Centro de Investigación en Matemáticas, A.C.

CIMAT

Caso de implementación de un servidor Restful Objects en GO

REPORTE TÉCNICO

Que para obtener el grado de

Maestro en Ingeniería de Software

P r e s e n t a

Octavio Reyes Pinedo

Director de Reporte Técnico

Alejandro García Fernández

Zacatecas, Zacatecas., 10 de septiembre de 2014

AGRADECIMIENTOS

Al Consejo Zacatecano de Ciencia y Tecnología (COZCyT) y al Centro de Investigación en Matemáticas A. C., Unidad Zacatecas (CIMAT), por haber financiado la realización de mis estudios a lo largo de la maestría en Ingeniería de Software.

A mis profesores José Guadalupe Hernández Reveles y Alejandro García Fernández, por su ayuda, tiempo y dedicación proporcionados para la realización de este proyecto.

A la Secretaría de la Función Pública, en particular al Director de Innovación Administrativa Manuel de Jesús Pérez Trejo, por darme la oportunidad de asistir a clases y asesorías para continuar con mis estudios.

A mi compañero de clases y amigo Omar Ibrahim Cabral Mier, por su ayuda técnica.

A mi familia, en especial a mi madre Ángela Reyes Pinedo, por el apoyo recibido durante estos años.

RESUMEN

Existen dos implementaciones de servidor de la arquitectura Restful Objects. Ambos servidores fueron desarrollados por el grupo oficial que define las especificaciones de Restful Objects. El problema es que actualmente no existe una implementación independiente de servidor Restful Objects, por lo que no se conoce si la especificación definida es replicable por un tercero. Actualmente los sistemas se desarrollan en capas, por lo que la complejidad aumenta, debido a que con más elementos, hay más problemas de comunicación entre las capas y las responsabilidades no están bien definidas, lo que provoca que los sistemas liberados no cumplan con las expectativas del usuario final. También el uso de API's es una buena opción para consumir servicios de terceros, estas API's pueden proporcionar servicios importantes, ejemplo de esto, son las API's creadas por empresas líderes como Amazon, Twitter y Google. Con la utilización de tecnologías que implementen Restful Objects, el desarrollo de sistemas y API's puede ser más sencillo, al reducir el proceso en un solo paso, al tener el modelado de objetos, se puede generar automáticamente la interfaz gráfica, esta simplificación de procesos puede servir de gran ayuda para los desarrolladores de sistemas transaccionales, en cualquier tipo de proyecto de manera general. En este reporte técnico se describe el proceso de creación de un nuevo servidor Restful Objects en el lenguaje GO, aunque éste no se logró implementar por completo, se explican recomendaciones para futuros proyectos.

PALABRAS CLAVE

Naked Objects
Restful Objects
Objetos Restful
Lenguaje GO
GOLANG
Reflection
Reflexión
Servicios REST
REST Services
GOREST
AROW

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

INTRODUCCIÓN

El desarrollo profesional de software es un proceso continuo, en el que se trata de alejarse de la complejidad accidental y acercarse a la complejidad esencial. En este sentido las bases de datos orientadas a objetos BDOO eliminan la complejidad accidental de la persistencia creada por las diferencias entre el modelo relacional y el modelo orientado a objetos, por ejemplo al evitar el mapeo manual de base de datos relacional a objetos y la codificación para ejecutar consultas SQL (Structured Query Language) de acuerdo al manejador utilizado. Con las BDOO solo se diseñan los objetos planos y el almacenamiento se realiza de manera automática. El patrón Naked Objects trata de hacer lo mismo para la interfaz de usuario, al diseñar los objetos planos, la interfaz de usuario puede ser creada automáticamente a partir de estos. De igual manera con la especificación de Restful Objects se busca generar la API (Application Programming Interface), mediante el diseño de los objetos planos para su utilización por parte de otro sistema.

Con los enfoques básicos de las BDOO, Naked Objects y Restful Objects, que se basan en un modelado de objetos planos para generar automáticamente interfaces y reducir la complejidad accidental, se pueden obtener las siguientes ventajas:

- Generación de código automática.- Al tener solamente un modelo de objetos correcto con persistencia, la interfaz de usuario y la API se puede generar automáticamente.

- Rápido prototipaje de aplicaciones.- Este proceso se enfoca en la interfaz de usuario, sin embargo, con este enfoque además de que la interfaz es generada automáticamente, la creación de prototipos puede suceder a nivel de lógica de negocios.
- Mejor modelado.- Cuando se pone todo el énfasis en el modelado de objetos, el usuario y programador pueden trabajar juntos para asegurarse que el modelo manifiesta fielmente el modelo mental del usuario. Lo que Pawson llama objetos conductualmente completos [1].
- Menor necesidad de depuración.- Debido a que las interfaces gráficas son generadas automáticamente hay menor probabilidad de errores humanos.
- Simplicidad.- Menos partes, menos cosas que romper. Cuando se tienen muchas capas o partes, cualquier modificación puede requerir cambios en varias secciones, lo que hace tedioso y complicado el proceso de actualización de funciones.
- Mantenibilidad.- La único que se necesita cambiar es la lógica de negocio y una vez que esté correcto, habrá menos dificultad para usarse.

Las Bases de Datos Orientadas a Objetos y los patrones Naked Objects y Restful Objects persiguen un mismo enfoque, modelar únicamente objetos planos y así generar automáticamente todos los componentes del sistema. Cuando se hable del enfoque a lo largo del desarrollo de este reporte se refiere a esto.

OBJETIVOS

Son muchos los beneficios que se obtienen de seguir el enfoque propuesto por Naked Objects de manera general y Restful Objects de manera particular para la generación de API's. Aquí surge la pregunta, si la propuesta es tan buena, ¿por qué este enfoque no es más común a pesar de que parece ser una ventaja competitiva para cualquier empresa dedicada al desarrollo software transaccional?

Con el fin de resolver el cuestionamiento, se decidió tratar de implementar un servidor propio para responder las siguientes preguntas:

1. ¿Es fácil de implementar Restful Objects en una nueva tecnología? Esto podría ayudar a desarrolladores en otros lenguajes como Ruby, PHP o GO para crear servidores en sus propias plataformas.
2. ¿Es la especificación de Restful Objects suficiente para crear un servidor propio?
3. En caso de que las respuestas a las dos cuestiones anteriores sean negativas, ¿Qué podría hacerse para hacer el desarrollo de un servidor Restful Objects más fácil?

RESTRICCIONES

Para responder las preguntas anteriores, se tomaron en cuenta las siguientes restricciones:

1. Tiempo. Se tuvieron solamente tres meses para desarrollar el servidor.
2. Equipo. Dos estudiantes/desarrolladores de tiempo completo.

3. Características. Solo se implementaría un servidor Restful Objects en este caso solo la API.
4. El servidor debería trabajar con el cliente genérico AROW o A Restful Object Workspace, que es conocido por ser compatible con las dos implementaciones de servidor actuales.

SECCIONES

En las siguientes secciones se exponen los resultados de esta investigación:

1. En los antecedentes se hace una profunda y amplia exploración de por qué Naked Objects y Restful Objects son importantes en el contexto del software y cómo está influyendo en su desarrollo.
2. La implementación propia de Restful Objects en GO. Se proporciona una descripción del proceso realizado para implementar el nuevo servidor.
3. La discusión y resultados. Se explica de manera subjetiva los resultados obtenidos al tratar de implementar el servidor propio en GO.
4. En las conclusiones y trabajo futuro. Se describe el resultado y sugerencias, para las personas que intenten hacer una nueva implementación en un futuro.
5. Anexos. Se expone a detalle, paso a paso cada una de las acciones tomadas para poner en práctica la creación e implementación del servidor en el lenguaje de programación GO.

1. ANTECEDENTES

En resumen, el reporte detalla la historia de tratar de generar una API de manera automática y los diferentes elementos que surgieron en el tiempo, para tratar de eliminar la complejidad accidental de la persistencia y presentación, generados por el modelo de tres capas a través de los Naked Objects y Restful Objects. En la Figura 1 se presentan las secciones del documento de manera gráfica.

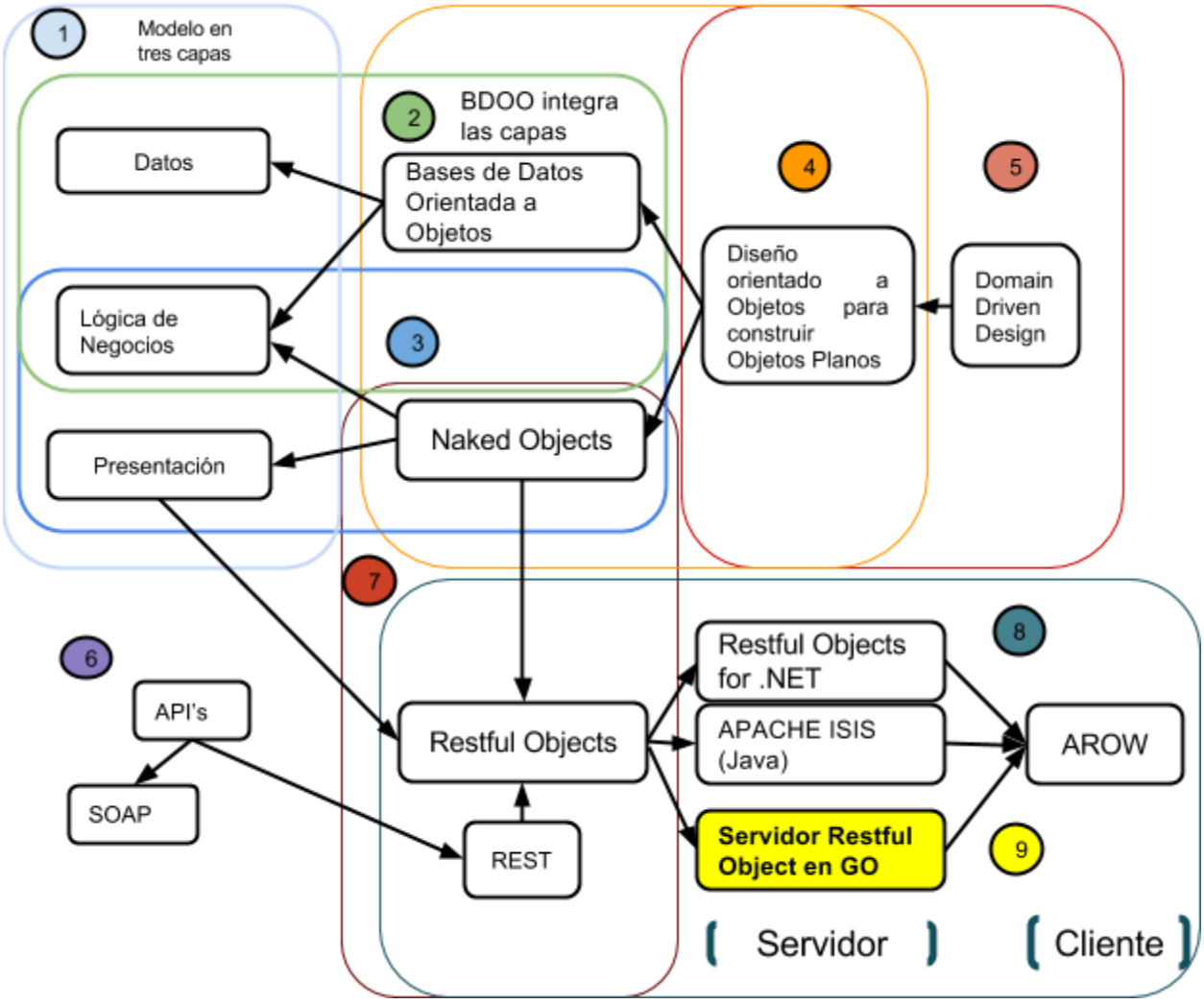

Figura 1. Conceptos del caso de implementación de servidor Restful Objects en GO.

El detalle de cada uno de los puntos representados en el diagrama se muestra a continuación.

1.1 EL PATRÓN MVC Y ARQUITECTURA DE TRES CAPAS

1.1.1 MVC

EL Modelo Vista Controlador, o MVC por sus siglas en inglés Model View Controller, es un patrón realmente antiguo, fue descrito por primera vez en 1979 por Tryve Reenskaug, que en ese entonces trabajaba en el lenguaje de programación Smalltalk en Xerox PARC. Después de más de 30 años, sigue siendo usado en la mayoría de los sitios web y frameworks. Prueba de esto es el uso en muchos frameworks de desarrollo como Ruby On Rails, Apple Cocoa, ASP .NET, Apache Struts, Cake PHP, CodeIgniter, JavaScriptMVC J2EE, entre otros [2].

Este patrón es ampliamente usado en aplicaciones World Wide Web, debido a los beneficios como reutilización de código y separación de responsabilidades, de hecho, está es la principal razón de su uso.

El argumento es que dada una clase de objeto de negocio será vista de diferentes formas, en diferentes plataformas, en diferentes contextos y usando diferentes representaciones visuales sería necesaria la incorporación de toda esta información en objetos abultados, duplicados y pesados si se usara una sola capa. Usando MVC, los objetos de modelo no tienen conocimiento de estos diferentes puntos de vista. Los objetos dedicados a la vista especifican que es lo que debe aparecer en cada vista, en

qué forma y saber cómo desplegarse visualmente. Los objetos controladores proveen un nexo entre los dos: poblar las vistas con atributos e invocar los métodos en los objetos en respuesta a eventos generados por el usuario [3].

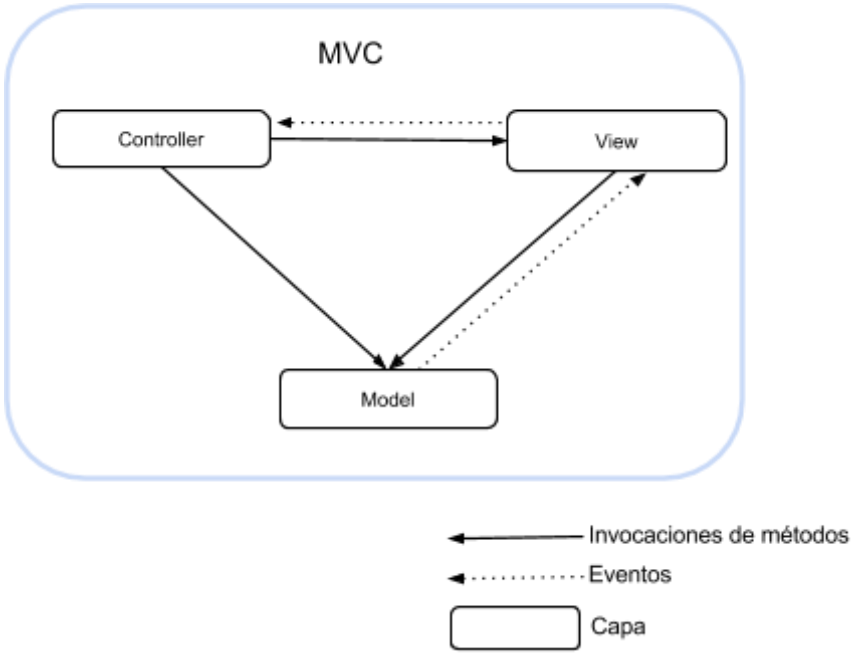

Figura 2. Modelo MVC.

En otras palabras, la capa Model se encarga de la lógica de negocio, con reglas, restricciones y comportamientos, la capa de View se basa en el aspecto visual o gráfico y la capa de Controller coordina las acciones realizadas entre View y Model (Figura 2).

Por otra parte, al modelado orientado a objetos le preocupa la forma de relacionar la estructura del software con la estructura de dominio de negocio del mundo real que el sistema está modelando. La motivación principal consiste en tener flexibilidad en el modelo, ya sea en respuesta a los cambios de requerimientos, o como respuesta a un problema en particular.

Si se fomenta el diseño de los sistemas de negocio desde el comportamiento de objetos completos, entonces se necesita superar la separación de procedimientos y datos en los diseños orientados a objetos. Para esto se requerirán nuevas técnicas y herramientas confiables [3]. Específicamente:

- Diseñar el modelo de objetos de manera interactiva con el usuario para lograr una eficiencia global en lugar de buscar eficiencias locales en la ejecución de tareas y comandos.
- En lugar de capturar los requisitos de un sistema como un conjunto de casos de uso y luego usarlos para identificar los objetos y sus responsabilidades, capturar el modelo de objetos que emerge como una forma concreta que los usuarios puedan identificar.

1.1.2 ARQUITECTURA TRES CAPAS

En la arquitectura de tres capas, el objetivo principal es la separación de responsabilidades. Para lograr esto se divide en tres capas [4]:

- La capa de presentación, es el nivel más alto que es la interfaz de usuario. La principal función es la traducción de tareas y resultados a algo que el usuario pueda entender.
- La capa de lógica, es la que coordina la aplicación, procesa los comandos, realiza las operaciones lógicas y realiza los cálculos.
- La capa de datos, aquí es donde la información es almacenada y recuperada desde una base de datos o sistema de archivos.

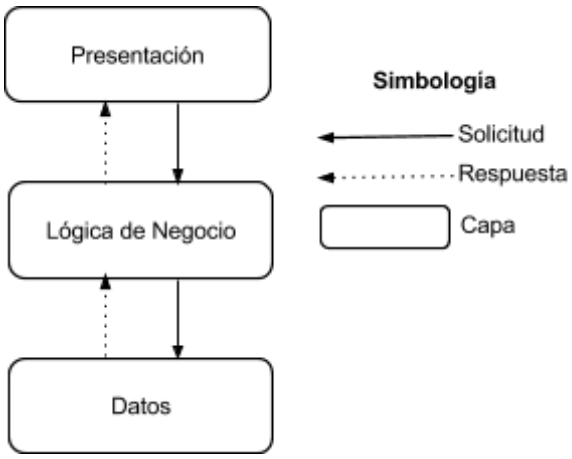

Figura 3. Arquitectura de Tres Capas.

1.1.3 RELACIÓN DEL PATRÓN MVC CON ARQUITECTURA DE TRES CAPAS

Tanto en el patrón MVC como en la arquitectura de tres capas, se busca la separación de responsabilidades mediante el uso de capas. Aunque una diferencia entre ambas es el tipo de comunicación. Una regla básica de la arquitectura 3 capas es que el capa de presentación nunca se comunica directamente con la capa de datos directamente. En este modelo toda la comunicación debe pasar a través de la capa intermedia (Lógica de negocio). Por lo tanto la comunicación es bidireccional (Figura 3). Mientras que en el patrón MVC, la comunicación es triangular, es decir, la vista manda las actualizaciones al controller, el controller actualiza el modelo y la vista obtiene actualizaciones directamente del modelo (Figura 2).

A pesar de las diferencias, no son mutuamente excluyentes, por ejemplo, el patrón MVC puede ser implementado en una arquitectura de tres capas. Desde la perspectiva de MVC, la vista y el controlador existen en la capa de presentación y el modelo abarca

las capas de negocio y de datos. Desde la perspectiva de tres capas esto significa que el modelo abarcará la capa de negocio y la capa de datos. El controlador y la vista existen en la capa de presentación [5]. Estas relaciones pueden verse en la Figura 4.

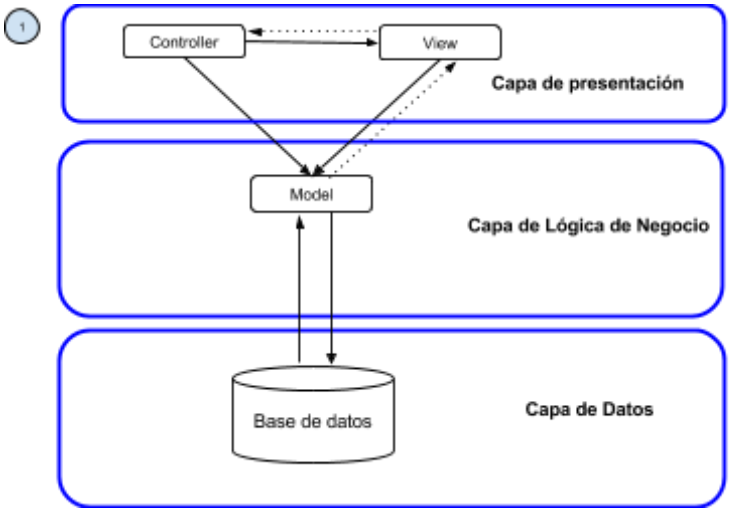

Figura 4. MVC Implementado en una arquitectura de tres capas.

A pesar de las ventajas de la división en capas, como la reutilización y escalabilidad, existen algunas desventajas importantes, por ejemplo, durante la fase de diseño, algunas responsabilidades no pueden quedar claras o correctamente separadas, provocando que operaciones de lógica de negocio aparezcan en las demás capas. Otro problema importante se encuentra en la facilidad de cambios, debido que, en muchas ocasiones un cambio en una capa puede requerir demasiadas modificaciones en las demás capas, generando gran inversión de tiempo y esfuerzo, debido a que el modelo tres capas provoca complejidad accidental, que se explica a continuación.

Como lo menciona Brooks [6] en su en ensayo ***No Silver Bullet***, no hay un desarrollo simple, ya sea en la tecnología o en la técnica de gestión, que por sí solo prometa incluso una mejora de orden de magnitud en productividad, fiabilidad y en simplicidad. Una de las premisas básicas del ensayo, hace referencia a dos tipos de complejidades, las accidentales y las esenciales. Las accidentales se relacionan con las restricciones que propiamente no son parte del problema, por ejemplo: particionar una base de datos, utilizar una base de datos relacional cuando el problema es un grafo, estos problemas pueden ser fácilmente corregidos con la selección de tecnología o para el caso, **el uso de tres capas agrega complejidad accidental al separar las responsabilidades**. La complejidad esencial por su parte se refiere en una entidad de software a la construcción de conceptos: conjuntos de datos, relaciones entre los elementos de datos, algoritmos e invocaciones de funciones. La esencia es abstracta. La construcción conceptual es la misma, en muchas representaciones diferentes. La parte más compleja de construir un software es la especificación, diseño y prueba del modelo conceptual, más que la labor de representar el modelo y mostrar la representación. Por lo tanto, parece que ha llegado el momento de abordar las partes esenciales de la tarea de software, los relacionados con la configuración de estructuras conceptuales abstractas de gran complejidad.

