Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS UNIVERSIDAD DE CHILE

Informe de Práctica

CC4901 – Práctica Profesional I

Empresa:	NIC Chile Research Labs
Alumno:	Manuel Olguín
Carrera:	Ingeniería Civil en Computación
RUT:	18.274.982 – 6
E-Mail:	molguin@dcc.uchile.cl
Tel:	+56 9 7463 6997

3 de abril de 2016
Santiago, Chile.

1. Certificado de la Empresa

2. Observaciones

Índice

Índice de figuras

3. Resumen

El trabajo se realizó entre Agosto y Octubre del 2015 en NIC Chile Research Labs, el laboratorio de investigación y desarrollo en protocolos de internet de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile. Consistió en

1. rediseñar e actualizar parte de la API del backend de BeCity, una aplicación Android desarrollada en el laboratorio, para que se adhiriera al modelo REST.

2. diseñar e implementar un microservicio de manejo de imágenes para el proyecto SUR.

BeCity es una aplicación móvil Android orientada a ciclistas, la cual pretende facilitar el tránsito del usuario por la ciudad. Cuenta con un servicio de búsqueda de rutas con ciclovías, y permite grabar los recorridos del usuario, guardando estadísticas de velocidad, distancia recorrida, calorías quemadas, etc. Funciona bajo el paradigna cliente-servidor: existe un servicio central (el backend) que almacena y maneja los datos, y la aplicación Android (el cliente) se comunica con este servicio para actualizar y obtener información. Esta comunicación se realiza mediante una API (Application Programming Interface) usando HTTP (HyperText Transfer Protocol), sobre la cual se realizó el trabajo previamente mencionado.

El rediseño de esta API tenía como fin su modernización y adaptación a estándares modernos, en específico el modelo REST (REpresentational State Transfer)

SUR (Southern URban observatory) es una plataforma que pretende centralizar todos los proyectos desarrollados en NICLabs. En este sentido, SUR se plantea como un backend unificado para los distintos servicios, ofreciendo funcionalidades comunes como autentificación de usuarios y manejo de recursos como imágenes, y generando una integración más estrecha entre los servicios.

El trabajo realizado en SUR consistió en el diseño y posterior implementación de un pequeño servidor de imágenes, capaz de administrar recursos gráficos de manera eficiente, escalable y simple. Dicho servicio se encuentra actualmente estable y funcionando en la plataforma SUR.

4. Introducción

4.1. Lugar de Trabajo

NIC Chile Research Labs (en adelante, NICLabs)[1], es el laboratorio de redes de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile. Fundado en 2007, sus principales áreas de investigación y desarrollo son protocolos de internet y seguridad informática.

El trabajo detallado en el presente informe se desarrolló entre octubre y diciembre del año 2015, de manera presencial en las instalaciones del laboratorio ubicadas en Av. Blanco Encalada 1975, Santiago, Chile.

4.2. Equipo de Trabajo

NICLabs cuenta tanto con desarrolladores e investigadores de tiempo completo, así como con estudiantes de pre- y postgrado. En específico, el trabajo se desarrolló en una sala compartida con otras 6 personas de distintas especialidades (diseñadores, ingenieros y estudiantes de pregrado).

En cuanto a los proyectos descritos en el presente informe, ambos se desarrollaron bajo la dirección de Felipe Lalanne[2], investigador y desarrollador experimentado del laboratorio. Cabe destacar que si bien existe un equipo de trabajo formalmente conformado para BeCity, éste sólo maneja y actualiza la versión actual del software, y por ende el practicante no se unió directamente a éste, ya que su trabajo consideraba el rediseño completo del backend. Por otro lado, no existe grupo de trabajo para SUR, y el practicante trabajó directamente bajo la tutela del supervisor.

4.3. Software y Conceptos Importantes

En términos generales, ambos proyectos se desarrollaron en el lenguaje de programación Python[4], utilizando además JetBrains Pycharm 5 Professional Edition[5] como IDE (*Integrated Development Environment* - Entorno de Desarrollo Integrado) principal.

