
4. Petición de una colección de recursos JSON API . . . 52
5. Petición de un recurso relacionado en JSON API . . . 52
6. Petición de un recurso en JSON API . . . 52
7. Encabezado HTTP de respuesta exitosa JSON API . . . 53
8. Respuesta exitosa a petición de una colección de recursos en JSON API . . . 53
9. Respuesta JSON API para una petición exitosa a un recurso no encontrado . . . 54
10. Respuesta JSON API para una petición exitosa a una colección de recursos vacía . . . 54
11. Respuesta JSON API para una petición exitosa a una relación 56
12. Petición de un recurso y relaciones asociadas en JSON API . . 56
13. Petición de un recurso utilizando *sparse fieldsets* e `include` en JSON API . . . 57
14. Petición indicando el orden de una colección de recursos en JSON API . . . 57
15. Petición indicando parámetros de paginación en JSON API . . 57
16. Preparación y arranque de Kong . . . 64
17. Comandos para agregar un API a Kong . . . 65
18. Comandos para habilitar autenticación mediante clave sobre una API . . . 66
19. Comandos para habilitar un *consumer* y su clave . . . 67
20. Comandos para probar la autenticación mediante clave antes habilitada . . . 68
21. Preparación y arranque de API Umbrella . . . 72
22. Prueba de uso de API Umbrella . . . 75
23. Preparación del servidor para instalar Tyk . . . 82
24. Instalación y arranque de Tyk . . . 83
25. Comando para descargar los fuentes de WSO2 ESB . . . 86
26. Comando para descomprimir zip . . . 86
27. Comando para construir WSO2 ESB . . . 87
28. Comandos para configurar variables de entorno . . . 87
29. Verificamos la variable de entorno JAVA_HOME . . . 87
30. Comando para iniciar el servicio WSO2 ESB . . . 88

31. Actualización del sistema de base 92
32. Instalación de Squid 92
33. Copia de respando de configuración de Squid 92
34. Configuración de Squid 92
35. Renicio del servicio Squid 93
36. Verificación del funcionamiento de Squid 93
37. Instalación de Varnish 94

Índice de cuadros

1. Dependencias de las aplicaciones con los servicios de la nube . 141