Brooks recomienda en resumen, centrarse en resolver la complejidad esencial de los problemas y en no agregar complejidad accidental, en la medida de lo posible. Reiterando, el modelo de tres capas y el modelo MVC al separar responsabilidades tiene ventajas importantes como la habilidad de poder ser presentado en múltiples plataformas encapsulando la lógica de negocio y la interacción con los datos. La

desventaja crítica de estos modelos es la complejidad accidental creada que provoca un bajo grado de mantenibilidad en el software desarrollado.

1.2. BASES DE DATOS ORIENTADAS A OBJETOS

El modelo de base de datos relacional pone énfasis en el contenido. Aunque este es el tipo de base de datos más ampliamente utilizado hasta la fecha tiene sus limitantes. De entrada, la estructura es rígida y falta de soporte para nuevos tipos de datos tales como gráficos, información en 2D o 3D, etc.

En la década de 1980 con la llegada de las metodologías y lenguajes orientados a objetos, se pudo dar la integración de capacidades de base de datos orientadas a objetos con lenguajes de programación proporcionando un entorno de programación unificado. Esto condujo al desarrollo de OODB, Object Oriented Databases - BDOO o Bases de Datos Orientadas a Objetos, donde los objetos son almacenados en las bases de datos en lugar de datos tales como números enteros, cadenas o números reales [7].

En una BDOO la creación y manipulación de los objetos ocurre en el lenguaje de programación del sistema y la persistencia a disco es transparente. Esto es contrastante con una base de datos que utiliza el SQL embebido o una llamada de interfaz como ODBC o JDBC. Usar un producto de base de datos de objetos significa tener un alto rendimiento y menos código que escribir. Con esta persistencia

transparente, la manipulación y persistencia de objetos se lleva a cabo directamente por el lenguaje de programación, esto se logra con el manejo inteligente de la caché [8] (Figura 5). De esta manera se integran a la capa de lógica de negocio y la capa de datos.

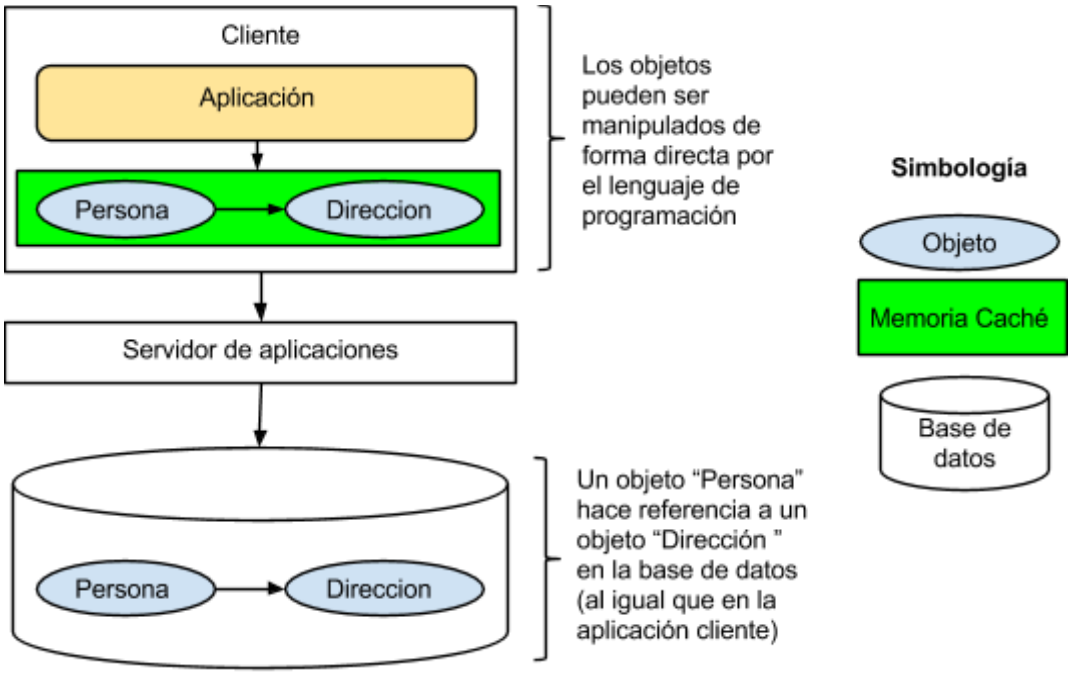

Figura 5. Persistencia transparente en base de datos orientada a objetos.

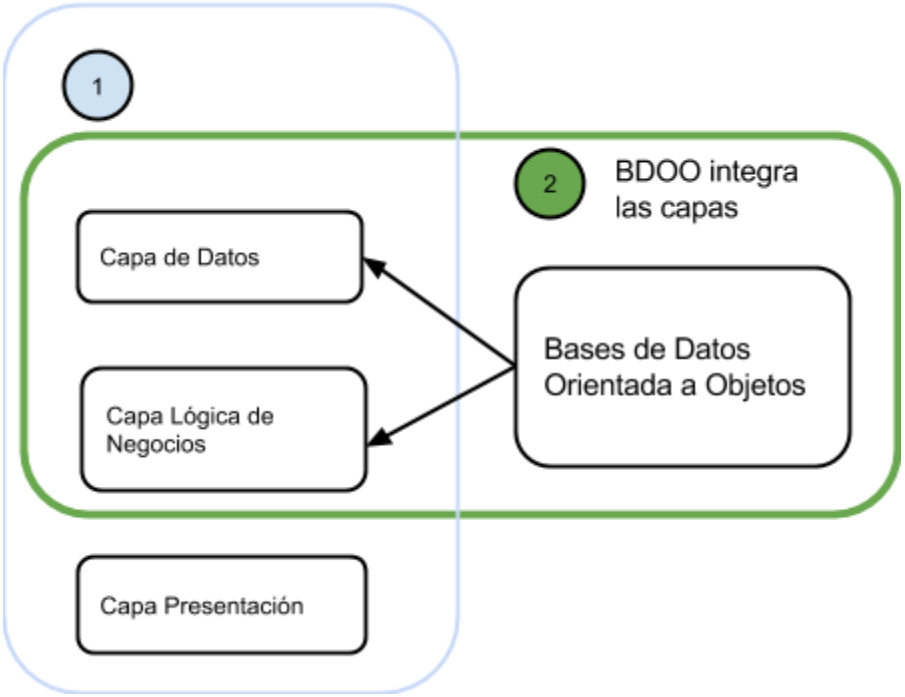

Figura 6. Integración de capas con las BDOO.

Debido a que cuando al actualizar cualquier valor de un objeto dentro de la aplicación, el cambio es realizado de manera automática y transparente en la base de datos, por lo tanto se puede unir o integrar la capa de datos y la capa de lógica de negocio en una sola estructura (Figura 6).

Entre las principales ventajas de las BDOO se encuentran [9]:

- Definir la información como objetos, mejora la comunicación entre usuarios, diseñadores y analistas.
- El uso del mismo lenguaje de programación para definir los datos y los métodos facilita el desarrollo de aplicaciones.
- Define nuevos tipos de datos a partir de los existentes.

- El modelo soporta el acceso a la navegabilidad, es decir un registro a la vez, ya que se pueden recorrer los objetos por medio de colecciones.

Sin embargo, también hay desventajas como las siguientes [9]:

- No se cuenta con un modelo universal aceptado para la manipulación de objetos, por lo que cada manejador utiliza sus propias reglas.
- El modelo puede resultar complejo y también puede resultar difícil la actualización de dicho modelo.
- No se cuenta con restricciones declarativas porque depende de los métodos definidos en los objetos.
- La operación de filtrado o búsqueda (SELECT de las relaciones) no se encuentra en todas las BDOO, por lo que depende del manejador de base de datos utilizado.
- A diferencia de las bases de datos relacionales, no existen foreign keys o llaves foráneas, por lo que las relaciones son directas entre los objetos

En resumen, el enfoque orientado a objetos ofrece la oportunidad de cumplir con los requerimientos del negocio, sin estar limitado a los tipos de datos y los lenguajes de consulta disponibles en los gestores de datos relacionales como SQL. Además de que la persistencia transparente manejada directamente en la base de datos sin la necesidad de capa intermedia, con lo que hay un gran ahorro de tiempo en codificación. Así pues se reduce la complejidad accidental.

1.3. NAKED OBJECTS

Naked Objects es un framework de código abierto con el cual automáticamente se despliegan las clases de dominio en una interfaz de usuario en el paradigma orientado a objetos. La interfaz puede ser una aplicación de escritorio o una aplicación web, esto sin codificación extra.

Debido a que Naked Objects es responsable de la capa de presentación, los desarrolladores se pueden enfocar exclusivamente al modelado del problema. Si se hace un cambio en alguna clase del modelo, cuando la aplicación se ejecuta, estos cambios son realizados de manera automática [10].

Naked Objects es un patrón arquitectural y también es un framework. El patrón fue originalmente concebido por Richard Pawson [10], como una forma de relacionar los stakeholders del negocio y expertos en desarrollo en aplicaciones más expresivas guiadas por dominio.

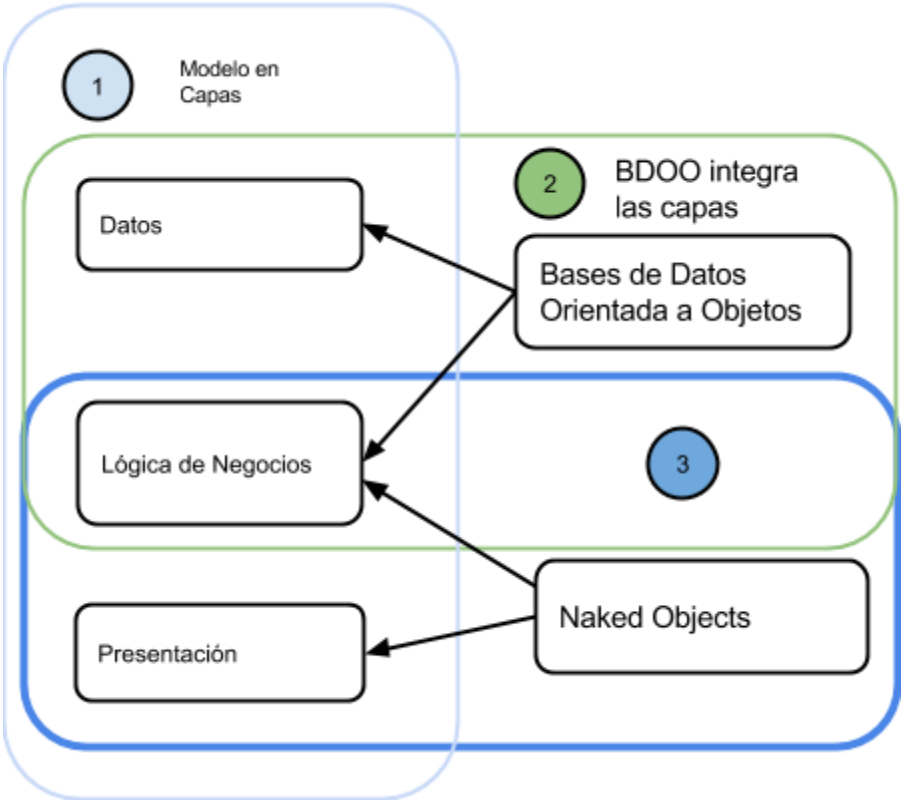

Figura 7. Capas que une Naked Objects.

El framework es una implementación del patrón que ayuda a desarrollar prototipos rápidos y desplegar aplicaciones guiadas por el modelo de dominio. Un modelo de dominio es un modelo conceptual de todos los temas relacionados con un problema. La rapidez resulta de poder programar una aplicación sin gastar demasiado tiempo en escribir código en la interfaz de usuario o en código para administrar la persistencia. De esta manera que se unen las capas tradicionales de lógica de negocios y presentación en un solo componente en base al patrón Naked Objects (Figura 7).

Entre los beneficios más importantes del uso de Naked Objects se encuentran:

- Mayor productividad en el desarrollo, debido a que no se requiere escribir el código para crear la interfaz de usuario.
- Mayor facilidad con la que una solicitud puede modificarse para adaptarse a los futuros cambios en los requerimientos del negocio. En parte esto se debe a la reducción en el número de capas desarrolladas que deben mantenerse en sincronización.
- El mantenimiento de los sistemas es más fácil, esto se logra mediante utilización forzada de objetos conductualmente completos.
- Fácil análisis y captura de requerimientos de negocio porque los objetos constituyen un lenguaje común entre usuarios y desarrolladores.
- Hace más sencillo y ágil el proceso de prototipado de aplicaciones.
- Mejoras en el soporte para el desarrollo basado en pruebas.

Entre las desventajas del uso de este patrón se encuentran las siguientes:

- La generación de interfaces de usuario genéricas puede ser una desventaja para algunas personas, debido a que a muchos desarrolladores les puede parecer mejor desarrollar su propia interfaz personalizada.
- Es útil principalmente para los usuarios “expertos”, que conocen el modelo de negocio y los procesos. Este enfoque no será tan útil para los usuarios que necesitan scripting pesado y llevarlos de la mano para terminar el proceso con éxito.
- El desarrollo de un framework que cumpla totalmente con el patrón Naked Objects es complicado y tardado, debido a la cantidad de características que

debe tener para trabajar con cualquier tipo de modelo de datos, por lo que se necesitaría de un framework altamente confiable.

Actualmente el patrón Naked Objects no es popular, aunque la idea es sólida. Por las ventajas mencionadas vale la pena que sea más conocido y que aumenten las contribuciones al proyecto. Este es uno de los objetivos del presente reporte.

1.4. OBJETOS PLANOS POJO Y POCO

POJO significa ***Plain Old Java Object***o en español **Objeto Java Antiguo Plano**. Estas siglas fueron creadas por Martin Fowler, Rebecca Parsons y Josh MacKenzie en septiembre de 2000. Un POJO es una instancia de una clase que no extiende ni implementa ningún framework en particular. Si se define una clase con atributos y unas cuantas operaciones, se tiene un simple y modesto POJO [11].

Hibernate es un framework de Java para **ORM,** *Object Relational Mapping* o Mapeo Objeto - Relacional, esto quiere decir que se puede establecer una relación entre una tabla de base de datos relacional y una clase de la aplicación, a esto se le conoce como mapeo. Hibernate funciona mejor si se siguen las reglas de programación POJO [12].

POCO o *Plain Old CLR Object* o en español **Objeto CLR Antiguo Plano**, es un juego de palabras que hacen referencia a POJO de la plataforma Java y es utilizado por los desarrolladores del framework .NET. En .NET POCO es la capacidad de agregar y

utilizar clases propias de datos personalizadas con el modelo de datos [13]. ADO .NET Entity Framework o EF conforma el framework .NET basado en POCO.

En el patrón Naked Objects, la base en el modelado es el diseño de los objetos de dominio, estos son los objetos planos. En cualquiera de los dos frameworks mencionados, tanto Hibernate como EF, al realizar cualquier cambio en un objeto dentro de la aplicación, este cambio se verá reflejado en la base de datos de manera transparente. Los objetos planos son el punto en común de Naked Objects y para las Bases de Datos Orientadas a Objetos (Figura 8). Y por tanto el punto clave que permite la generación automática de las tres capas de una aplicación.

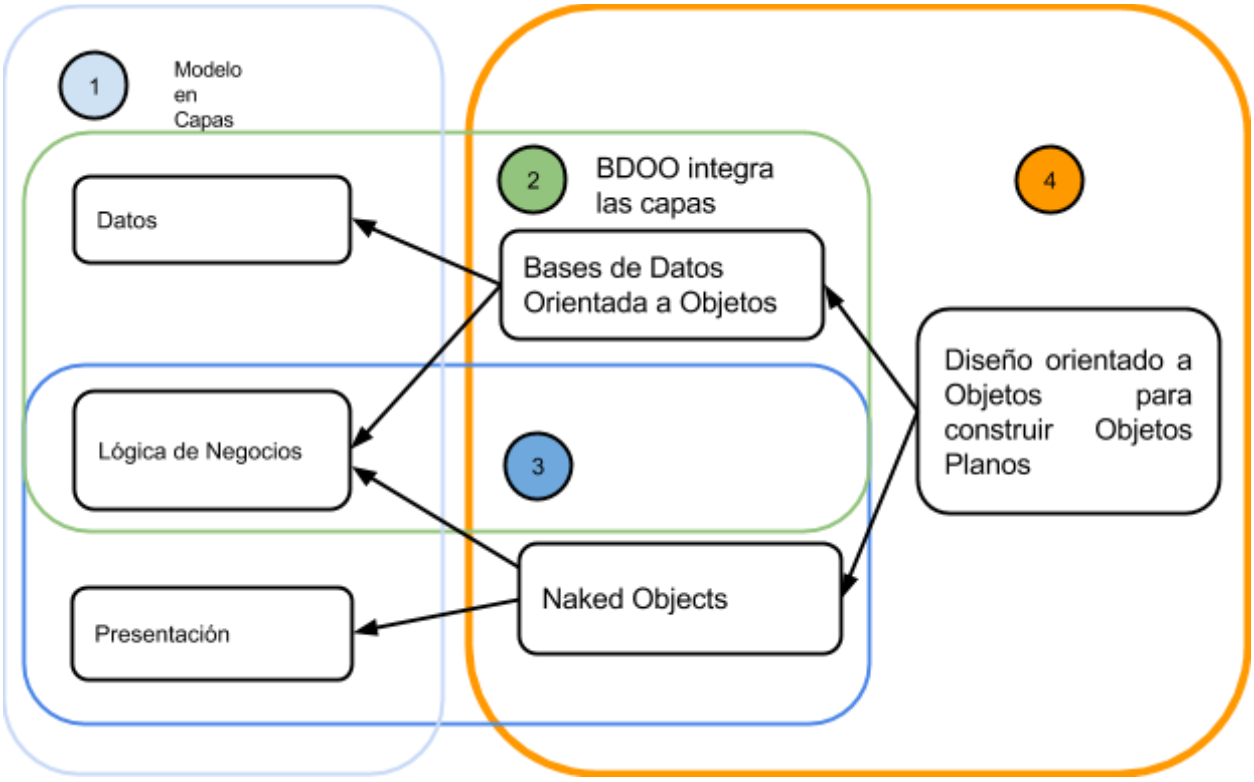

Figura 8. Objetos Planos y su relación con Naked Objects y BDOO.

Es importante que los objetos planos modelen con claridad los objetos de dominio del negocio. El modelo debe representar las relaciones, validaciones y/o condiciones. Para lograrlo de una manera estándar, surge la técnica Domain Driven Design que se explica a continuación.

1.5. DOMAIN DRIVEN DESIGN (DDD)

La metodología de *Domain-Driven Design* o Diseño Guiado por el Dominio sirve para modelar un sistema teniendo como base y principal elemento el problema que se desea solucionar de una forma macro, más allá de los datos. Es un enfoque para el desarrollo de software con necesidades complejas, mediante una profunda conexión entre la implementación, los conceptos del modelo y núcleo del negocio [10].

Uno de sus principales beneficios es el desarrollo de un lenguaje ubicuo común entre los expertos de negocio y los desarrolladores de software. Además de la sujeción del código al dominio del negocio.

Un problema común al practicar DDD es que es difícil involucrar a los expertos del negocio. Es casi imposible que estos expertos aprendan a modelar diagramas en UML, *Unified Modeling Language* o Lenguaje Unificado de Modelado, es mucho más fácil involucrarlos a través de prototipos rápidos, donde puedan interactuar con los resultados. Pero la mayoría de los prototipos se concentran en la capa de presentación, a expensas del pensamiento abstracto o del modelo subyacente.

Crear sistemas o aplicaciones empresariales es difícil. Se debe conocer una gran cantidad de tecnologías: frameworks para manejar la presentación, APIs para la persistencia, tecnologías para autenticación, entre otras. Ninguno de estos es importante por sí mismo para los usuarios del negocio.

Domain-Driven Design es una propuesta para la construcción de software de aplicación que se centra en la parte que importa en las aplicaciones empresariales: el núcleo del dominio del negocio. En lugar de poner todo el esfuerzo en cuestiones técnicas o de presentación, se trabaja para identificar los conceptos clave que desea que la aplicación maneje. Por lo que se puede utilizar un equipo formado por desarrolladores y expertos del dominio de negocio. Juntos trabajan para asegurarse de que cada uno entiende al otro mediante el uso de la base común del propio dominio.

El reto es como traer a la vida el modelo de dominio para que sea revisado, verificado y refinado. Aquí es donde es útil el patrón Naked Objects, que se aplica cuando se diseñan los objetos del modelo de dominio como POJOs o POCOs, para generar de manera automática el modelado en una interfaz de usuario (Figura 9).