4.3.1. REST

El modelo REST (por sus siglas en inglés, "Representational State Transfer") es una serie de restricciones arquitecturales que, aplicadas a un servicio web, inducen una serie de propiedades deseables (e.g. escalabilidad, estabilidad, rendimiento, etc). Al estar relacionado sólamente con la estructura arquitectural del software, no especifica detalles de implementación, y es independiente del lenguage de programación escogido.

A continuación, se expondrán brevemente las restricciones impuestas por el modelo REST:

1. **Modelo Cliente - Servidor** Debe existir una clara separación de responsabilidades mediante una interfaz uniforme. Por ejemplo, el servidor debe encargarse del almacenamiento y procesamiento de los datos, pero no de la representación visual de éstos, y vice-versa en el caso del cliente.
2. **Statelessness - Ausencia de Estados** El servidor no debe guardar información del estado del cliente entre solicitudes. El estado es almacenado por el cliente; cada solicitud debe ser independiente y contener toda la información necesaria para realizarse.
3. **Caché** El servidor debe poder guardar solicitudes en caché para poder responder futuras solicitudes de manera más expedita.
4. **Transparencia de Capas** El cliente no debe poder determinar si está conectado directamente al servidor o a un servicio intermedio (proxy, etc).
5. **Uniformidad de la Interfaz** A su vez puede separarse en:
   - *Separación de recursos y su representación* La representación externa de los recursos (datos) debe ser independiente de como son almacenados por el servidor. A su vez, estas representaciones contienen toda la información necesaria para poder modificar el recurso original.
   - *Identificación Uniforme de Recursos* Por ejemplo, mediante URI's[6].
   - *Mensajes Autodescriptivos* Cada mensaje incluye información de cómo procesarse.

4.3.2. API

Una API (Application Programming Interface), en el contexto de servicios web, es una interfaz de interacción entre los componentes de un modelo servidor-cliente. Específicamente en el caso de una API RESTful, consiste en un conjunto de puntos de enlace (o, cómo se les referirá en el presente informe, *"vistas"*) públicos en el servidor, a través de los cuales un cliente puede comunicarse e interactuar con el servicio central, utilizando verbos estándar de HTTP (`GET`, `PUT`, `POST` y `DELETE`).

4.3.3. PostgreSQL

PostgreSQL[3] es una de las principales implementaciones modernas del lenguaje SQL (Structured Query Language - Lenguaje Estructurado de Consultas) para bases de datos. Es un sistema de base de datos relacional ampliamente utilizado en la industria, y en específico, en BeCity se ocupa para el almacenamiento y consulta de los datos proporcionados por los usuarios.

4.3.4. Python

Python es un lenguaje de programación interpretado, dinámico y de alto nivel, con énfasis en la fácil lectura de su código y su brevedad sintáctica. Es un lenguaje multipropósito, apto para programas de pequeña y gran escala, y con soporte para múltiples paradigmas de programación (orientado a objetos, funcional, imperativo, etc).

En el contexto del trabajo realizado, Python, principalmente en su versión 2.7 (aunque en algunas instancias se ocupó también la versión más nueva del lenguaje, 3.5), fue el lenguaje escogido para el desarrollo de los servicios. Esto, ya que debido a su popularidad, cuenta con una gran cantidad de librerías bien documentadas y mantenidas que facilitaron el desarrollo.

4.3.5. JSON

JSON (JavaScript Object Notation), es un estándar abierto de codificación de datos para la comunicación entre servidor y cliente, en la cual los datos se codifican en pares atributo-valor en "cleartext" (es decir, texto legible por humanos). Por ejemplo, el siguiente extracto ilustra una posible codificación de la FCFM en JSON:

```
{
    "Nombre": "Facultad de Ciencias Fisicas y Matematicas",
    "Universidad": "Universidad de Chile",
    "Direccion": {
        "Calle": "Beauchef",
        "Numero": 850,
        "Comuna": "Santiago",
        "Region": "Metropolitana",
        "Pais": "CHILE"
    }
}
```

4.3.6. Redis

Redis puede describirse como un "almacén de estructuras de datos en disco", y se utiliza comúnmente como base de datos y transmisor de mensajes; además, en conjunto con la librería *RQWorker*[cite here] de Python, se puede utilizar como una cola de ejecución de tareas. Éste fue el uso que se le dió en el servidor de imágenes, para ejecutar ciertas tareas de tiempo de ejecución más largo de manera diferida.