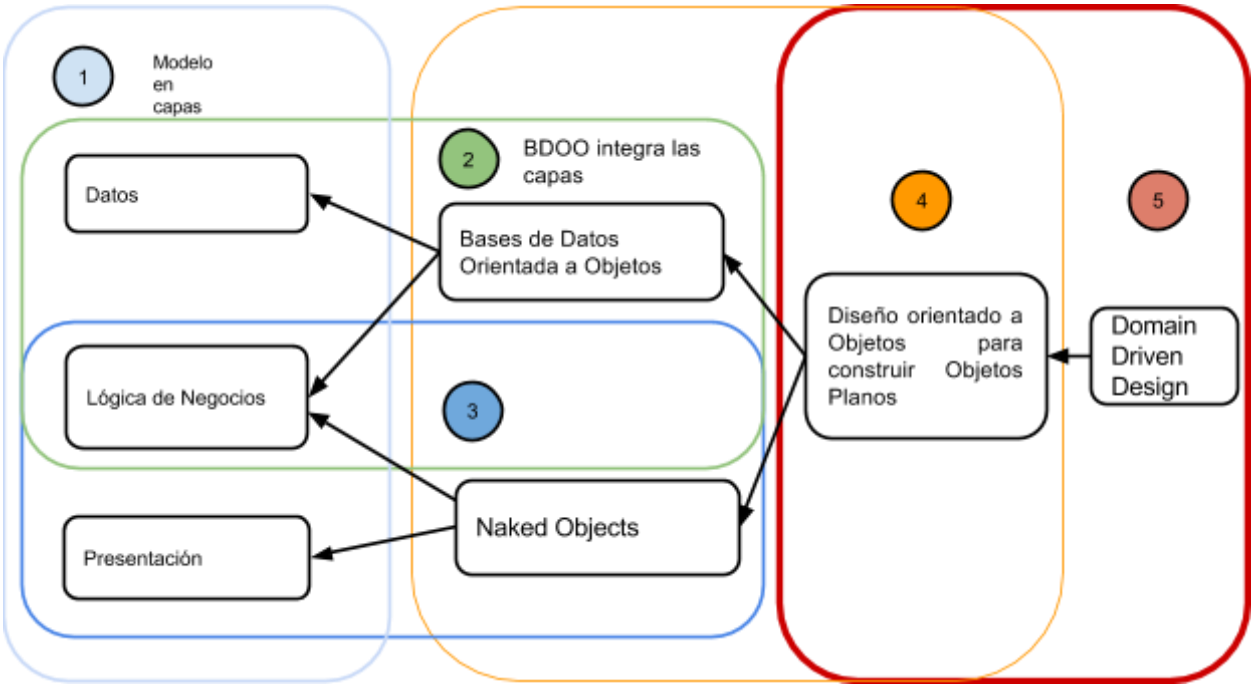

Figura 9. Domain Driven Design relacionado con los Objetos Planos.

Los dos principales elementos de DDD son el lenguaje ubicuo y el MDD, *Model Driven Design* o Diseño Guiado por el Dominio.

El lenguaje ubicuo consiste en conseguir que el equipo completo, expertos del dominio y desarrolladores, se comuniquen transparentemente usando los conceptos del dominio y utilicen los mismos términos al hablar, en diagramas en forma escrita, en presentaciones y hasta en código.

Con el MDD existe una forma directa y literal de representar el modelo en términos de software. Éste debe cumplir ciertos requerimientos. De entrada, formar el lenguaje ubicuo y ser representable en código. Cambiar el código significa cambiar el modelo y viceversa, refinar el modelo requiere un cambio en el código.

El uso de lenguajes orientados a objetos facilita la representación, ya que la orientación a objetos está basada en el tratamiento de objetos reales en el mundo [14]. Se pueden expresar conceptos de dominio usando clases y las relaciones entre ellas usando asociaciones.

DDD a través de sus dos elementos permite un modelo más acercado a la realidad, con una amplia colaboración de expertos de negocio y desarrolladores. Los objetos planos definidos por DDD y gracias a Naked Objects y las Bases de Datos Orientadas a Objetos se puede generar el sistema en un solo movimiento. Como resultado se elimina prácticamente toda la complejidad accidental creada por el modelo de tres capas sin perder sus beneficios, en teoría.

1.6. API'S EN LA ACTUALIDAD

En prácticamente en todos los ámbitos existen dependencias externas de un sistema para realizar funciones que el propio sistema no es capaz de realizar o por lo menos no por sí mismo. Ejemplo de ello es abrir una caja de seguridad bancaria. Del mismo modo, prácticamente todo el software existente tiene que hacer solicitudes a otros sistemas de software para realizar ciertas operaciones. Para lograr esto, el programa solicitante usa un conjunto de peticiones estandarizadas llamadas API's que se han definido para que el sistema de software cliente pueda utilizar las funciones disponibles que desee emplear. Casi todas las aplicaciones dependen de las API's del sistema

operativo subyacente para ejecutar funciones básicas como el acceso al sistema de archivos.

Los desarrolladores corporativos deben considerar la inclusión de API's en sus sistemas, sobre todo si esperan que las aplicaciones perduren e interactúen con otras. Conforme pasa el tiempo, la probabilidad de que una aplicación tenga que aprovechar los servicios de otra aumenta. La previsión de incluir API's ahorra tiempo al no tener que buscar y revisar el código fuente [15].

Las dos principales tecnologías para la creación de API's son REST o *Representational State Transfer*) y SOAP o *Simple Object Access Protocol* (Figura 10). REST presenta varias ventajas sobre SOAP, como se menciona a continuación:

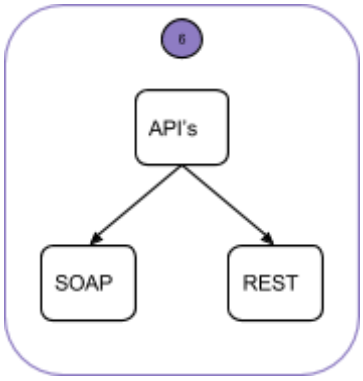

Figura 10. Principales tecnologías para la realización de API's.

- REST es un tipo de arquitectura de desarrollo web que se apoya totalmente en el protocolo HTTP.
- REST puede considerarse un framework para construir aplicaciones web que respeta HTTP y por tanto permite crear servicios y aplicaciones que pueden ser usadas por cualquier dispositivo o cliente que soporte HTTP, por lo que es más

simple y convencional que otras alternativas que se han usado en los últimos diez años como SOAP y XML-RPC. REST el tipo de arquitectura más natural y estándar para crear API's, para servicios orientados a Internet.

- Con REST es mucho más simple la creación de clientes, el desarrollo de API's y la documentación es fácil de entender.
- REST permite diferentes formatos como JSON, a diferencia de SOAP que solo permite XML.
- Otras ventajas son el rendimiento y escalabilidad [16].

Existe un gran número de aplicaciones REST en Internet, prácticamente cualquier servicio accesible mediante una petición HTTP GET. Entre las implementaciones más importantes que ofrecen API's públicas basadas en REST, se encuentran: Amazon, Facebook, Twitter, Yahoo! y Google Apps [17].

Las API's son muy importantes hoy en día, sin embargo algunas prácticas malas que afectan el buen uso de estas son: el mantenerlas en secreto, cambiarlas frecuentemente y mala o nula documentación.

Los desarrolladores de aplicaciones y los proveedores deben asegurarse de que sus API's sean comprensibles para su uso futuro. Una API es inútil a menos que se haya documentado, sin embargo, a pesar de ello, algunas empresas han dejado sus API's sin documentación alguna. Estás API's por tanto no son utilizadas o tienen un riesgo muy alto ser ignoradas [15].

En resumen, por su propia naturaleza los servicios web actuales requieren la construcción y uso de API's para su interacción con otros sistemas de software. El modelado de objetos planos ya genera la interfaz y la propia lógica de negocio a través del patrón de Naked Objects. A través de la especificación Restful Objects, el mismo modelo se utiliza para generar también las API's con tecnología REST. Esto lo explicamos a continuación.

1.7. RESTFUL OBJECTS

Restful Objects es una especificación pública y un patrón para la construcción de un API bajo la tecnología REST. Ya que se cumple el estándar de REST, es posible tener acceso a la funcionalidad entera del API solamente siguiendo los links o ligas de un recurso. Restful Objects provee además acceso al comportamiento completo de los objetos, es decir incluye a los métodos.

Restful Objects es la elección directa para proveer una API basada en REST para desarrolladores que utilizan DDD y por tanto ya tienen o están desarrollando un modelo de objetos de dominio.

Existen beneficios adicionales al ahorro de esfuerzo como la validación y verificación, ya que la capa lógica puede ser ejecutada haciendo pruebas de unidad rápidas y usando poca memoria. También se facilita la documentación debido a que es guiada por el modelo de objeto de dominio, por lo que puede ser documentada con técnicas establecidas como UML [18].

Existen dos frameworks de código abierto que implementan Restful Objects, *Naked Objects para .NET* y *Apache Isis*. Estos frameworks toman un modelo de objeto de dominio, utilizando POCOs o POJOs respectivamente, para así crear una API REST sin necesidad de escribir más código. Ambos frameworks implementan el patrón Naked Objects, por lo que generan la interfaz de usuario orientada a objetos automáticamente.

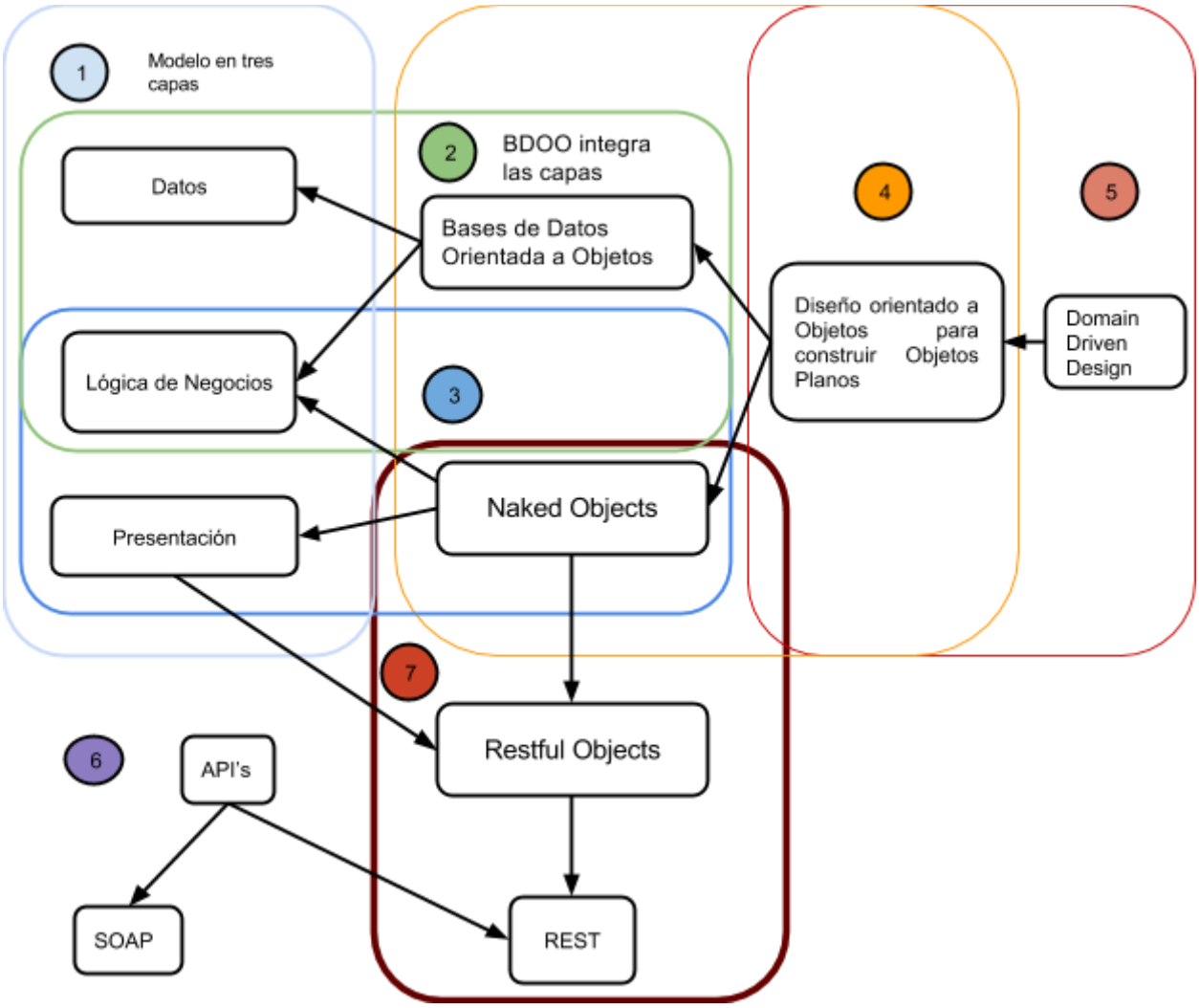

Figura 11. Relación Restful Objects con Naked Objects y REST.

De la misma manera siguiendo el patrón Naked Objects se generan interfaces gráficas, la especificación de Restful Objects extiende el modelo generado el API basada en REST (Figura 11). De esta manera se satisface la necesidad actual de contar con API's para la interacción con otros sistemas. Actualmente existen solo dos servidores que cumplen con las especificaciones existentes de Restful Objects y un cliente genérico. A continuación se detalla la tecnología utilizada y su implementación.

1.8. IMPLEMENTACIONES CLIENTE Y SERVIDOR RESTFUL OBJECTS

La especificación de Restful objects tiene dos tipos de implementaciones, la referente al lado cliente y la correspondiente al lado del servidor. Del lado del servidor existen dos frameworks de código abierto: 1) **Restful Objects for .NET** version 1.0 y 2) **Apache Isis** versión 0.55.

Restful Objects for .NET es una implementación completa de la especificación, en versión beta, sólo porque se hace uso de la API del framework de Microsoft Web parte de ASP .NET MVC4 [18].

Restful Objects Apache ISIS se ejecuta en la plataforma Java, este framework no está terminado, pero en la actualidad implementa una temprana especificación, por lo que necesita más pruebas antes de su liberación [18].

Con estos frameworks es posible tomar un modelo de objetos de dominio, escrito como POCOs y POJOs respectivamente y crear una API completa de REST que cumpla con la especificación de Restful Objects, sin escribir más código (Figura 12).

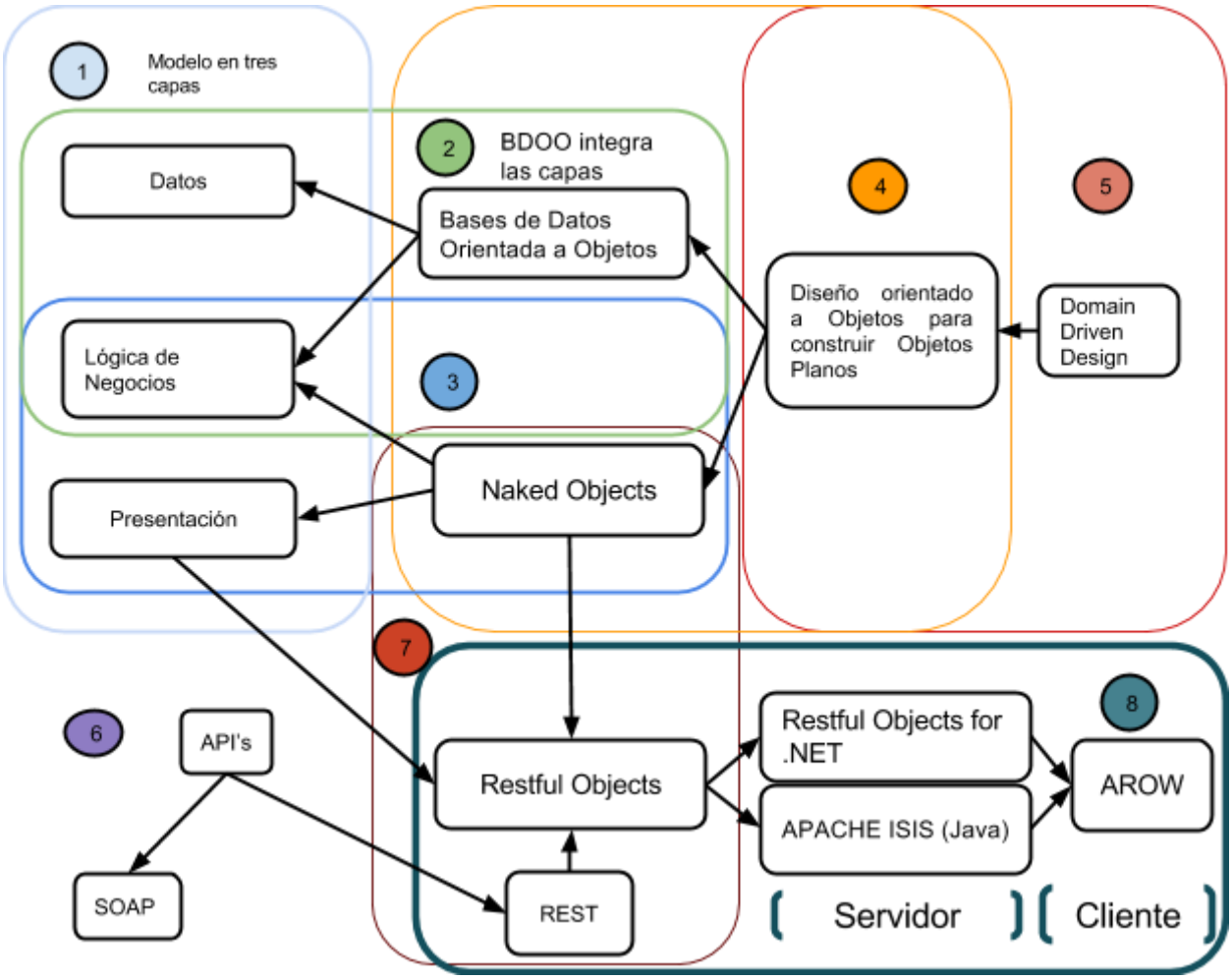

Figura 12. Implementaciones servidor-cliente Restful Objects

Del lado del cliente, se encuentra en desarrollo el proyecto **AROW** o *A Restful Objects Workspace*, que es una aplicación en una página web (Figura 12). Es un código abierto que usa Javascript. En éste, el Javascript hace llamadas a la API de Restful Objects en un servidor y renderiza los resultados como HTML en el navegador,

mediante una interfaz amigable. Esta aplicación continúa en desarrollo por Adam Howard. Para el proceso de experimentación, se eligió AROW debido a que por el momento, es el proyecto que tiene más reconocimiento, en cuanto al cumplimiento de reglas de Restful Objects. El código fuente del cliente AROW se encuentra en GitHub en la dirección https://github.com/adamhoward/AROW.

2. IMPLEMENTACIÓN RESTFUL OBJECTS EN GO

A la fecha de inicio del presente reporte, no existía una implementación de un servidor independiente de Restful Objects, por tanto, no se tenía la certeza de la posibilidad de ser replicada por un tercero.

Para comprobar lo anterior se propuso seguir la especificación de Restful Objects para crear un servidor en un lenguaje de programación diferente a Java y .Net. Para validar el servidor se utilizó el único cliente genérico existente llamado AROW, esta aplicación tiene muy poco código en comparación con la complejidad y cantidad de datos que envían y administran los servidores Restful Objects. El lenguaje de programación elegido es GO (Figura 13). GO está diseñado para la programación Web concurrente, tiene su propio recolector de basura, además de la capacidad de usar reflexión lo que lo convierte en un lenguaje muy potente. GO es muy sencillo de aprender debido a su sintaxis clara y concisa similar al lenguaje C.

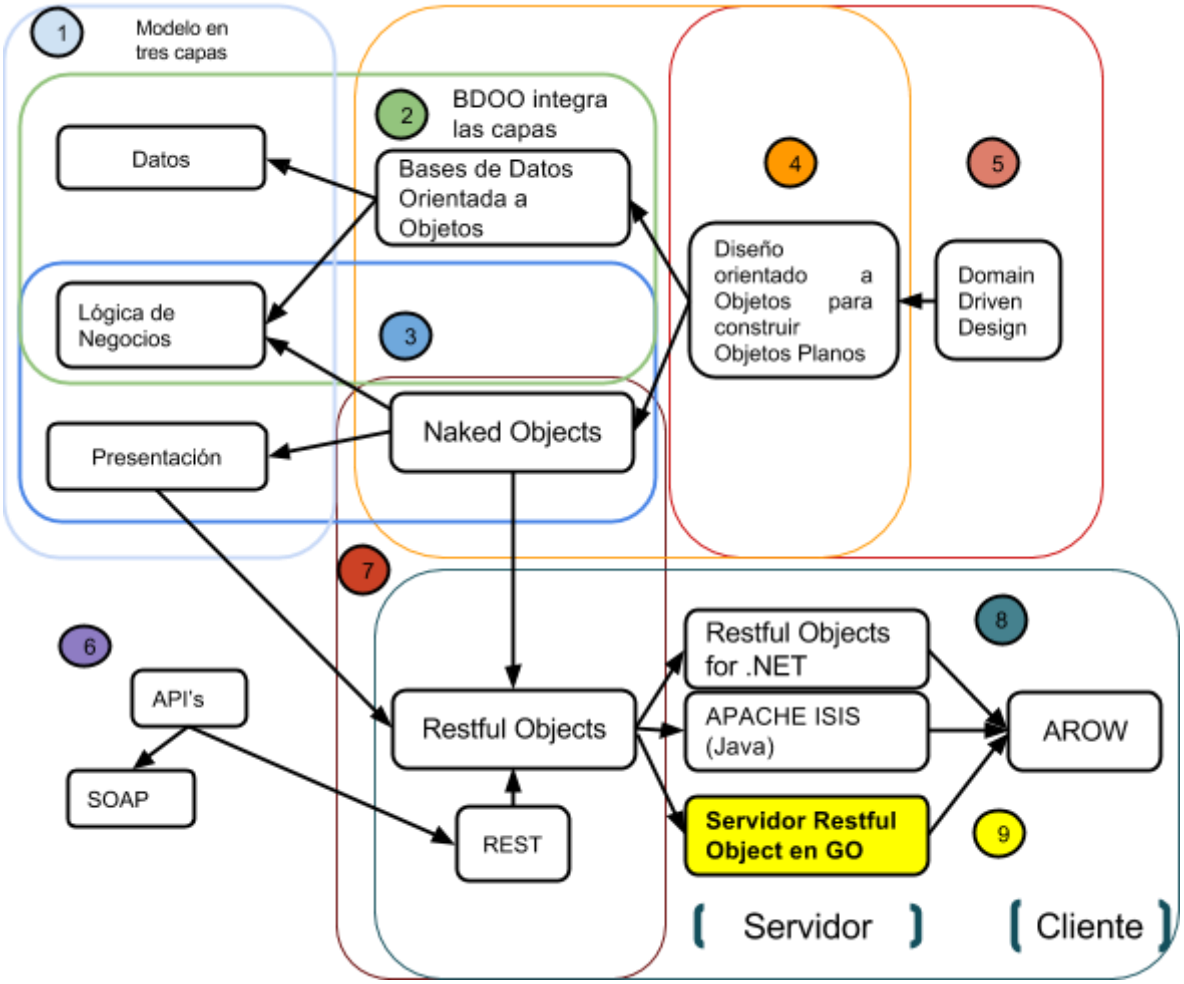

Figura 13. Servidor Restful Objects en GO.

Por su parte, AROW fue creado para aceptar las dos implementaciones de servidor que cumplen con el estándar Restful Objects, estas son la versión 1.0 (Restful Objects for .Net) y la versión 0.55 (Apache ISIS).No está probada su integración con algún otro servidor, o por lo menos no hay documentación de pruebas. Aunque existen clientes genéricos, se seleccionó el cliente AROW porque por el momento, es el más maduro. Una ventaja adicional es que además es atractivo visualmente.

Se consideró seleccionar otros lenguajes de programación como PHP y Ruby, pero se seleccionó GO ya que ofrece un mejor rendimiento. Un caso práctico que demostró este atributo de calidad, ocurrió en la empresa Iron.io, a principios del año 2013. Su aplicación IronWorker pasó de usar 30 servidores a solamente 2. Después de que se puso en marcha su nueva versión de GO, hubo una reducción a dos servidores, realmente se necesitaba solo un servidor pero se usaron dos por cuestiones de redundancia. La utilización del CPU fue menor al 5%, y el proceso entero comenzó a usar solo unos cientos de KB's de memoria al arrancar, contra los 50 MB que utilizaba anteriormente en su versión antigua desarrollada en Ruby [19]. Esto se debe a que en GO, la concurrencia es una prioridad, el estándar del núcleo de la librería tiene casi todo lo necesario para construir servicios, además de que se compila de manera rápida.

GOREST es un framework estilo REST para el lenguaje de programación GO, además de que va un paso más allá, mediante la adición de una capa que hace las tareas tediosas automatizadas [20], por lo que fue una tecnología muy útil en la realización de este proyecto.

El proceso de pruebas se dividió en dos fases principales:

- La creación de los servicios REST con el lenguaje de programación GO, utilizando reflection o reflexión, que es un paradigma de programación, que permite conocer la forma en la que están construidas las clases del código que se está ejecutando (ingeniería inversa), con el objetivo de crear un servidor que cumpliera con la especificación Restful Objects.

- El proceso de experimentación y pruebas de integración del servidor creado en GO con el cliente AROW.

Cada fase fue separada en tareas principales que se describen a continuación (Figura 14).

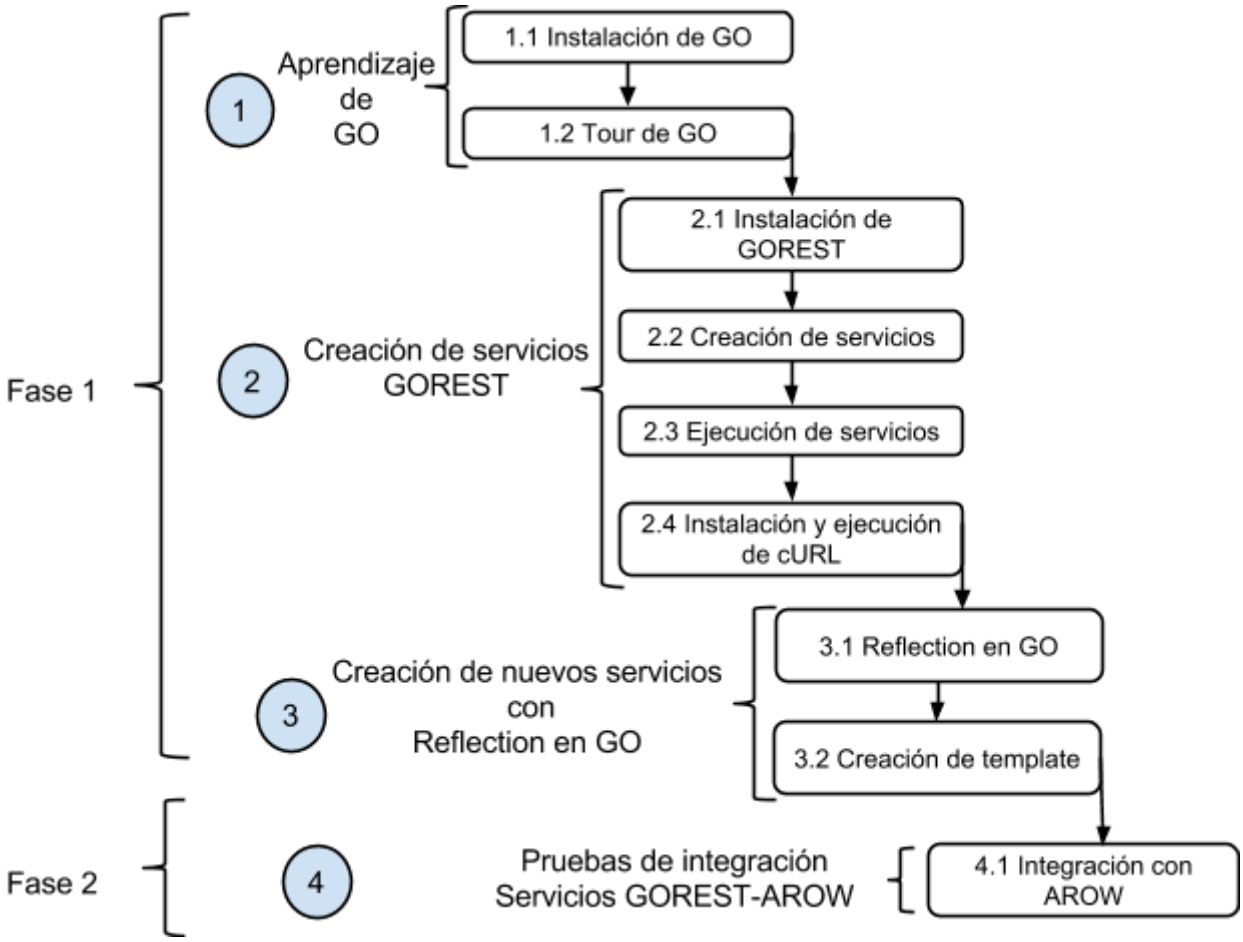

Figura 14.Proceso para realización de pruebas Servicios GOREST con el cliente AROW.

La primera tarea fue la instalación de GO, para después seguir un pequeño curso de programación, para comprender la sintaxis, comandos, estructuras entre otras opciones

del lenguaje, este curso se encuentra en la página del Tour de Go, en la dirección http://tour.golang.org/#1.

La segunda tarea consistió en la instalación de GOREST (que se puede descargar en la dirección https://code.google.com/p/gorest/), para la creación de algunos servicios sencillos. El ejemplo utilizado se refiere alos servicios de declaraciones patrimoniales para servidores públicos de un gobierno estatal en este contexto, con base a la estructura de un declarante. Para ejecutar dichos servicios y posteriormente probarlos de manera rápida con la ayuda de cURL, que es un herramienta para ejecutar comandos en consola y simular acciones u operaciones simulando un navegador web. Para realizar estas pruebas se crearon servicios REST con las opciones de POST, GET, DELETE y UPDATE. Estas pruebas se realizaron en los sistemas operativos Ubuntu 12.04 y MAC OS X 10.8.

La tercera tarea fue la creación automática de los servicios REST en base a una estructura cualquiera. Es decir, si se realiza una estructura de una casa, automáticamente se crearán los servicios de POST, GET, DELETE y UPDATE con los atributos definidos en la estructura casa como número de habitaciones, número de baños, etc. Aquí se realizó la codificación de un programa en GO que, mediante reflection o reflexión, obtiene los atributos de una estructura para crear los servicios REST, para finalmente completar un template o plantilla que contiene las acciones a ejecutar como POST, GET, DELETE y UPDATE.

La cuarta tarea fue la integración de estos servicios con AROW. Sin embargo, aunque se cumplió con la especificación de Restful Objects, nuestros servicios no podían ser consumidos por el cliente AROW. Esta fue la primera indicación de que los servicios de prueba no eran compatibles con el cliente AROW.

La primera acción que se tomó fue leer nuevamente la especificación y comprobar que los servicios de prueba la cumplieran la especificación. Al comprobar que efectivamente se estaba siguiendo, se decidió concentrarse en el lado del cliente y analizar el funcionamiento de AROW. Para ello, bajo el código fuente de los servidores de Restful Objects. Net y Apache ISIS y se hizo un seguimiento paso por paso para ver qué era lo que el cliente AROW estaba solicitando así como la respuesta brindada por los servidores.

Al iniciar las pruebas, siguiendo paso por paso la ejecución de AROW, se observó que solicitaba información o campos muy específicos, que no se mencionan ni en la especificación de Restful Ojects, ni en ningún otro documento en RestfulObjects.org . Por ejemplo: no se detallaba el contenido exacto que se requería en nuevos servicios y estructuras como DomainTypes, Version, Users, Services y el contenido o campos del recurso Links (Ver bitácora de implementación en la sección de ANEXOS).

Aún con estos problemas, se intentó continuar con la implementación del servidor, así que se trató de enviar toda la información que el cliente AROW solicitaba. Sin embargo, conforme se agregaban más campos, seguía solicitando otros y así sucesivamente. Al no tener documentación, esta labor se convirtió en una tarea demasiado complicada y

consumió el tiempo disponible para la codificación del servidor. Finalmente, se declaró el experimento fallido.

El ANEXO “Bitácora de implementación” describe de manera detallada todo el proceso y códigos que se ejecutaron para tratar de implementar el servidor de Restful Objects en GO. Esto con el objetivo de que sea útil para reproducir y mejorar el experimento.

3. DISCUSIÓN Y RESULTADOS

En la sección anterior “Implementación de Restful Objects” y en el Anexo “Bitácora de Implementación”, se expone de manera puntual las acciones tomadas para tratar de implementar un servidor Restful Objects. En esta sección se explica de manera sintetizada la interpretación de la experiencia en el desarrollo. Para guiar esta discusión se utilizó la técnica de los “5 por qués” y el análisis plus/delta.

La técnica de los “5 por qués” pretende identificar las relaciones causa y efecto de los problemas [21]. A diferencia de otros métodos de resolución de problemas complejos, no implica la segmentación de datos, pruebas de hipótesis, regresión u otras herramientas avanzadas. Al responder repetidamente la pregunta “¿Por qué?” al menos cinco veces, se puede identificar la causa raíz de un problema.

El análisis "Plus/Delta" es una técnica que provee retroalimentación de una experiencia o evento y colecta ideas para futuras mejoras. Por medio de esta técnica se busca

capturar los aspectos positivos (+) que han resultado de los esfuerzos de mejora continua y los deltas (Δ) que son los cambios que todavía tienen que hacerse [22].

Aplicación de la técnica "5 Por qués"

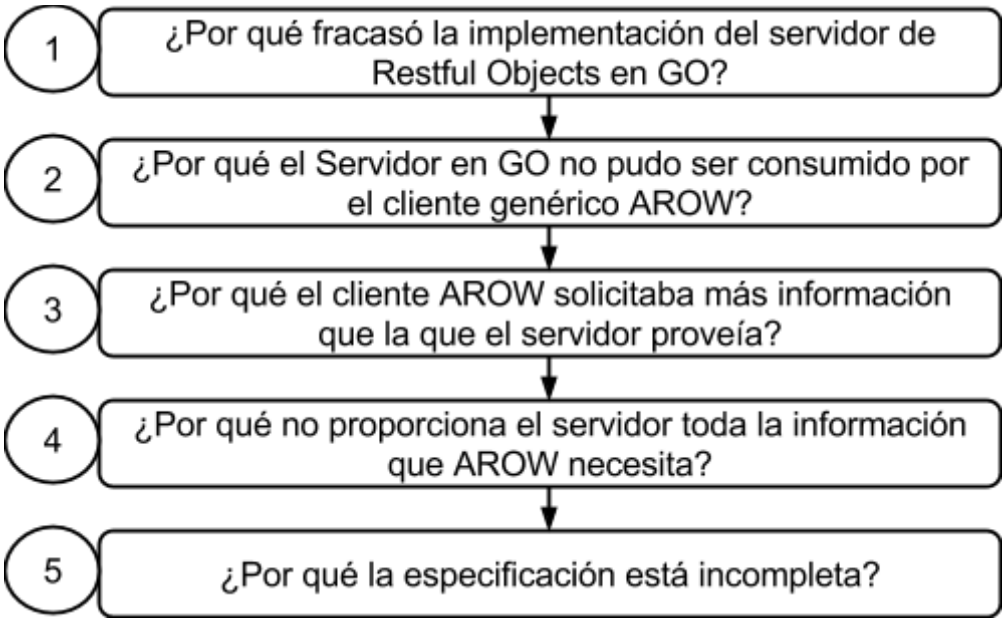

Figura 15. Los "5 por qués" del proyecto.

1. ¿Por qué fracasó la implementación del servidor de Restful Objects en GO?

Primeramente, no se considera como un fracaso, realmente se hizo un servidor que sigue la especificación Restful, usando el lenguaje de programación GO. Lo que falló fue el que el servidor pudiera ser consumido por el cliente genérico AROW.

2. ¿Por qué el Servidor en GO no pudo ser consumido por el cliente genérico AROW?

Porque el cliente AROW solicitaba más campos e información que los que el servidor proveía.

3. ¿Por qué el cliente AROW solicitaba más información que la que el servidor proveía?

Porque para dibujar la interfaz de usuario, AROW necesita más información que la que le proporciona la especificación de Restful Objects. Esa información sí es proporcionada por los servidores en .Net y en Apache ISIS.

4. ¿Por qué no proporciona el servidor toda la información que AROW necesita?

Porque en la especificación esa información no la documenta, es decir, la especificación está incompleta.

5. ¿Por qué la especificación está incompleta?

Porque se hizo antes que los servidores en .Net y Java y cuando se desarrollaron los servidores no se regresó a complementar la especificación.

De este ejercicio se concluye que la especificación de Restful Objects, como se encuentra publicada en la versión 1.0 está incompleta y es necesario actualizarla para que incluya detalles más específicos, que fueron descubiertos al tratar de implementar los servidores .Net y Apache Isis. Hubiera sido de gran ayuda algún documento con ejemplos sencillos de cómo funcionan los servidores Restful Objects.

Aplicación de la técnica “Análisis Plus Delta”

¿Qué se hizo bien y se debería repetir en el futuro?

- No sólo leer la especificación, sino tratar de crear un servidor que la implemente. Al tratar de realizar la implementación se obtuvo un mejor conocimiento de la especificación.
- Usar un cliente genérico que se sabe funciona con las implementaciones existentes para evaluar la compatibilidad de la implementación propia.
- Dividir el trabajo en tareas y realizar juntas constantes para revisar avances.
- Hacer consultas a personas con un mayor conocimiento acerca de las tecnologías utilizadas, por ejemplo con comandos sencillos, que no se toman en cuenta en el curso de introducción del Tour de GO o seguridad en servidores para los permisos y consumir los servicios REST.

¿Qué se podría mejorar en el futuro?

- Se esperó demasiado para hacer las pruebas con el cliente genérico, lo cual causó que, cuando se descubrió lo incompleto de la especificación ya se había consumido todo el tiempo disponible.
- El desarrollo se basó sólo en el documento de la especificación y se pudo haber estudiado los servidores que ya existían y así ver todos los datos antes de iniciar con la codificación de la implementación de servidor GO.
- Realizar el proyecto con un lenguaje mejor conocido. La codificación del servidor se hizo en GO, que era un lenguaje totalmente desconocido, por lo que,

el tiempo para realizar la implementación consumió mucho del tiempo disponible para el experimento.

- En un futuro se podría hacer una implementación estática del servidor, que demostrara funcionamiento correcto con AROW. Esta implementación estática serviría como una especificación ejecutable, así posteriormente, las pruebas de unidad bien escritas podrán proveer una especificación de trabajo para el código funcional y dando como resultado pruebas de unidad que efectivamente se conviertan en una significante porción de documentación técnica[23].
- De esta técnica se deriva que, en un futuro intento será conveniente iniciar con las pruebas lo más temprano posible, para evitar imprevistos y retrasos en las implementaciones.

En resumen la experiencia de este experimento muestra el valor de las documentaciones completas y actualizadas. Además mostró que es necesario siempre regresar a actualizar la especificación con el aprendizaje de la implementación y plantea estudiar con más atención las especificaciones ejecutables, ya que estas podrían ofrecer una alternativa para facilitar la implementación de futuros servidores.

4. CONCLUSIONES Y TRABAJO FUTURO

Después de terminar el servidor en GO, al tratar de validarlo, se encontró que en realidad la especificación está incompleta, ya que el cliente consume muchos más datos de los que el servidor proporciona. Lo cual lleva a las siguientes conclusiones:

- Si el experimento hubiera comenzado haciendo un análisis más detallado del código fuente del cliente AROW y un solo servidor, se pudieron haber especificado una serie de pruebas, que podrían ser válidas para la implementación del servidor en GO y para posteriores implementaciones.

- Hay un conocimiento tácito de quien crea e implementa los servidores en Java y .NET, pero el problema es que no se encuentra registrado ninguna especificación documentada de estas versiones, aunque los dos servidores fueron desarrollados por el equipo core que propone Restful Objects

- La especificación de Restful Objects todavía no está completa. Esto puede deberse a que existen pocos desarrolladores familiarizados con este tema.

- Sería muy útil tener documentados algunos ejemplos sencillos de los mensajes y parámetros enviados por los servidores, explicando cómo se hace la generación de las distintas funciones utilizadas en la interfaz gráfica para el usuario final.

- Las especificaciones y documentación frecuentemente se queda atrás de las versiones más recientes de software.

- Hay inconsistencias entre la documentación y las dos implementaciones existentes.

Para otro equipo que en un futuro desee hacer una implementación de Restful Objects, se le hacen las siguientes recomendaciones:

- Hacer ingeniería inversa. Es decir, observar los mensajes que se envían entre el cliente AROW y los servidores existentes (.NET o Java), primero concentrándose en la versión que sea más simple (Java), con el objetivo de obtener una serie de pruebas estándar que validen a otros posibles nuevos servidores.

- Realizar un servidor *dummy* con la menor complejidad posible, es decir, crear una lista de servicios que regresen respuestas sencillas predeterminadas. Un ejemplo de esto, es utilizar programas o páginas como Apiary (http://apiary.io), para concentrarse primero en las salidas preestablecidas, antes de comenzar con la codificación de funciones que tal vez puedan implicar gran inversión de tiempo.

- Finalmente, complementar la especificación Restful Objects con el resultado de dicha ingeniería inversa.

Aunque no se logró validar el servidor en GO, el paradigma Restful Objects parece sólido y este reporte técnico sienta las bases para que otro equipo en el futuro, continúe con este proyecto.

"No fracasé, sólo descubrí 9,999 maneras de cómo no hacer una bombilla."

Thomas Alva Edison.

5. REFERENCIAS

[1] Pawson R. (2004). *Naked Objects*. Consultado mayo 12, 2014, de University of Dublin, Trinity College Sitio web: http://downloads.nakedobjects.net/resources/Pawson%20thesis.pdf

[2] Salihefendic A. (2011). *Model View Controller: History, theory and usage* . Consultado junio 15, 2014, de Amix Sitio web: http://amix.dk/blog/post/19615

[3] Pawson R & Matthews R. (2002), *Naked Objects*, Estados Unidos: John Willey & Sons, LTD.

[4] Tarhini A. (2011). *Concepts of Three-Tier Architecture*, Consultado mayo 25, 2014, de Blog de Ali Tarhini Sitio web: http://alitarhini.wordpress.com/2011/01/22/concepts-of-three-tier-architecture/

[5] Rawsthorne P. (2011). *MVC in a three-tier architecture*, Consultado mayo 26, 2014, de Critical Technology Sitio web: http://criticaltechnology.blogspot.mx/2011/10/mvc-in-three-tier-architecture.html

[6] Brooks F. P. (2002), *The Mythical Man-Month Essays on Software Engineering*, Estados Unidos: Addison-Wesley.

[7] Gopalakrishna A. & Udayashankar B. (2012). *Object Oriented Databases.* Consultado mayo 28, 2014, de University of Colorado, Department of Computer Science
Sitio web: http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/udayashankar.pdf

[8] Brooks F. P. & Barry J. K. (2013). *Transparent Persistence in Object Databases.*Consultado julio 1, 2014, de Service Architecture Sitio web: http://www.service-architecture.com/articles/object-oriented-databases/transparent_persistence.html

[9] López D. (2013). *Base de datos: enfoque orientado a objetos.* Consultado julio 3, 2014, de Cibertec Sitio web: https://my.laureate.net/Faculty/webinars/Documents/2013Agosto_Base%20de%20Datos%20Enfoque%20Orientado%20Objetos.pdf

[10] Haywood D. (2009), *Domain-Driven Design Using Naked Objects*, Estados Unidos: Pragmatic Bookshelf.