4.3.7. BeCity

BeCity es una aplicación para smartphones con el sistema operativo Android, la cual tiene como fin la recolección de datos de ciclistas (por ejemplo, recorridos populares en una ciudad o comuna). Además, ayuda a los usuarios a transitar de manera segura por la ciudad, ofreciendo rutas con ciclovías e información atingente.

4.3.8. SUR - Southern Urban Observatory

SUR es pretende ser un servicio de uso interno de INRIA Chile y NICLabs; una plataforma centralizada para todos los proyectos de ambos laboratorios. Pretende unificar de ésta manera muchos servicios que se encuentran replicados en los distintos proyectos (por ejemplo, manejo de usuarios) bajo una misma platforma, intengrando así de manera mucho más eficiente distintos proyectos y servicios otorgados por los laboratorios.

4.4. Situación Previa

4.4.1. BeCity

En agosto del 2015, BeCity como aplicación ya se encontraba en funcionamiento, siendo distribuída a través de la tienda de aplicaciones de Google, la "Play Store", y con su backend montado en el servidor de NICLabs. Sin embargo, si bien contaba con una API completa para la comunicación entre clientes y backend, ésta no seguía el estándar REST para servicios web, principalmente en lo que se refiere a la uniformidad de la interfaz y la transparencia de capas, además de no seguir el estándar HTTP (e.g. todas las solicitudes ocupaban el verbo GET independiente de si solicitaban o no un recurso del servidor). Dado estas condiciones, se había decidido reconstruirla desde cero siguiendo rigurosamente el modelo REST, para asegurar su futura escalabilidad y estabilidad.

4.4.2. SUR

Cuando el practicamente comenzó su trabajo en NICLabs, SUR como proyecto todavía no existía, por lo que todo el trabajo se desarrolló desde cero.

4.5. Descripción General del Trabajo

4.5.1. BeCity

El practicante diseñó e implementó parte de la nueva API, en específico el módulo de manejo de usuarios, siguiendo las restricciones del modelo REST para servicios web. El trabajo realizado se puede dividir en dos etapas:

1. *Diseño y Documentación*
   En esta etapa, el practicante estudió la API existente del módulo escogido para entender el funcionamiento interno de éste, y los métodos que presentaba su interfaz. Luego, se diseñó y documentó la nueva versión del módulo, con especial énfasis en la nueva interfaz.

2. *Implementación y Testeo*
   Luego de diseñado y documentado el módulo, se procedió a la implementación de las funcionalidades en código. Se utilizó el proceso de desarrollo "Test Driven Development", es decir, primero se escribían tests para funcionalidades deseadas, y luego se implementaban.

4.5.2. SUR

El trabajo realizado en SUR consistió en el desarrollo de un servicio de manejo de imágenes centralizado para todos los proyectos de la plataforma. El servicio debía básicamente almacenar y servir imágenes proporcionadas por los proyectos, por ejemplo para avatares de usuario. Se requería especial énfasis en la modularidad del servicio, de tal manera de que pudiese interactuar con cualquier otro proyecto que se adhiriera a la plataforma SUR en el futuro.

El trabajo consistió de dos etapas:

1. *Diseño del Servicio*
   Para comenzar, se determinaron los requerimientos del servicio; las funcionalidades deseadas y la manera de interactuar con éste.

2. *Implementación*
   Ya establecidos los requerimientos, se implementó el sistema deseado.

5. Trabajo Realizado

5.1. BeCity

5.2. SUR

Como se ha mencionado anteriormente, el proyecto SUR pretende ser una plataforma central para todos los proyectos de INRIA Chile y NICLabs - esto con el fin de centralizar varios servicios que actualmente se encuentran "duplicados" en todos los proyectos (por ejemplo, manejo de usuarios y de imágenes). Un servicio central como SUR permitiría que cierta información se pudiese compartir de manera transparente entre los proyectos y servicios, permitiedo por ejemplo que usuarios usen sus mismas credenciales de inicio de sesión, evitando la redundancia de tener que crear cuentas en cada uno de los servicios.
Bajo esta lógica fue que al practicante se le solicitó la implementación del servicio central de imágenes, el cual permitiese compartir de manera transparente y eficiente recursos de imágenes (por ejemplo, avatares de usuario) entre los proyectos.