[11] Suarez R. (2010). *POJOS en java web*. Consultado julio 5, 2014, de Rah Suarez's Blog Sitio web: http://rahsuarez.wordpress.com/2010/03/18/pojos-en-java-web/

[12] Documentación de Hibernate, Consultado julio 6, 2014, de Hibernate, Sitio web: http://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/persistent-classes.html

[13] Sabnis M. (2011). *ADO.NET EF 4.0: Working with Plain Old CLR Objects (POCO) Classes*. Consultado julio 6, 2014, de dotnetcurry, Sitio web: http://www.dotnetcurry.com/showarticle.aspx?ID=725

[14] Isora S. (2001). *Object-oriented real-world modeling revisited*. Consultado Julio 10, 2014, de ScienceDirect, SitioWeb: http://www.sciencedirect.com/science/article/pii/S0164121201000590

[15] Orenstein D. (2000). *QuickStudy: Application Programming Interface (API), de ComputerWorld,* Consultado julio 7, 2014, de ComputerWorld, Sitio web: http://www.computerworld.com/s/article/43487/Application_Programming_Interface

[16] Francia S. (2010). *REST VS SOAP.*Consultado julio 11, 2014, de spf13, Sitio Web: http://spf13.com/post/soap-vs-rest

[17] Definición de REST Sitio Web: http://es.wikipedia.org/wiki/Representational_State_Transfer

[18] Haywood D. & Pawson R. (2012). *Introducing: Restful Objects*. Consultado mayo 3, 2014, de Infoq, Sitio web: http://www.infoq.com/articles/Intro_Restful_Objects

[19] Travis Reeder. (2013). *How We Went from 30 Servers to 2: Go*, Consultado mayo 6, 2014, de Iron.io Sitio web: http://blog.iron.io/2013/03/how-we-went-from-30-servers-to-2-go.html

[20] Página Oficial de GOREST, Sitio Web: https://code.google.com/p/gorest/

[21] Serrat O. (2009). *The Five Whys Technique*, de Knowledge, Consultado agosto 13, 2014, de Solutions Sitio web: http://www.adb.org/sites/default/files/pub/2009/the-five-whys-technique.pdf

[22] Chaneski W. (2008). *A Simple Technique For Evaluating Success*, Consultado agosto 14, 2014 de Modern Machine Shop Sitio web: http://www.mmsonline.com/columns/a-simple-technique-for-evaluating-success

[23] Ambler S. W. (2006). *Agile Best Practice: Executable Specifications*, Consultado agosto 14, 2014, de Agile Modeling Sitio web: http://agilemodeling.com/essays/executableSpecifications.htm#Figure1StepsOfTDD

6. ANEXOS

BITÁCORA DE IMPLEMENTACIÓN

A continuación se describe paso a paso las tareas ejecutadas para realizar el experimento para implementar un servidor Restful Objects en GO. El código fuente de estos programas se encuentran en repositorio de GitHub en la dirección:

https://github.com/octavioreyes1/RestfulObjectsGO.

6.1. APRENDIZAJE DE GO

6.1.1 Instalación de GO

La instalación del lenguaje de programación GO y el framework GOREST, se realizó en dos sistemas operativos (MAC OS X versión 10.8.3. y Ubuntu 12.04):

Paso 1: Instalación de mercurial, se utiliza para crear una copia del repositorio de GOREST:

En Ubuntu:

* Sudo apt-get install mercurial (el sistema operativo indicará si se necesita instalar otras librerías dependientes)

En Mac:

* Bajar la librería mercurial-2.5.1+20130210-py2.7-macosx10.8.
* Abrir y arrastrar a la carpeta de aplicaciones.

Paso 2: Instalación de GO:

Ubuntu:

* Ejecutar *sudo apt-get install golang-go* en modo consola

Mac

* Bajar la paquetería go1.0.3.darwin-amd64-signed.pkg

* Abrir y arrastrar a la carpeta de aplicaciones

Paso 3: Probar GO.

En este paso se recomienda elaborar un pequeño código de “hola mundo”. Para lo cual se abre un archivo, usando cualquier editor de texto, como por ejemplo Gedit o Sublimetext y agregar el siguiente código:

```
package main
import "fmt"
func main(){
fmt.Printf("hello, world\n")
}
```

Se guarda el archivo como “hello.go”. Para ejecutarlo, se necesita entrar en el modo consola del sistema operativo, entrar al directorio donde se encuentre el archivo hello.go y ejecutar el comando:

```
#go run hello.go
```

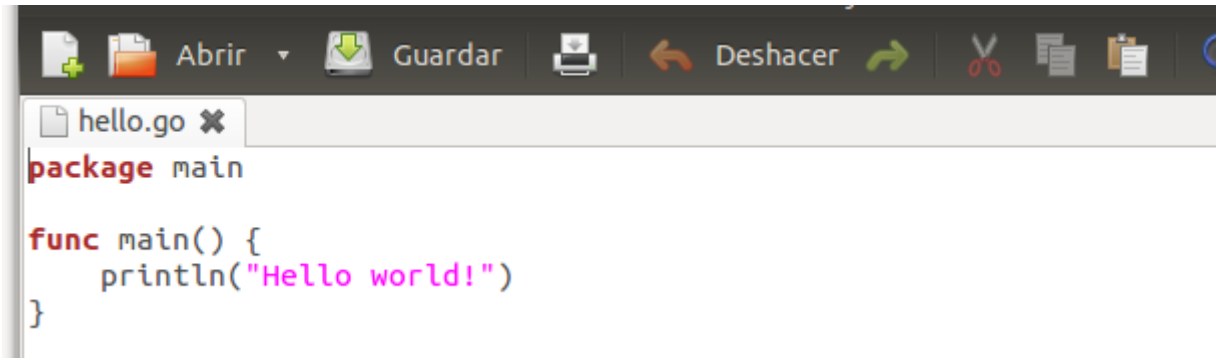

Figura A.1.1. Archivo “hello.go” creado con Gedit en Ubuntu 12.04.

```
root@escaner:/home/escaner/Documentos# go run hello.go
Hello world!
root@escaner:/home/escaner/Documentos#
```

Figura A.1.2. Ejecución del archivo “hello.go”

6.1.2 El tour de GO

Para la aplicación del proyecto fue necesario el estudio del lenguaje de programación GO, por lo cual es necesario tomar el tour de la página principal del proyecto de GOLANG (golang.org/) en la sección A tour of Go, en el cual se sugiere la elaboración de alrededor de 70 ejercicios de programación. Los ejercicios son similares al que se muestra a continuación:

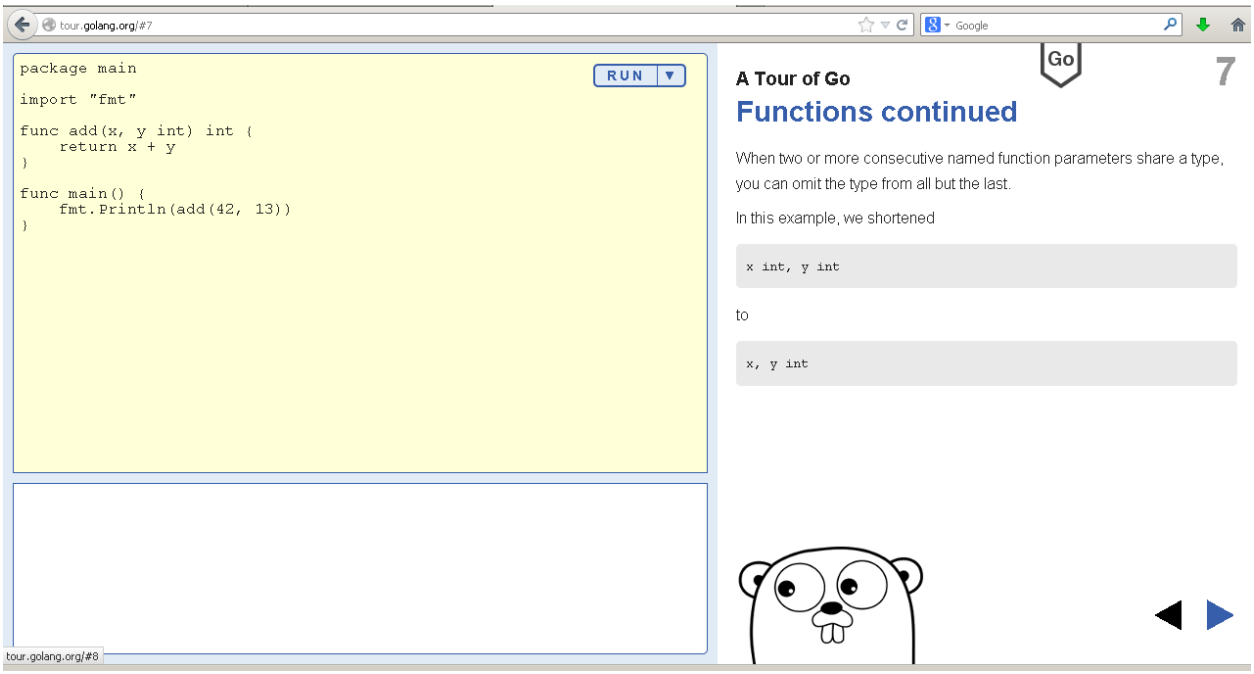

Figura A.1.3. Ejercicio 7 del tour de GO.

6.2. CREACIÓN DE SERVICIOS GOREST

6.2.1 Instalación de GOREST

Una vez concluido el tutorial de GO, se procedió con la elaboración de un servicio web con GO, para ello se tiene que bajar el framework GOREST (http://code.google.com/p/gorest/), para realizar esto se ejecuta el siguiente comando de GO:

```
#go get code.google.com/p/gorest.
```

Al terminar de descargar el framework, se creó un servicio de declarantes (estructura base de un sistema de declaraciones patrimoniales para trabajadores de Gobierno), para efectuar las operaciones de altas, bajas cambios y consultas de declarantes, dicho servicio se realizó en base al ejemplo mostrado en la página principal de GOREST (http://code.google.com/p/gorest/), como se muestra en la siguiente ilustración.

Example Usage:

Install: *go get code.google.com/p/gorest*

A full workin example is provided with the sources.

```
func main(){
    gorest.RegisterService(new(OrderService))
    http.Handle("/",gorest.Handle())
    http.ListenAndServe(":8787",nil)

}

//**********************Define Service**************************

type OrderService struct{
    //Service level config
    gorest.RestService    `root:"/orders-service/" consumes:"application/json" produces:"application/json"`

    //End-Point level configs: Field names must be the same as the corresponding method names,
    // but not-exported (starts with lowercase)

    userDetails gorest.EndPoint `method:"GET" path:"/users/{Id:int}" output:"User"`
    listItems   gorest.EndPoint `method:"GET" path:"/items/" output:"[]Item"`
    addItem     gorest.EndPoint `method:"POST" path:"/items/" postdata:"Item"`

    //On a real app for placeOrder below, the POST URL would probably be just /orders/, this is just to
    // demo the ability of mixing post-data parameters with URL mapped parameters.
    placeOrder  gorest.EndPoint `method:"POST" path:"/orders/new/{UserId:int}/{RequestDiscount:bool}" postdata:"Order"`
    viewOrder     gorest.EndPoint `method:"GET" path:"/orders/{OrderId:int}" output:"Order"`
    deleteOrder     gorest.EndPoint `method:"DELETE" path:"/orders/{OrderId:int}"`


}

//Handler Methods: Method names must be the same as in config, but exported (starts with uppercase)



func(serv OrderService) UserDetails(Id int) (u User){
    if user,found:=userStore[Id];found{
        u =user
        return
```

Figura A.2.1. Ejemplo de un servicio GOREST en la página web principal de GOREST

Para obtener una copia local del repositorio de GOREST, se ejecuta el siguiente comando de consola:

hg clone https://code.google.com/p/gorest/

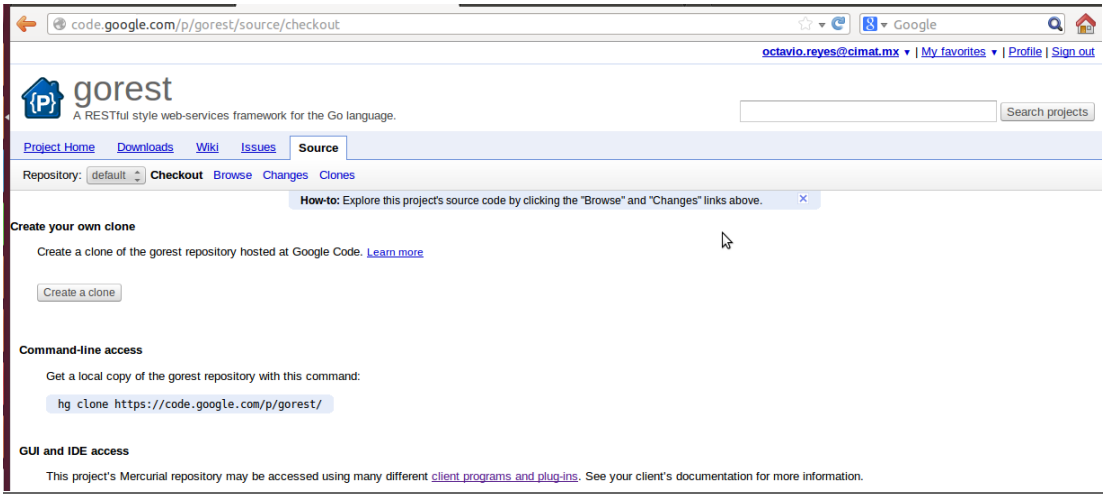

Figura A.2.2. Repositorio GOREST.

En la imagen siguiente se puede observar la lista de librerías instaladas después de ejecutar el comando anterior.

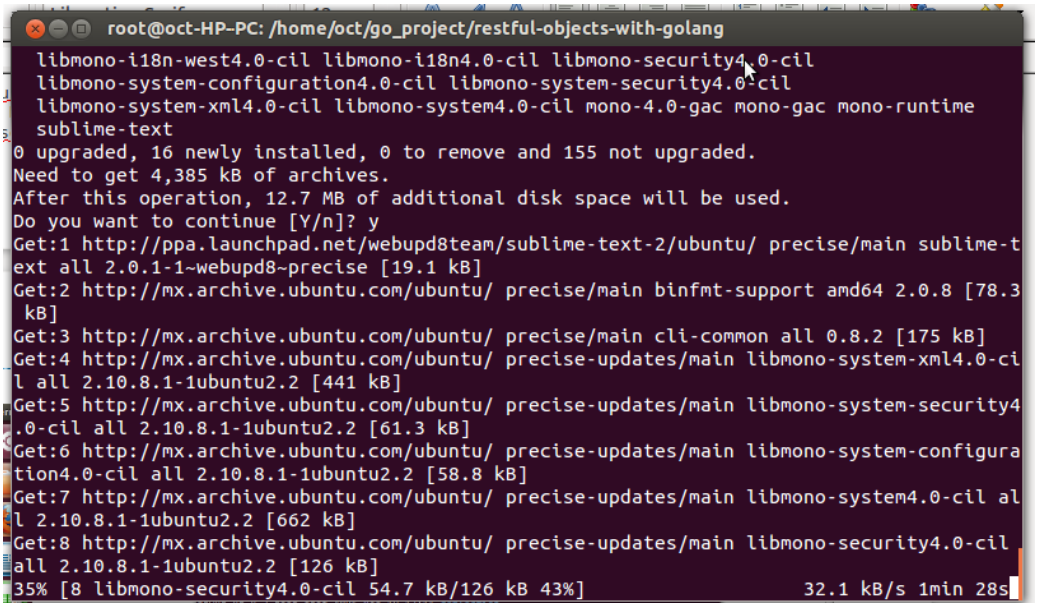

Figura A.2.3. Instalación de librerías en Ubuntu 12.04.

6.2.2 Creación de servicios

Una vez comprendido el ejemplo del servicio web, proporcionado en la página web de Gorest, se procedió con la codificación del servicio de declarantes patrimoniales con el nombre de declaranet-service, los componentes principales son Declarante (persona que hace una declaración patrimonial), DeclaranetService (servicios para agregar, borrar, etc. declarantes) y main (inicialización de servicios).

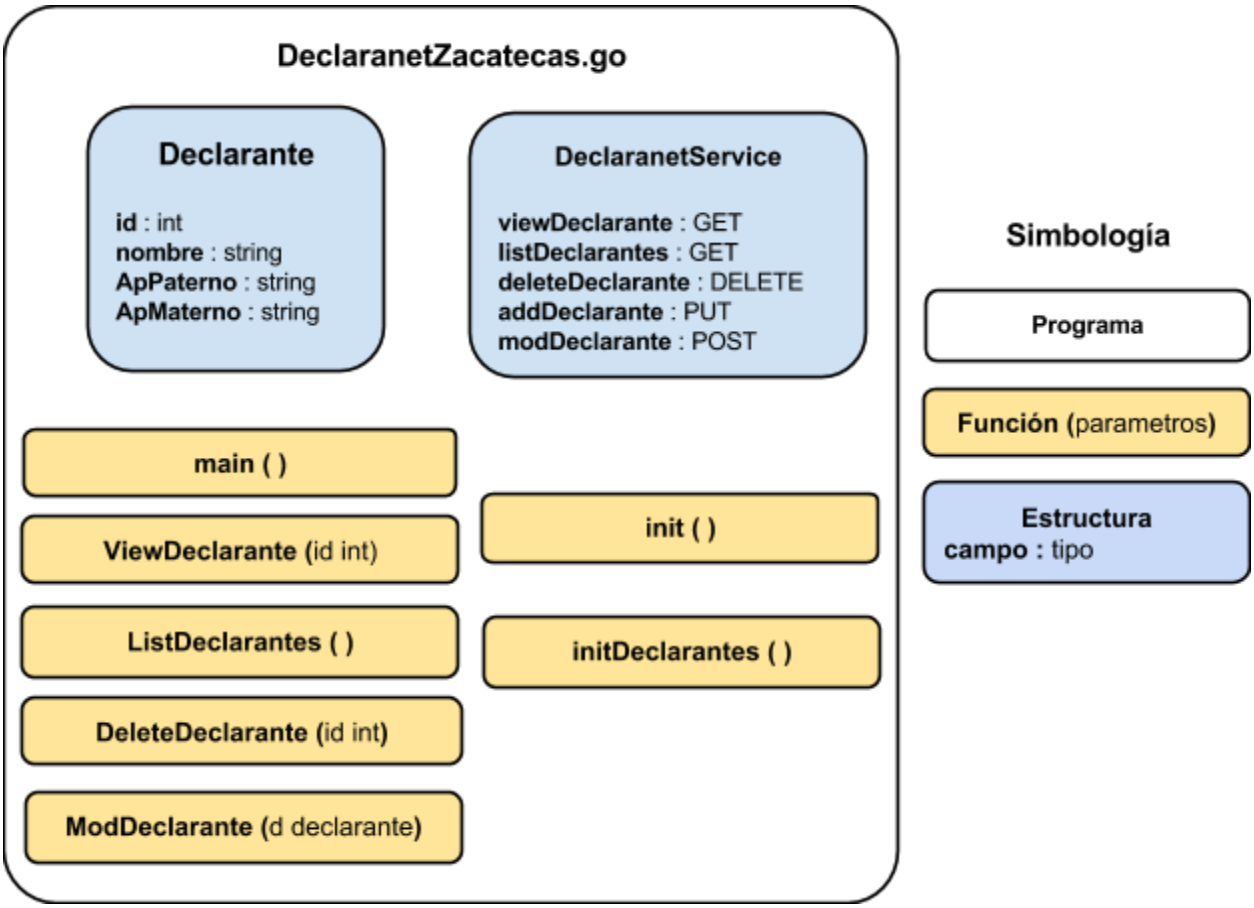

Figura A.2.4. Diagrama general de programa declaranetZacatecas.go.

La estructura del servicio lleva el nombre de DeclaranetService, en esta estructura se definen los “endpoints” o direcciones URL desde las que podemos

accesar a los servicios. Solamente se describen las rutas o direcciones URL, parámetros de entrada y valores de retorno (no se incluye la implementación).

La función principal se define como *main*, en esta parte se define la implementación completa de los servicios disponibles. Se asignan los mismos nombres de los servicios declarados en la estructura *DeclaranetService*, pero deben iniciar con letra mayúscula.

La estructura Declarante contiene los datos de la persona obligada a realizar su declaración patrimonial, contiene un identificador de tipo entero (id), nombre (string), apPaterno (string) y apMaterno (string).

La estructura Declarante se relaciona con la estructura DeclaranetService que es donde se definen las rutas de los servicios. También se relaciona con la estructura principal (main) al momento de declarar e inicializar una lista de declarantes por default.

En la siguiente imagen, se puede observar una parte del código fuente del archivo declaranetZacatecas.go

```
type DeclaranetService struct {
        //Service level config
        gorest.RestService `root:"/declaranet-service/" consumes:"application/json" produces:"application/json"`

        //End-Point level configs: Field names must be the same as the corresponding method names,
        // but not-exported (starts with lowercase)


        viewDeclarante   gorest.EndPoint `method:"GET" path:"/declarantes/{Id:int}" output:"Declarante"`
        listDeclarantes   gorest.EndPoint `method:"GET" path:"/declarantes/" output:"[]Declarante"`
        deleteDeclarante gorest.EndPoint `method:"DELETE" path:"/declarantes/{Id:int}"`
        addDeclarante     gorest.EndPoint `method:"PUT" path:"/declarantes/" postdata:"Declarante"`
        modDeclarante     gorest.EndPoint `method:"POST" path:"/declarantes/" postdata:"Declarante"`

}
```

Figura A.2.5. Servicios disponibles en declaranetZacatecas.go

Los *end points* proporcionados por este programa en GO, pueden ser consumidos a través de JSON. Estos *end points* son las URLs a través de las cuales podemos accesar para realizar las operaciones de altas, bajas, cambios y consultas de declarantes.

A continuación se muestra el código fuente del archivo **declaranetZacatecas.go** que se encuentra en la dirección:

https://github.com/octavioreyes1/RestfulObjectsGO/blob/master/declaranetZacatecas.go.

```
package main
// Librerías necesarias
import (
	"code.google.com/p/gorest"
	"net/http"
	"strconv"
)
// Función principal
func main(){
gorest.RegisterService(new(DeclaranetService))
  http.Handle("/",gorest.Handle())
http.ListenAndServe(":8787",nil)
}
/// Estructura principal que contiene la lista de servicios GET, POST, PUT y DELETE
type DeclaranetService  struct {
	//Service level config
```

```
        gorest.RestService `root:"/declaranet-service/" consumes:"application/json"
produces:"application/json"`

        //End-Point level configs: Field names must be the same as the corresponding method names,
        // but not-exported (starts with lowercase)


       viewDeclarante  gorest.EndPoint `method:"GET" path:"/declarantes/{Id:int}" output:"Declarante"`
        listDeclarantes  gorest.EndPoint `method:"GET" path:"/declarantes/" output:"[]Declarante"`
        deleteDeclarante  gorest.EndPoint `method:"DELETE" path:"/declarantes/{Id:int}"`
        addDeclarante  gorest.EndPoint `method:"PUT" path:"/declarantes/" postdata:"Declarante"`
        modDeclarante  gorest.EndPoint `method:"POST" path:"/declarantes/" postdata:"Declarante"`
}
////LISTAR DECLARANTES
func (serv DeclaranetService) ListDeclarantes() []Declarante {
        serv.ResponseBuilder().CacheMaxAge(60 * 60 * 24) //List cacheable for a day. More work to
come on this, Etag, etc
        return declaranteStore
}
////// VER UN DECLARANTE
func (serv DeclaranetService) ViewDeclarante(id int) (retDeclarante Declarante) {
        for _, declarante := range declaranteStore {
        if declarante.Id == id {
        retDeclarante = declarante
        return
        }
        }
    serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
```

```
}

/////// BORRAR UN DECLARANTE
func (serv DeclaranetService) DeleteDeclarante(id int) {
        for pos, declarante := range declaranteStore {
        if declarante.Id == id {
        declaranteStore[pos] = declarante
        return //Default http code for DELETE is 200
        }
        }
        serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
}
///////// AGREGAR UN DECLARANTE
func (serv DeclaranetService) AddDeclarante(d Declarante) {
        //Item Id not in database, so create new
        d.Id = len(declaranteStore)+1
        declaranteStore = append(declaranteStore, d)
    serv.ResponseBuilder().Created("http://localhost:8787/declarante-service/declarantes/" + string(d.Id))
//Created, http 201
}
//////// MODIFICAR DATOS DE UN DECLARANTE
func (serv DeclaranetService) ModDeclarante(d Declarante) {
        for pos, declarante := range declaranteStore {
        if declarante.Id == d.Id {
        declaranteStore[pos] = d
                serv.ResponseBuilder().SetResponseCode(200) //Updated http 200, or you could just
return without setting this. 200 is the default for POST
```

```
		return
		}
		}
}
//**************** End of service ******************
/// Estructura Declarante y sus campos
type Declarante struct {
		Id		int
		Nombre string
		ApPaterno string
		ApMaterno string
}
var (
// Declación de un arreglo de Declarantes
		declaranteStore []Declarante
)
// Función de inicialización para crear un arreglo de estructuras de tipo Declarante
func init() {
		declaranteStore = make([]Declarante, 0)
		initDeclarantes()
}
////////// INICIALIZAR UN ARREGLO CON 10 DECLARANTES
func initDeclarantes() {
		for i := 1; i <= 10; i++ {
		declaranteStore = append(declaranteStore, Declarante{Id: i,
		Nombre: "Nombre" + strconv.Itoa(i),
		ApPaterno: "Apellido P." + strconv.Itoa(i),
			ApMaterno: "Apellido M." + strconv.Itoa(i)})
```

```
        }
}
```

6.2.3 Ejecución del servicio declaranet-service

Una vez que se terminó la codificación del servicio, para iniciarlo se ejecuta el siguiente comando, en la consola del sistema operativo (MAC o Ubuntu):

```
# go run declaranetZacatecas.go
```

Una vez ejecutado el comando, podemos ver que el servicio está inicializado, indicando que los servicios han sido registrados y se encuentran disponibles.

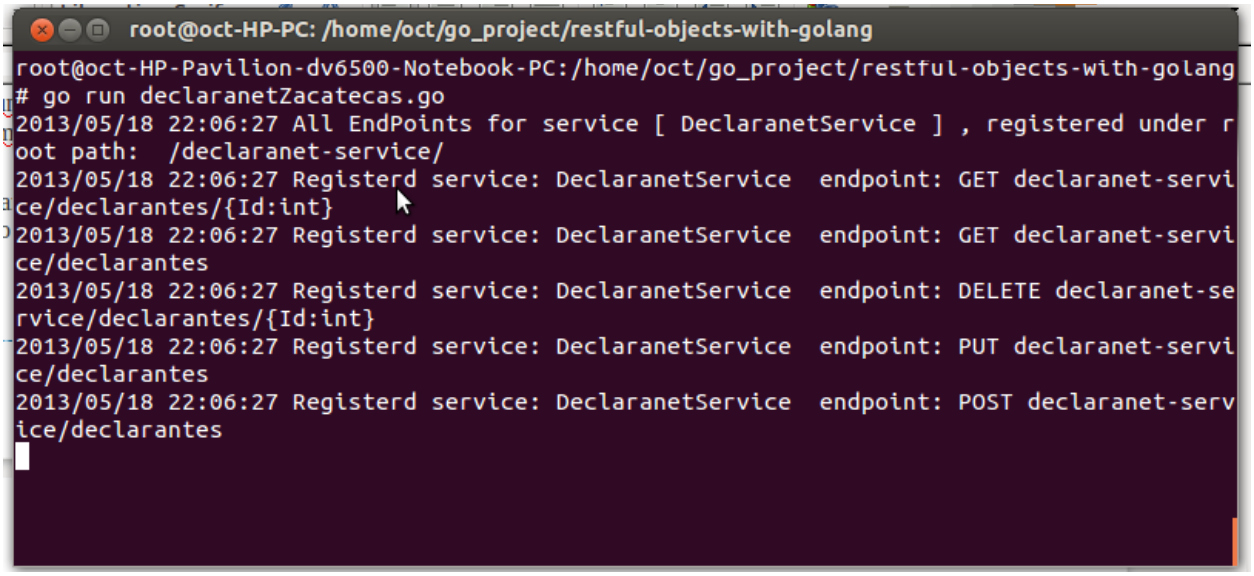

Figura A.2.6. Confirmación de servicios disponibles en DeclaranetService.

Para probar la API del servicio GO, antes de usarlo en una aplicación cliente, se puede invocar el servicio REST directamente desde un navegador con solo teclear la URL correspondiente. Por ejemplo:

- http://localhost:8787/declaranetservice/declarantes Para ver lista completa de declarantes.
- http://localhost:8787/declaranetservice/declarantes/1 Para ver los datos de un solo declarante por ID, el ID en el ejemplo es 1.

Para probar las operaciones como bajas o cambios, no podemos utilizar el navegador web solamente, por lo cual una alternativa es usar cURL, que es una solución versátil para probar los servicios Restful, mediante una utilidad de línea de comandos, para transferir datos por medio de la sintaxis de una URL.

cURL es una herramienta de línea de comandos para transferir archivos con sintaxis URL, soporte FTP, FTPS, TFTP, HTTP, HTTPS, TELNET, DICT, FILE y LDAP. También soporta certificados SSL, HTTP POST, HTTP PUT, envío por FTP, HTTP que forman la base de carga, proxies, cookies, autenticación de usuario + contraseña (Basic, Digest, NTLM, Negotiate, Kerberos), además de reanudar la transferencia de archivos, túneles proxy, entre otras opciones.

6.2.4 Instalación y ejecución de cURL

Para instalar cURL, al entrar en modo consola se ejecuta el siguiente comando:

```
# sudo apt-get install curl
```

Una vez instalado, se pueden probar los servicios GOREST, sin la necesidad de utilizar un navegador web.

Comandos CURL

Para poder probar las acciones de altas, cambios y consultas, se ejecuta en una terminal de consola:

#go run declaranetZacatecas.go

Una vez iniciado el servicio de declaranet-service, en otra terminal de consola ejecutamos los siguientes comandos para probar el funcionamiento correcto.

Para listar todos los declarantes ejecutamos el comando:

curl -i -X GEThttp://localhost:8787/declaranet-service/declarantes/

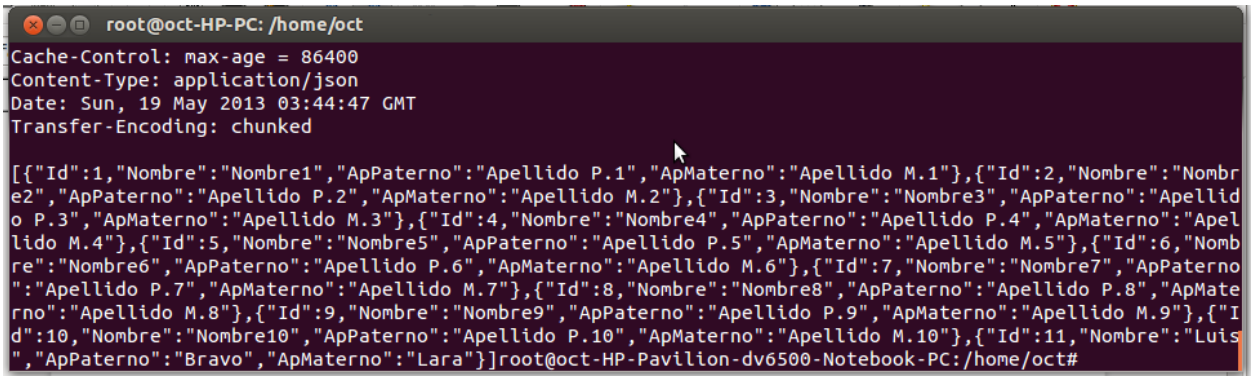

Figura A.2.7. Listado de declarantes.

Ø Para ver los datos de un solo declarante por ID (en este caso el ID es 1):

curl -i -X GEThttp://localhost:8787/declaranet-service/declarantes/1

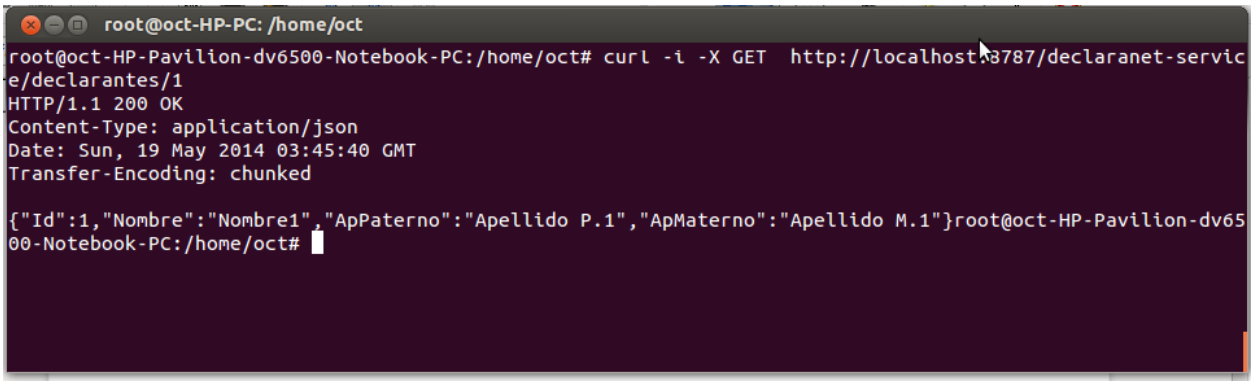

Figura A.2.8. Datos del empleado con el ID igual a 1.

Ø Para borrar un declarante por ID (el ID=1):

curl -i -X DELETEhttp://localhost:8787/declaranet-service/declarantes/1

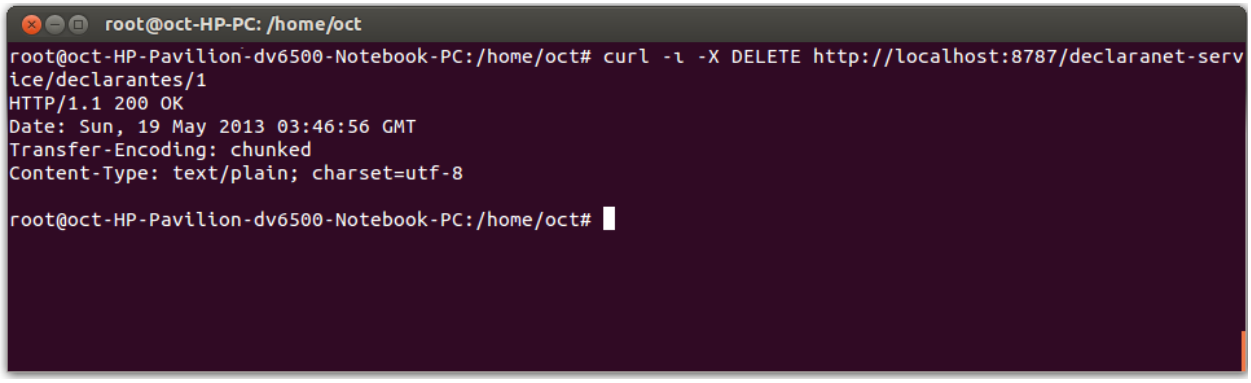

Figura A.2.9. Eliminación de declarante con el ID igual a 1.

Ø Agregar un nuevo declarante, al insertar un nuevo declarante automáticamente se le asigna un ID autonumérico:

curl -i -X PUT -H 'Content-Type: application/json' -d '{"Nombre": "Juan", "ApPaterno": "Lara", "ApMaterno": "Bravo"}'http://localhost:8787/declaranet-service/declarantes/

Figura A.2.10. Insertar un declarante con cURL.

Una vez ejecutado el comando para agregar un declarante, aparece la confirmación de que el nuevo declarante fue agregado con éxito.

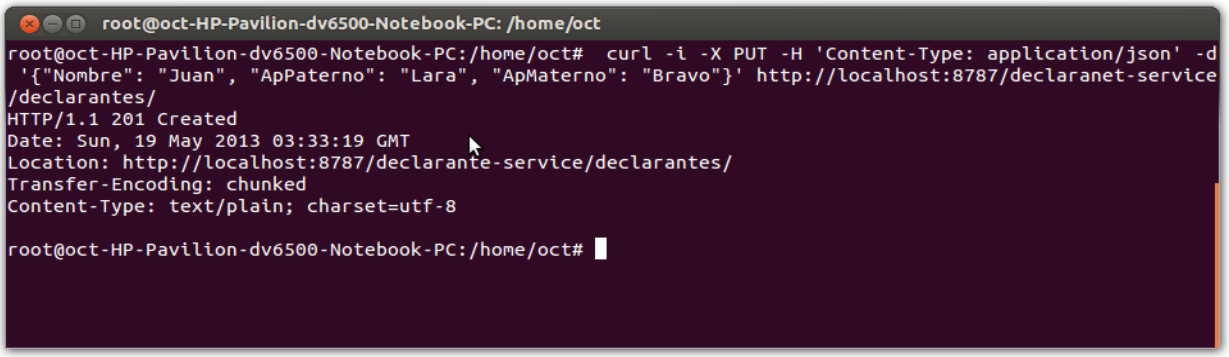

Figura A.2.11. Mensaje de confirmación de declarante insertado.

Se puede comprobar que el nuevo declarante fue creado con la ayuda del navegador web, tecleando la URL:

http://localhost:8787/declarantes/declarante-service/declarantes/11

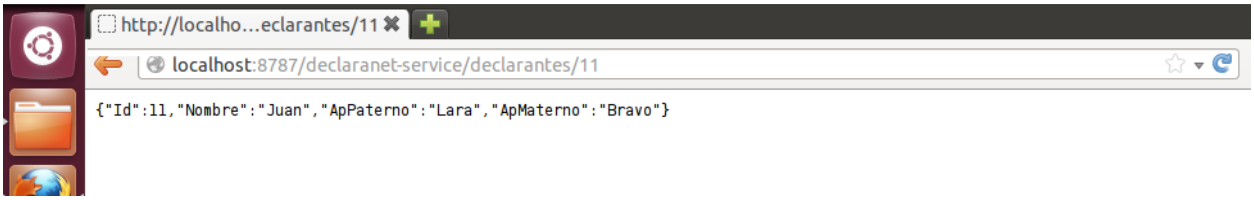

Figura A.2.12. Declarante con ID 11 visualizado en el navegador web Firefox.

Es importante aclarar que el número 11 fue el ID del nuevo declarante. Si se agrega después un nuevo declarante, se le asignará automáticamente el ID 12.Para reducir el tiempo de codificación, todas las operaciones para guardar datos de los declarantes se hacen en arreglos temporales, por lo que no se utilizó ningún manejador de base de datos.

Ø Modificar datos de un declarante:

curl -i -X POST -H 'Content-Type: applicatino/json' -d '{"Id": 11, "Nombre": "Luis", "ApPaterno": "Bravo", "ApMaterno": "Lara"}'

Se pueden confirmar los cambios realizados al consultar el declarante por ID. Por ejemplo al teclear la URL:

http://localhost:8787/declaranet-service/declarantes/11

Figura A.2.13. Datos del declarante con el ID con valor de 11 en el navegador web.

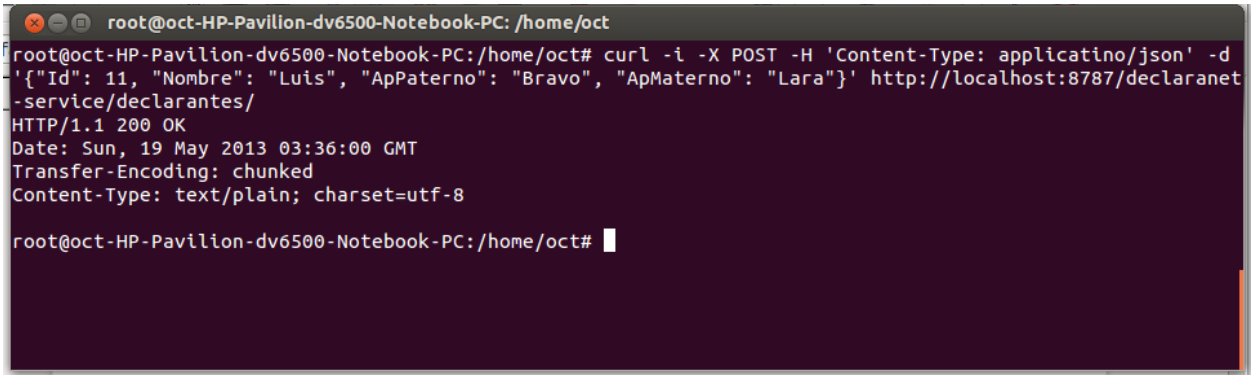

Figura A.2.14. Confirmación en consola, que indica que los cambios fueron realizados.

6.3. CREACIÓN DE NUEVOS SERVICIOS CON REFLECTION EN GO

6.3.1 Reflection con GO

Reflection o reflexión es un paradigma de programación, que permite conocer la forma en la que están construidas las clases del código que se está ejecutando (ingeniería inversa).

Durante el proyecto se utilizó Reflection, con la finalidad de crear de manera automática, la implementación de los servicios de altas, bajas, cambios y consultas de cualquier estructura, mediante la utilización de una plantilla y contando únicamente el código de la estructura con sus correspondientes campos.

De manera general, en la siguiente figura se describe el proceso que se lleva a cabo durante la generación automática de servicios, utilizando como base una estructura realizada en el lenguaje GO. En las siguientes páginas se describe a detalle el funcionamiento de cada código que aparece en la figura.

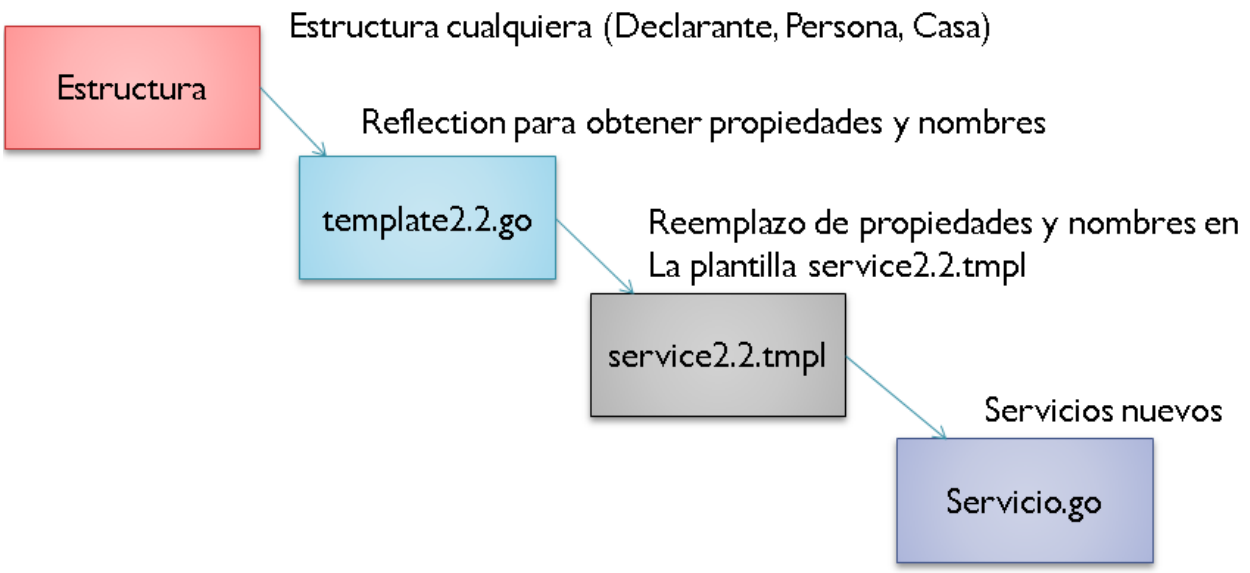

Figura A.3.1. Proceso general para generar servicios automáticamente.

En las siguientes imágenes se muestra un ejemplo sencillo de cómo obtener los atributos o propiedades de una estructura en GO. En el lenguaje GO no se utilizan clases y objetos, solamente se utilizan estructuras.