5.2.1. Especificaciones

El servicio en cuestión debía seguir una serie de especificaciones generales que aseguraban su eficiencia, escalabilidad y su compatibilidad con los demás servicios de la plataforma.

1. *Independencia:* La implementación del servicio de imágenes debía ser independiente y agnóstica de la implementación del resto de los servicios. Es decir, si bien podía exister interdependencia entre el servicio de imágenes, y (por ejemplo) el servicio de autentificación, ésta no podía depender de la implementación de cada uno. Esto para asegurar que en el futuro, los servicios pudiesen actualizarse o cambiarse completamente sin perturbar el resto de los servicios en la plataforma.

2. *Adherencia al modelo REST* El servicio debía basarse en el paradigma cliente-servidor, y seguir las especificaciones del modelo REST; comunicación mediante verbos HTTP, no almacenar estados, etc.

3. *Simplicidad* El código debía ser simple y conciso, y no agregar demasiada complejidad al sistema.

4. *Instántaniedad* Finalmente, la respuesta del servicio a solicitudes por parte de clientes debiese ser expedita, y no causar esperas prolongadas.

Además, se establecieron una serie de requerimientos funcionales de los servicios que debía proveer el sistema:

1. Almacenamiento de imágenes. Por simplicidad, sólo se debía aceptar imágenes en el formato *Portable Network Graphics*, **.png**.

2. Descarga de imágenes del servidor, en tamaños y resoluciones no necesariamente iguales a las originales.

3. Eliminacion de imágenes almacenadas.

5.2.2. Diseño Arquitectural

En concordancia con las especificaciones y requerimientos anteriores, se llegó al diseño arquitectural detallado en esta sección.

El lenguaje escogido para la implementación fue nuevamente Python 2.7, principalmente por su amplio repertorio de librerías para desarrollo web, además de su simplicidad y elegancia sintáctica. Adicionalmente, es el lenguaje en el que están escritos todos los demás servicios de SUR, asegurando así una coherencia arquitectural interna de la plataforma.

Entre las librerías del lenguaje que fueron utilizadas, podemos destacar:

- Flask[7], la cual proporciona un framework básico para el desarrollo de aplicaciones web en Python. Por ejemplo, simplifica la ejecución de una aplicación Python como un servicio web asíncrono, y provee abstracciones para la implementación de endpoints (URL's) para interactuar con la aplicación.
- Pillow[10], librería de manejo de imágenes. Provee métodos de alteración de imágenes (transformaciones de tamaño, formato, etc).

Por otro lado, se descartó utilizar una base de datos para el servicio, ya que toda la información de las imágenes se puede manejar a través de *metadatos* y almacenarse en el sistema de archivos. Agregar un sistema de bases de datos aumentaría la complejidad del sistema de manera innecesaria.

Finalmente, para poder asegurar la rapidez del servicio, se decidió realizar ciertas solicitudes de manera diferida – esto es, no se ejecutan inmediatamente, sino que quedan en una cola de solicitudes y eventualmente se resuelven. Esto se implementó para solicitudes por parte de clientes que no requieren una respuesta inmediata del servidor y que además son intensas en términos de tiempo de ejecución (por ejemplo, la carga de una nueva imagen al servicio, ya que ésta operación requiere además un procesamiento posterior). Se efectuó mediante el uso de un servidor Redis en conjunto con el servicio de imágenes, el cual almacena y ejecuta las operaciones en la cola.

5.2.3. Implementación

La implementación del código se llevó a cabo siguiendo una estrategia iterativa basada en tests, implementando funcionalidades básicas y asegurándose de su correcto funcionamiento mediante tests antes de pasar a funcionalidades más complejas.

1. Carga de imágenes al servidor Ésta fue la primera funcionalidad en implementarse en el servidor, dada que puede considerarse una de las dos funcionalidades fundamentales del servicio (la otra sería la descarga de imágenes).