*ReflectionDeclarante.go

```
package main
import (
        "fmt"
        "reflect"
)

func main(){
    for name,  mtype := range attributes(&Declarante{}) {
          fmt.Printf("Name: %s, Type %s\n", name, mtype.Name())
        }

}

func attributes(m interface{}) (map[string]reflect.Type) {
     typ:=reflect.TypeOf(m)

     if typ.Kind()==reflect.Ptr{
          typ=typ.Elem()

}

attrs:=make(map[string]reflect.Type)

if typ.Kind() !=reflect.Struct{
     fmt.Printf("%w type cant have attributes inspected\n", typ.Kind())
    return attrs
}

for i:=0; i < typ.NumField(); i++ {
     p:=typ.Field(i)
          if !p.Anonymous {
                    attrs[p.Name]=p.Type
      }
}

return attrs
}
```

Figura A.3.2. Primera parte del programa de Reflection, tomando como basela estructura “declarante”.

*ReflectionDeclarante.go

```
func attributes(m interface{}) (map[string]reflect.Type) {
      typ:=reflect.TypeOf(m)

      if typ.Kind()==reflect.Ptr{
           typ=typ.Elem()

}

attrs:=make(map[string]reflect.Type)

if typ.Kind() !=reflect.Struct{
      fmt.Printf("%w type cant have attributes inspected\n", typ.Kind())
     return attrs
}

for i:=0; i < typ.NumField(); i++ {
      p:=typ.Field(i)
           if !p.Anonymous {
                     attrs[p.Name]=p.Type
       }
}

return attrs
}


type Declarante struct {
        Id        int
        Nombre string
        ApPaterno  string
        ApMaterno  string

}
```

Figura A.3.3. Segunda parte del programa de Reflection, tomando como base la estructura “declarante”.

6.3.2 Creación de template

Una vez dominados los conceptos de Reflection, se continuó con el desarrollo de un programa que genera los servicios de altas, bajas, cambios y consultas de cualquier estructura de manera automática. Esto se hizo con la finalidad de crear servicios REST (GET, PUT, POST y DELETE) para cualquier otro tipo de estructuras como podrían ser “personas”, “casas”, etc.

Para esto, se utilizó como base los servicios proporcionados en el archivo declaranetZacatecas.go. Entonces aplicando Reflection, se obtienen los atributos o propiedades (nombre, apellidos, etc.) y nombre de la estructura (en este caso declarante es el nombre de la estructura). La estructura puede contener propiedades

de cualquier tipo y de cualquier nombre, pero cabe aclarar que para que el servicio sea generado de manera correcta, la única condición es que la estructura siempre contenga una propiedad ID de tipo entera.

En el archivo template2.2.go, tenemos la ejecución de Reflection de la estructura “declarante”, una vez que obtenemos las propiedades y el nombre de la estructura, el siguiente paso es reemplazar estas propiedades y nombres en el archivo service2.2.tmpl.

Después en el archivo con extensión tmpl (que contiene la base de los servicios para altas, bajas, cambios y consultas) se hacen los reemplazos correspondientes, como lo son nombre del servicio e inicialización de valores de los campos, en el archivo service 2.2.tmpl que contiene la plantilla con los servicios REST, solo faltaría agregar el nombre de la estructura y los campos obtenidos mediante Reflection..

El código fuente del archivo **template2.2.go**, puede encontrarse en la dirección https://github.com/octavioreyes1/RestfulObjectsGO/blob/master/template2.2.go

A continuación se muestra el contenido del archivo template2.2.go

```
package main
import "text/template"
import "os"
import "reflect"
import "fmt"
import "strings"
```

```
// Estructura
type Item struct {
	Text string
}
func main() {
	de := reflect.TypeOf(Declarante{})
	var ClassName string = fmt.Sprintf("%s", de.Name())
	var className string = strings.ToLower(ClassName)
	var parts []string
	for name, mtype := range attributes(&Declarante{}) {
	if name != "Id" {
	parts = append(parts, fmt.Sprintf("%s %s\n", name, mtype.Name()))
	}
	}
	var campos string = strings.Join(parts, "")
	t, _ := template.ParseFiles("service2.2.tmpl")
	// field names don't have to be capitalized
	params := map[string]interface{}{"serviceName": "declaranet",
	"ServiceName": "Declaranet"}
	params["ServiceName"] = ClassName
	params["serviceName"] = className
	params["ClassName"] = ClassName
	params["className"] = className
	//********** CREAR CAMPOS DE LA ESTRUCTURA **********//
	params["campos"] = campos
	//********** INICIALIZAR VALORES **********//
	var partsini []string
	for i := 0; i < de.NumField(); i++ {
```

```
        field := de.Field(i)
        if field.Name != "Id" {
        partsini = append(partsini, fmt.Sprintf("%s: \"%s\" + strconv.Itoa(i),\n", field.Name, field.Name))
        }
        }
        var camposini string = strings.Join(partsini, "")
        sz := len(camposini)
        if sz > 0 && camposini[sz-2] == ',' {
        camposini = camposini[:sz-2]
        }
        camposini = camposini + "})"
        params["camposini"] = camposini
        // EJEMPLO
        //          Nombre: "Nombre" + strconv.Itoa(i),
        //          ApPaterno: "Apellido P." + strconv.Itoa(i),
        //          ApMaterno: "Apellido M." + strconv.Itoa(i)})
        //********* REEMPLAZAR VALORES ****************
        t.Execute(os.Stdout, params)
}
// REFLECTION
func attributes(m interface{}) map[string]reflect.Type {
        typ := reflect.TypeOf(m)
        if typ.Kind() == reflect.Ptr {
        typ = typ.Elem()
        }
        attrs := make(map[string]reflect.Type)
        if typ.Kind() != reflect.Struct {
        fmt.Printf("%w type cant have attributes inspected\n", typ.Kind())
```

```
        return attrs
        }
        for i := 0; i < typ.NumField(); i++ {
        p := typ.Field(i)
        if !p.Anonymous {
        attrs[p.Name] = p.Type
        }
        }
        return attrs
}
// ESTRUCTURA
type Declarante struct {
        Id      int
        Nombre string
        ApPaterno string
        ApMaterno string
        Rfc     string
}
var (
        declaranteStore []Declarante
)
```

En el archivo **service2.2.tmpl** solo se hacen los reemplazos correspondientes, por ejemplo, si el nombre de la estructura de entrada fuera “Auto”, en este archivo se realizaría el reemplazo de la cadena {{.ClassName}} por la palabra “Auto”, y en la cadena {{.campos}} se reemplazaría por el nombre de los campos y sus tipos de datos de la estructura “Auto” (por ejemplo marca: string, color: string, etc). A continuación se

muestra el contenido del programa con la plantilla que contiene la base de la implementación de los servicios.

El código fuente del archivo **service2.2.tmpl** se encuentra en la direcciónhttps://github.com/octavioreyes1/RestfulObjectsGO/blob/master/service2.2.tmpl.

```
package main
import (
	"code.google.com/p/gorest"
	"net/http"
	"strconv"
)
func main() {
	gorest.RegisterService(new(DeclaranteService))
	http.Handle("/", gorest.Handle())
	http.ListenAndServe(":8787", nil)
}
type {{.ClassName}}Service struct {
	//Service level config
	gorest.RestService `root:"/{{.className}}-service/" consumes:"application/json" produces:"application/json"`
	//End-Point level configs: Field names must be the same as the corresponding method names,
	// but not-exported (starts with lowercase)
	//deleteDeclarante gorest.EndPoint `method:"DELETE" path:"/{{.className}}s/del/{Id:int}"`
	view{{.ClassName}} gorest.EndPoint `method:"GET" path:"/{{.className}}s/{Id:int}" output:"{{.ClassName}}"`
```

```
	list{{.ClassName}}s gorest.EndPoint `method:"GET" path:"/{{.className}}s/" output:"[]{{.ClassName}}"`
	delete{{.ClassName}} gorest.EndPoint `method:"DELETE" path:"/{{.className}}s/{Id:int}"`
	add{{.ClassName}} gorest.EndPoint `method:"PUT" path:"/{{.className}}s/" postdata:"{{.ClassName}}"`
	mod{{.ClassName}} gorest.EndPoint `method:"POST" path:"/{{.className}}s/" postdata:"{{.ClassName}}"`
}
func (serv {{.ClassName}}Service) List{{.ClassName}}s() []{{.ClassName}} {
	serv.ResponseBuilder().CacheMaxAge(60 * 60 * 24) //List cacheable for a day. More work to come on this, Etag, etc
	return {{.className}}Store
}
func (serv {{.ClassName}}Service) View{{.ClassName}}(id int) (ret{{.ClassName}} {{.ClassName}}) {
	for _, {{.className}} := range {{.className}}Store {
	if {{.className}}.Id == id {
	ret{{.ClassName}} = {{.className}}
	return
	}
	}
	serv.ResponseBuilder().SetResponseCode(404).Overide(true)
	return
}
func (serv {{.ClassName}}Service) Delete{{.ClassName}}(id int) {
	for pos, {{.className}} := range {{.className}}Store {
	if {{.className}}.Id == id {
	{{.className}}Store[pos] = {{.className}}
	return //Default http code for DELETE is 200
```

```
		}
		}
		serv.ResponseBuilder().SetResponseCode(404).Overide(true)
		return
}
func (serv {{.ClassName}}Service) Add{{.ClassName}}(temp {{.ClassName}}) {
		//Item Id not in database, so create new
		temp.Id = len({{.className}}Store) + 1
		{{.className}}Store = append({{.className}}Store, temp)
		serv.ResponseBuilder().Created("http://localhost:8787/declarante-service/{{.className}}s/" +
string(temp.Id)) //Created, http 201
}
func (serv {{.ClassName}}Service) Mod{{.ClassName}}(temp {{.ClassName}}) {
		for pos, {{.className}} := range {{.className}}Store {
		if {{.className}}.Id == temp.Id {
		{{.className}}Store[pos] = temp
		serv.ResponseBuilder().SetResponseCode(200) //Updated http 200, or you could just return
without setting this. 200 is the default for POST
		return
		}
		}
}
//**************** End of service ******************
type {{.ClassName}} struct {
		Id int
		{{.campos}}
}
var (
```

```
        {{.className}}Store   []{{.ClassName}}
)
func init() {
        {{.className}}Store  = make([]{{.ClassName}},  0)
        init{{.ClassName}}s()
}
func init{{.ClassName}}s()  {
        for i := 1; i <= 10; i++ {
          {{.className}}Store = append({{.className}}Store,  {{.ClassName}}{Id:  i,
        {{.camposini}}
        }
}
```

Creación de archivo GO en base al template

Para crear nuestro archivo ejecutable con los servicios de altas, bajas, cambios y consultas se ejecuta el siguiente comando en modo consola:

```
#go run template2.2.go > misServicios.go
```

Lo que hace este comando es reemplazar los valores del template, por los valores de la estructura a la que se le haya aplicado Reflection. Una vez terminado el reemplazo, se envía el resultado a un archivo nuevo llamado misServicios.go.

6.4. PRUEBAS DE INTEGRACIÓN CON AROW

La segunda parte del desarrollo de este proyecto fue la integración con el cliente AROW. Una vez probada la generación de servicios GOREST, para distintas estructuras, se procedió con el inicio de la integración de los servicios con AROW.

Seguridad de puertos del servidor

Cuando se generan los servicios, se crean en el puerto 8787, pero por cuestiones de seguridad, no se puede acceder a dichos servicios de manera directa por el puerto 80 (Apache), por lo tanto hay dos alternativas.

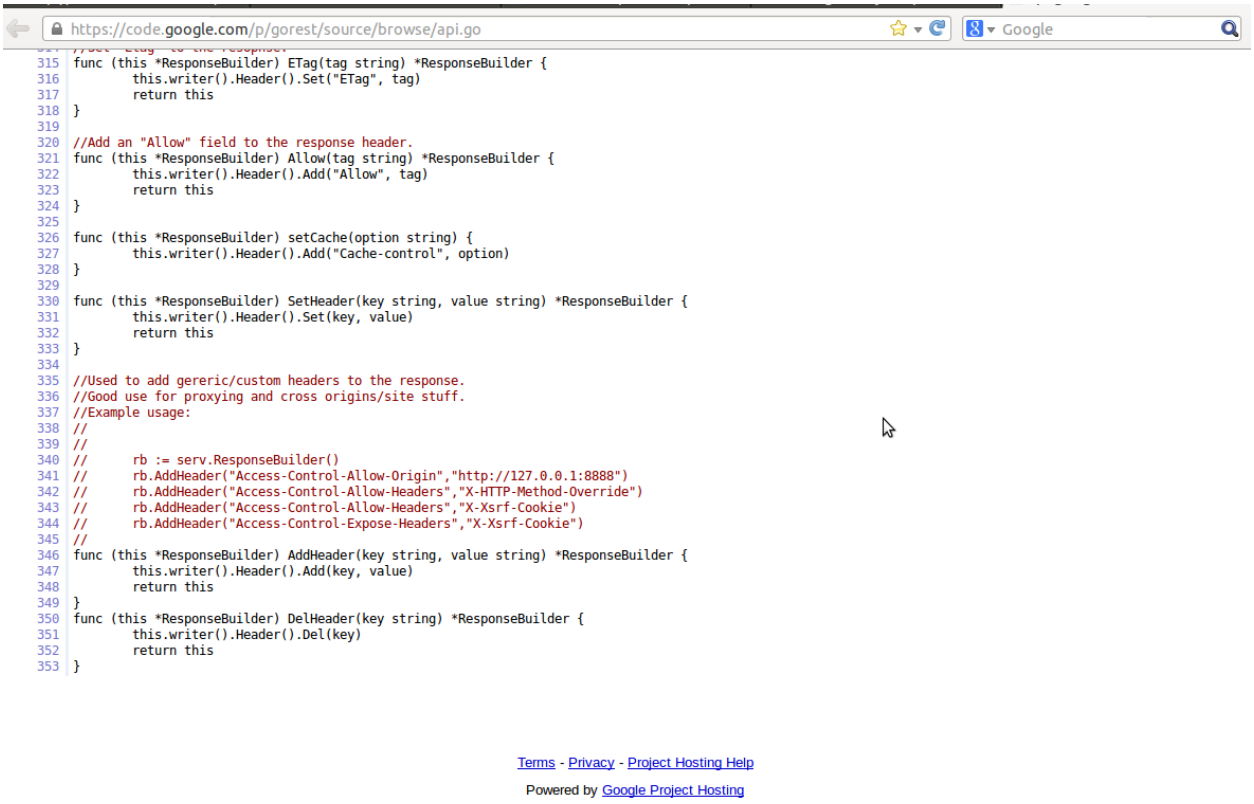

Figura A.4.1. Headers comentarizados en GOREST.

La primera opción se considera la más sencilla, y consiste en agregar “headers”, en nuestro archivo GO, indicando la IP del servidor o con “*” indicamos que cualquier servidor pueden acceder a los servicios de GOREST.

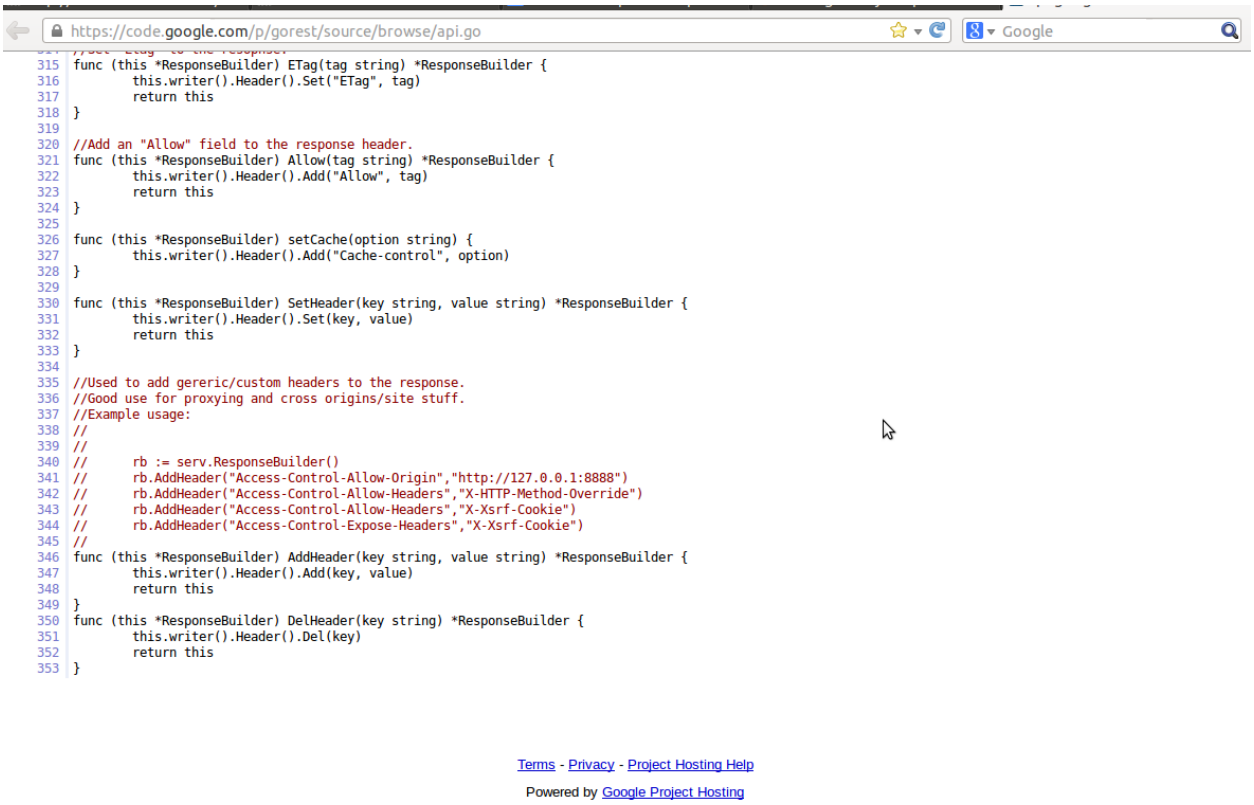

Figura A.4.2. Ejemplo del uso de headers en la página oficial de GOREST

La dirección de la página oficial de GOREST es:

https://code.google.com/p/gorest/source/browse/api.go

La segunda opción es la de modificar la configuración del servidor de Apache, mediante la adición de un Virtual Host.

Para modificar el archivo, primero hay que asegurarse que el mod_proxy está activado. Se tienen que activar dos módulos que son: proxy y proxy_http usando a2enmod. Para esto ejecutamos en consola:

root@server# a2enmod proxy

Module proxy installed; run /etc/init.d/apache2 force-reload to enable.

root@server# a2enmod proxy_http

Enabling proxy as a dependency

This module is already enabled!

Module proxy_http installed; run /etc/init.d/apache2 force-reload to enable.

Una vez ejecutados con éxito los comandos anteriores, se agrega un nuevo virtual host con una configuración similar a la que se muestra a continuación.

<VirtualHost *:80>

ServerAdmin webmastergo@example.com

ServerName www.examplego.com

ServerAlias examplego.com

DocumentRoot /var/www/

ProxyPass /go/ http://localhost:8787/

<Directory />

Options FollowSymLinks

AllowOverride None

</Directory>

<Directory /var/www/>

Options Indexes FollowSymLinks MultiViews

AllowOverride None

Order allow,deny

allow from all

</Directory>

</VirtualHost>

Con la línea ProxyPass indicamos que cada vez que se teclea http://localhost/go se hace una redirección a http://localhost:8787, que es la dirección donde se encuentran los servicios GOREST.

Es posible utilizar cualquiera de las dos alternativas o también las dos opciones al mismo tiempo, es decir, el proxy de un nuevo Virtual Host y además los “Headers” dentro de nuestro programa GO indicando los permisos de los servidores que pueden consumir los servicios. El archivo declaranetZacatecasPermisos.go con los headers integrados se encuentra en la dirección:

https://github.com/octavioreyes1/RestfulObjectsGO/blob/master/declaranetZacatecasPermisos.go

Cliente AROW

Una vez que se tiene acceso a los servicios de GOREST. Se procedió a descargar el cliente de AROW, en un principio se asumió que este cliente debería de

cargar las librerías de los servicios web, solamente indicando la página de inicio de los servicios en el archivo *AROW.html* como se muestra en la siguiente figura:

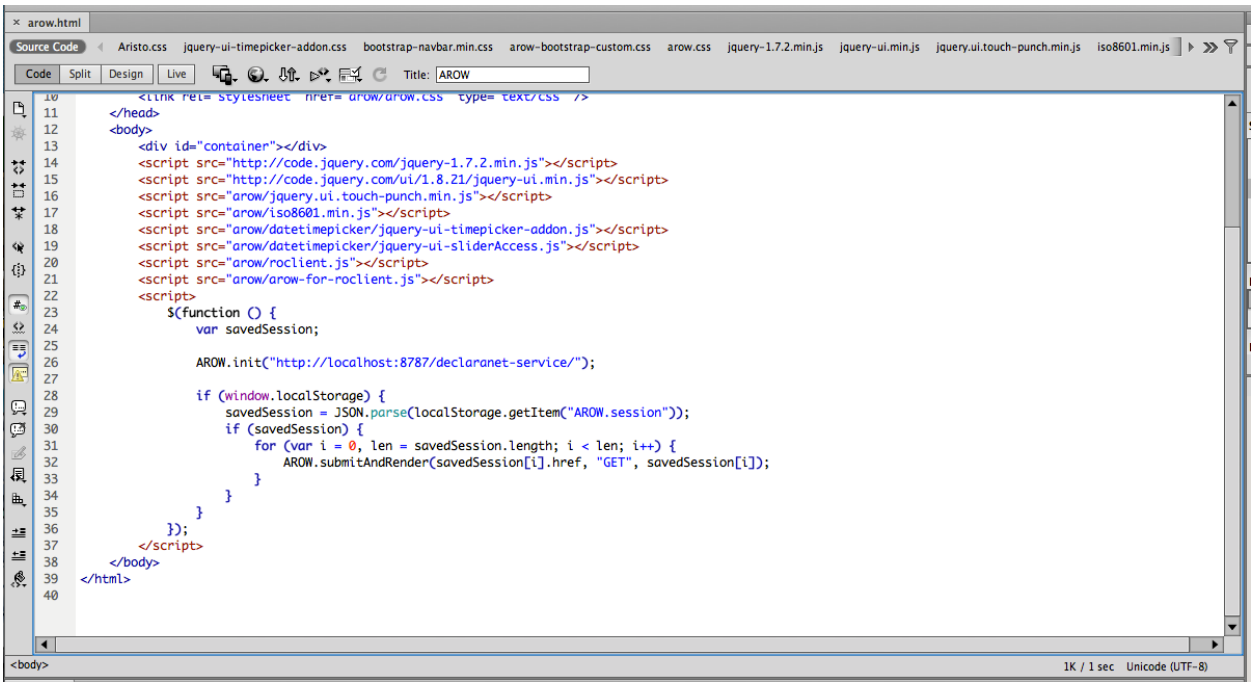

Figura A.4.3. Código fuente de la página principal de AROW.