Funciona de la siguiente manera: el servidor expone públicamente una vista (en este caso, "/v1/images") que acepta solicitudes HTTP con el verbo POST. Esta solicitud debe incluir los siguientes HEADERS:

```
Content-Type: multipart/form-data
Content-MD5: <checksum MD5 del archivo>
```

Además, debe incluir el archivo a subir en formato binario en el cuerpo de la solicitud.

Al recibir la solicitud, el servidor primero ejecuta pruebas para verificar la existencia y validez de los datos proporcionados. En esta etapa el proceso puede arrojar tres variantes del mismo error HTTP "400 Bad Request", el cual se utiliza para notificar al cliente de que su solicitud no cumple con el formato requerido. En este caso, se lanza el error en caso de ausencia del archivo a cargar, o en caso de que la solicitud no incluya un HEADER con el hash MD5 de la imagen o que éste esté incorrecto.
Verificada la validez de la solicitud, el servidor pasa a almacenar y procesar la imagen (si la imagen ya existe en el sistema de archivos, el servidor simplemente retorna un mensaje HTTP "201 Created"). En esta etapa también se verifica que el archivo cargado sea efectivamente una imagen, y el servidor retorna nuevamente un "400 Bad Request" de no ser así.

El procesamiento de las imágenes se hace de manera diferida - al terminar las verificaciones mencionadas previamente, el servidor guarda la imagen en disco y agrega un procedimiento a la cola de procesamiento del servidor Redis (además de retornar "201 Created" junto con el hash de la imagen al cliente). La rutina diferida de procesamiento ejecutada por el servidor Redis es la siguiente:

```
1  def process_image(filename):
2      """
3      Deferred image handling routine.
4      Generates a set of images of various resolutions from an original image.
5      :param filename: ID of the original image.
6      :rtype: None
7      """
8      try:
9          o_img = Image.open(UPLOAD_FOLDER + filename + '/original.png', 'r')
10         o_img.save(UPLOAD_FOLDER + filename + '/original.png', format='png')
11         resolutions = app.config['STD_RESOLUTIONS']
12
13         for resolution in resolutions:
14             n_img = o_img.copy()
15             n_img.thumbnail(resolution, Image.ANTIALIAS)
16             n_img.save('{0}{1}/{2}.png'.format(UPLOAD_FOLDER, filename, resolution[0],
       resolution[1]), format='png')
17             n_img.close()
```

```
18
19          o_img.close()
20      except IOError:
21          shutil.rmtree(UPLOAD_FOLDER + filename)
```

Para evitar manejar los nombres de las imágenes (por ejemplo, por consideraciones de seguridad), éstas se identifican mediante su hash MD5: al recibir una imagen nueva, el servidor crea un directorio en la carpeta de imágenes cuyo nombre es el hash de la imagen, y luego almacena la imagen recibida bajo el nombre "original.png" dentro de dicho directorio. Luego, la rutina diferida "process_image()" se encarga de crear múltiples resoluciones de la misma imagen para su rápido acceso en el futuro, las cuales se almacenan bajo el mismo directorio mencionado anteriormente.

Finalmente, una imagen con hash MD5 `CB5DED429F491D8337FC58006468CF35` cargada correctamente al servidor quedaría almacenada de la siguiente manera:

```
UPLOAD_FOLDER/
    CB5DED429F491D8337FC58006468CF35/
        original.png
        48.png
        128.png
        256.png
        ...
```

Donde los archivos `48.png`, `128.png`, etc., corresponden a resoluciones específicas precalculadas para su rápido acceso en el futuro.

2. Descarga de imágenes La descarga de imágenes fue la segunda funcionalidad en implementarse, y funciona como se detalla a continuación.

Para descargar una imagen, el cliente debe conocer su ID (la cual corresponde al hash MD5 de la imagen). El servidor a su vez cuenta con una vista "/v1/images/<image_id>"; la notación <*image_id*> denota que esta parte del URL es variable - en específico, el cliente debe reemplazar <*image_id*> por la ID de la imagen que se solicita. Esta vista acepta los métodos GET y DELETE de HTTP (el verbo correspondiente a la solicitud de descarga es, evidentemente, el verbo GET), además de una serie de parámetros HTTP en el URL:

1. tamaño precalculado: El cliente puede solicitar un tamaño precalculado de la imagen usando el parámetro *"thumbnail=<thumb_size>"* (de no existir, se creará).

2. a escala: Utilizando el parámetro *"scale=<scale>"*, se puede solicitar la imagen a cierta escala respecto al original.

3. transformada: Finalmente, el cliente puede solicitar la transformación de la imagen mediante el uso de los parámetros *"width=<w>"* y *"height=<h>"* para especificar directamente el ancho y la altura de la imagen deseados, respectivamente.

Por ejemplo, un cliente que desee obtener la imagen `CB5DED429F491D8337FC58006468CF35` transformada a 1280x720 (sin mantener la relación de aspecto), debería efectuar la siguiente solicitud:

```
GET /v1/images/CB5DED429F491D8337FC58006468CF35?width=1280&height=720
```

Internamente, las solicitudes son manejadas por el servidor de la siguiente manera: Primero, se verifica la existencia de la imagen, y de no existir se retorna un código de error "404 Not Found" al cliente. Verificada ya la existencia de la imagen solicitada, se procede a servir la imagen de acuerdo a los parámetros solicitados por el cliente (si la solicitud no incluye parámetros, se retorna directamente la imagen original). De especial interés son aquellas solicitudes con el parámetro "thumbnail", ya que éstas son solicitudes para resoluciones populares y estándar de la imagen. Tienen el efecto de que, de no existir el tamaño solicitado, este es creado y almacenado por el servidor, de tal manera que futuras solicitudes para dicho tamaño sean más expeditas.

```
# Thumbnail requests are done first, because they are faster
# System now checks if there is a precomputed size stored and sends that instead.
try:
    thumb = int(request.args.get('thumbnail', 0))
except ValueError:
    raise BadRequest400(error="Width, Height and Thumbnail must be Integers, Scale must be Float")
except TypeError:
    raise BadRequest400(error="Width, Height and Thumbnail must be Integers, Scale must be Float")

filename = UPLOAD_FOLDER + image_id + '/' + str(thumb) + '.png'

if thumb != 0 and os.path.exists(filename):  # if it exists, send it
    img = Image.open(filename)
    try:
        return send_image(img)
    except IOError:
        app.logger.error('Error when trying to send image ' + image_id)
        raise IOError

elif thumb != 0:  # if it doesn't, create and then send it
    img = img.copy()
    img.thumbnail((thumb, thumb), Image.ANTIALIAS)
    img.save(filename)

    try:
        return send_image(img)
    except IOError:
        app.logger.error('Error when trying to send image ' + image_id)
        raise IOError
```

Extracto de código encargado de manejar solicitudes de tamaños precalculados.

Finalmente, se procesan las solicitudes con los parámetros de escala, ancho y alto, ya que estos requieren procesamiento en el lugar de la imagen, y demoran más.

6. Conclusiones

Referencias

[1] *NIC Chile Research Labs* - `http://niclabs.cl`

[2] *Felipe Lalanne* - `https://github.com/pipex`

[3] *PostgreSQL - "The world's most advanced open source database."*
`http://www.postgresql.org/`

[4] *The Python Programming Language* - `https://www.python.org/`

[5] *JetBrains Pycharm* - `https://www.jetbrains.com/pycharm/`

[6] *URI - Uniform Resource Identifier*: cadena de caracteres utilizada para identificar recursos específicos, siguiendo un esquema definido.
`https://en.wikipedia.org/wiki/Uniform_Resource_Identifier`

[7] *Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions.*
`http://flask.pocoo.org/`

[8] *Restless - A lightweight REST miniframework for Python.*
`https://restless.readthedocs.org/en/latest/`

[9] *SQLAlchemy - The Python SQL Toolkit and Object Relational Mapper*
`http://www.sqlalchemy.org/`

[10] *PILLOW - Python Imaging Library fork*
`https://pillow.readthedocs.org/en/3.1.x/`