El código fuente de AROW, está realizado en HTML con JQuery, por lo que para instalarlo lo único que necesitamos es contar con el servicio HTTP de Apache.

Para instalar AROW, solo hay que bajar el archivo con la extensión .zip de GitHub (https://github.com/adamhoward/AROW), se descomprime la carpeta y se copia en la raíz del www del servicio Apache.

El único cambio realizado en el código fuente de AROW fue agregar la ruta del servidor GO en la página de inicio, que es el archivo arow.html. La línea de código quedó de la siguiente forma:

AROW.init (“http://localhost:8787/declaranet-service/”);

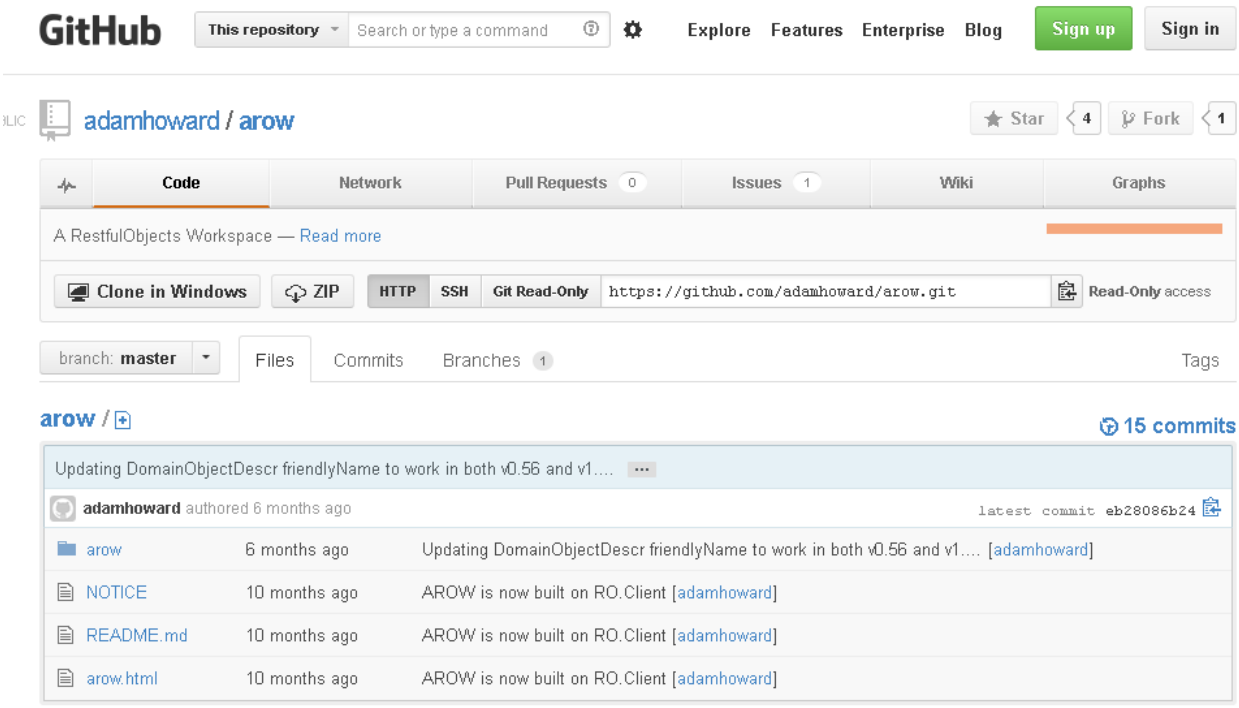

Figura A.4.4. Repositorio Github de AROW.

El primer problema que se tuvo con el cliente, fue el de los permisos que necesita la librería de jQuery para consumir los recursos del puerto 8787, que es el puerto utilizado por el servicio declaranet-service. Después de probar con varias alternativas, se concluyó que la manera más óptima de tener permiso para consumir los servicios, fue agregar a cada uno de los métodos del servicio web las siguientes líneas de código:

```
func (serv DeclaranetService)  Servicios()(j  Json){
rb:= serv.ResponseBuilder()
rb.AddHeader("Access-Control-Allow-Origin",  "*")
    rb.AddHeader("Access-Control-Allow-Headers",  "X-HTTP-Method-Override")
    rb.AddHeader("Access-Control-Allow-Headers",  "X-Xsrf-Cookie")
    rb.AddHeader("Access-Control-Expose-Headers",  "X-Xsrf-Cookie").
```

Con el código anterior lo que hace GO antes de montar el servicio, es dar permisos (señalados mediante el signo de “*”) para usar el servicio web.

Una vez asignados los permisos en todos los servicios, se presentó un problema importante en el uso de AROW como cliente, dado que al momento de consumir el servicio web, el navegador de internet (en modo desarrollador) generaba varios errores como se muestra a continuación:

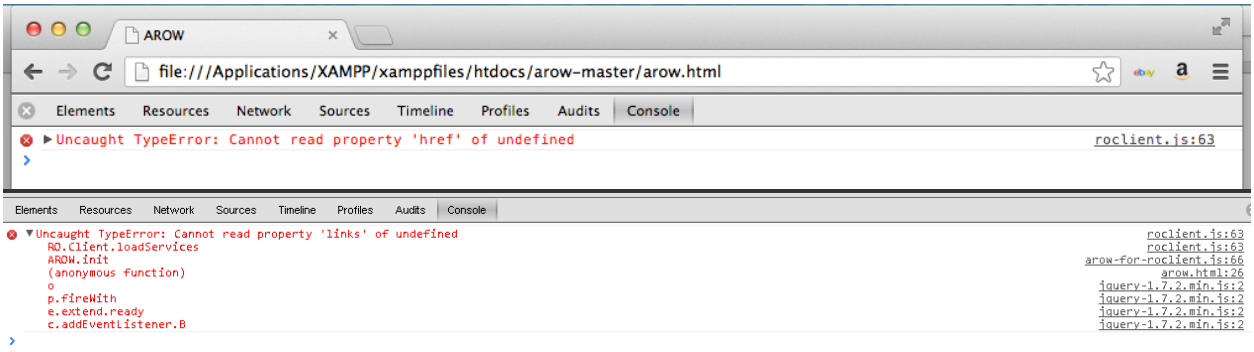

Figura A.4.5. Errores al ejecutar AROW.

Por esta razón se decidió obtener y copiar la estructura de los servicios web solicitados por el cliente de AROW, después de realizar varias pruebas, se observó que el servicio web respondía con varios JSON encode, para lo cual se procedió a elaborar dichos servicios y agregarlos al servicio declaranet-service, para que respondiera de manera similar a la solicitada por AROW, agregando los siguientes códigos:

```
func (serv DeclaranetService) Self()(j Json){
        rb:= serv.ResponseBuilder()
  rb.AddHeader("Access-Control-Allow-Origin", "*")
```

```
   rb.AddHeader("Access-Control-Allow-Headers", "X-HTTP-Method-Override")
   rb.AddHeader("Access-Control-Allow-Headers", "X-Xsrf-Cookie")
   rb.AddHeader("Access-Control-Expose-Headers", "X-Xsrf-Cookie")
        link1 := Link{Rel:"self", Method:"GET", Type:"application/json; profile=\"urn:org.restfulobjects:
repr-types/homepage\"; charset=utf-8"}
        link2 := Link{Rel:"user", Method:"GET",
Type:"application/json;profile=\"urn:org.restfulobjects/homepage\"", Href:"http://localhost:8787/declaranet-
service/user"}
        link3 := Link{Rel:"services", Method:"GET",
Type:"application/json;profile=\"urn:org.restfulobjects/list\"", Href:"http://localhost:8787/declaranet-
service/services"}
        link4 := Link{Rel:"version", Method:"GET",
Type:"application/json;profile=\"urn:org.restfulobjects/version\"", Href:"http://localhost:8787/declaranet-
service/version"}
        link10 := Link{Rel:"urn:orgp.restfulobjects:rels/declarante", Method:"GET", Type:"application/json;
profile=\"urn:org.restfulobjects: repr-types/declarante\"; charset=utf-8",
Href:"http://localhost:8787/declaranet-service/declarantes"}
        linkStore := []Link{link1, link2, link3, link4, link10}
        n := Json{Links:linkStore, Extensions:Ext{}}
// Procesamiento de ejemplo
        y := 2
        if (y==2){
        j=n
        return
        }
        j = n
   serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
```

}

De esta manera el servicio web logró responder en la raíz del sitio con los datos necesarios que fueron solicitados por AROW, lo cual se puede ver cuando navegamos en la siguiente dirección:

http://localhost:8787/declaranet-service/

Figura A.4.6. Servicio “declaranet-service”.

Al parecer con este cambio, se avanzó bastante pero el cliente de AROW, siguió solitando otros servicios para cuatro estructuras principales que son: Services, Users, DomainTypes y Version, por lo cual se agregaron las siguientes funciones:

DOMAINTYPES

```
func (serv DeclaranetService) Domain()(j Jsondomain){
        rb:= serv.ResponseBuilder()
  rb.AddHeader("Access-Control-Allow-Origin", "*")
  rb.AddHeader("Access-Control-Allow-Headers", "X-HTTP-Method-Override")
  rb.AddHeader("Access-Control-Allow-Headers", "X-Xsrf-Cookie")
  rb.AddHeader("Access-Control-Expose-Headers", "X-Xsrf-Cookie")
        link1 := Link{Rel:"self",Method:"GET",
Type:"application/json;profile=\"urn:org.restfulobjects/typelist\"", Href:"http://localhost:8787/declaranet-
service/domainTypes"}
        linkStore := []Link{link1}
```

```
        n := Jsondomain{Links:linkStore,  Values:linkStore, Extensions:Ext{}}
// Procesamiento sencillo de ejemplo
        y  := 2
        if (y==2){
        j=n
        return
        }
        j = n
        serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
}
```

De esta manera cuando se navega por el servicio en la página web:

http://localhost:8787/declaranet-service/domainTypes

El navegador entonces muestra el siguiente resultado:

Figura A.4.7. Servicio “domainTypes”

SERVICES:

```
func (serv DeclaranetService)  Servicios()(j  Json){
        rb:= serv.ResponseBuilder()
    rb.AddHeader("Access-Control-Allow-Origin",  "*")
    rb.AddHeader("Access-Control-Allow-Headers",  "X-HTTP-Method-Override")
    rb.AddHeader("Access-Control-Allow-Headers",  "X-Xsrf-Cookie")
    rb.AddHeader("Access-Control-Expose-Headers",  "X-Xsrf-Cookie")
```

```
link1 := Link{Rel:"self",Method:"GET", Type:"application/json; profile=\"urn:org.restfulobjects:repr-types/list\"; charset=utf-8; x-ro-element-type=\"System.Object\"", Href:"http://localhost:8787/declaranet-service/services"}
        link2 := Link{Rel:"up",Method:"GET", Type:"application/json; profile=\"urn:org.restfulobjects:repr-types/homepage\"; charset=utf-8", Href:"http://localhost:8787/declaranet-service/"}
        linkStore := []Link{link1, link2}
        n := Json{Links:linkStore, Extensions:Ext{}}
// Procesamiento sencillo de ejemplo
        y := 2

        if (y==2){
        j=n
         return
        }
        j = n
    serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
}
```

Una vez agregado el servicio podemos ver la respuesta en la URL:

http://localhost:8787/declaranet-service/services

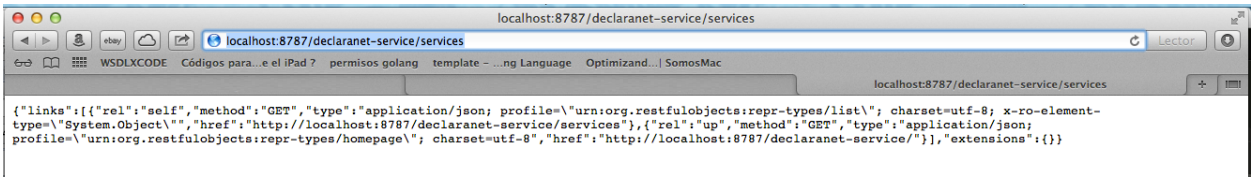

Figura A.4.8. Servicio “Services”.

VERSION

```
func (serv DeclaranetService) Version()(j Jsonver){
        rb:= serv.ResponseBuilder()
```

```
    rb.AddHeader("Access-Control-Allow-Origin", "*")
    rb.AddHeader("Access-Control-Allow-Headers", "X-HTTP-Method-Override")
    rb.AddHeader("Access-Control-Allow-Headers", "X-Xsrf-Cookie")
    rb.AddHeader("Access-Control-Expose-Headers", "X-Xsrf-Cookie")
        link1 := Link{Rel:"self",Method:"GET",
Type:"application/json;profile=\"urn:org.restfulobjects/version\"", Href:"http://localhost:8787/declaranet-
service/version"}
        linkStore := []Link{link1}
        n := Jsonver{Links:linkStore, Specversion:"1.0", ImplVersion:"1.1.0", Extensions:Ext{}}
        y := 2
        if (y==2){
        j=n
        return
        }
    j = n
    serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
}
```

Una vez adicionado el servicio, podemos ver el resultado en la URL:

http://localhost:8787/declaranet-service/version

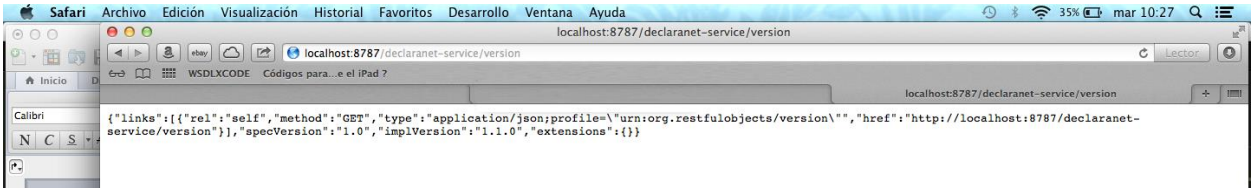

Figura A.4.9. Servicio “version”.

USERS

```
func (serv DeclaranetService) Usuarios()(j Jsonuser){
        rb:= serv.ResponseBuilder()
```

```
    rb.AddHeader("Access-Control-Allow-Origin", "*")
    rb.AddHeader("Access-Control-Allow-Headers", "X-HTTP-Method-Override")
    rb.AddHeader("Access-Control-Allow-Headers", "X-Xsrf-Cookie")
    rb.AddHeader("Access-Control-Expose-Headers", "X-Xsrf-Cookie")
        link1 := Link{Rel:"self",Method:"GET", Type:"application/json; profile=\"urn:org.restfulobjects:repr-types/user\"; charset=utf-8", Href:"http://localhost:8787/declaranet-service/user"}
        link2 := Link{Rel:"up",Method:"GET", Type:"application/json; profile=\"urn:org.restfulobjects:repr-types/homepage\"; charset=utf-8", Href:"http://localhost:8787/declaranet-service/"}
        linkStore := []Link{link1, link2}
    rol:=[]Rol{}
        n := Jsonuser{Links:linkStore, Extensions:Ext{}, UserName:"", Roles:rol}
// Procesamiento sencillo de ejemplo
        y := 2
        if (y==2){
        j=n
        return
        }
        j = n
    serv.ResponseBuilder().SetResponseCode(404).Overide(true)
        return
}
```

Este nuevo servicio se puede ver en la URL:

http://localhost:8787/declaranet-service/users

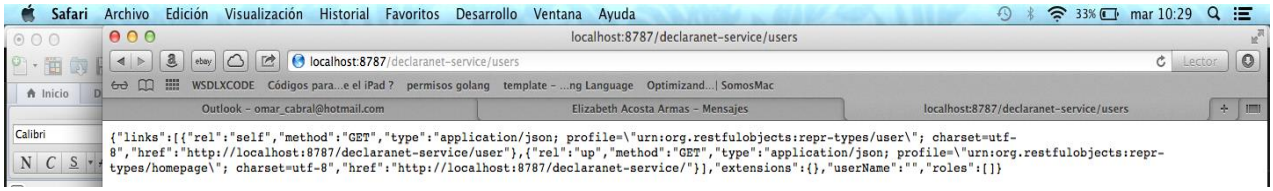

Figura A.4.10. Nuevo servicio “users”

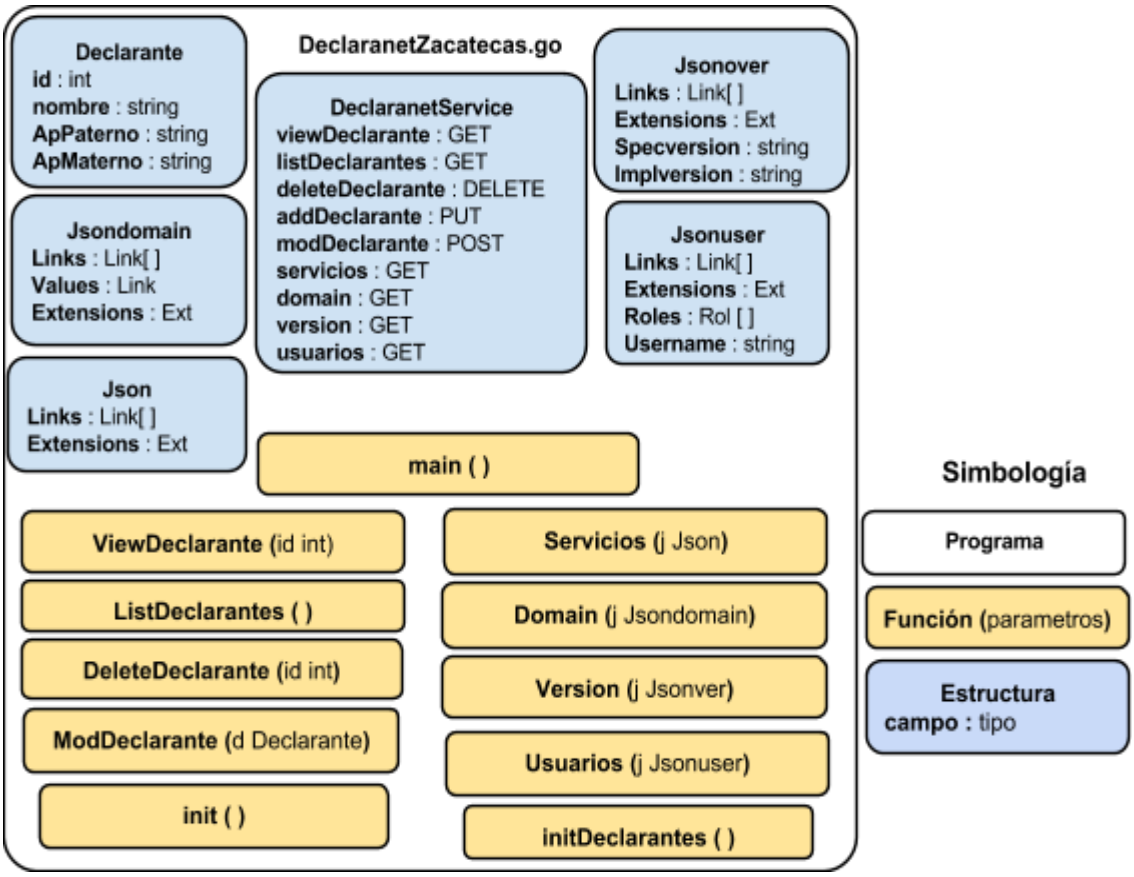

Figura A.4.11. Nuevo diagrama general de programa declaranetZacatecas.go

En la figura anterior se muestra un diagrama del proyecto y las funciones, estructuras y métodos con las cuales se pretendía que funcionara el cliente de AROW, es decir, hacer algo similar al cliente de AROW, que exponen en el ejemplo de Github. Para lograr dichos objetivos se agregaron nuevas estructuras, que se adecuaron para que el servicio proporcionara la salida correspondiente en JSON.

Las funciones agregadas al proyecto fueron:

* Servicios (Services) para describir los servicios expuestos por el servicio

* Usuarios (Users) para el uso de usuarios y roles

* Domain (DomainTypes) para definir el dominio

* Version define las versiones del servicio

Las funciones descritas anteriormente utilizan las siguientes estructuras para que puedan obtener una salida en pantalla adecuada de JSON encode:

* Link

* Rol

* Ext

* Jsondomain

* Json

* Jsonover

* Jsonuser

El código fuente del archivo GO, con las estructuras y funciones mencionadas se encuentra en la dirección:

https://github.com/octavioreyes1/RestfulObjectsGO/blob/master/declaranetZacatecasPermisosAROW.go

Un ejemplo de esto sería cuando el usuario teclee la urlhttp://localhost:8787/declaranet-service/users, aparecerá en el navegador:

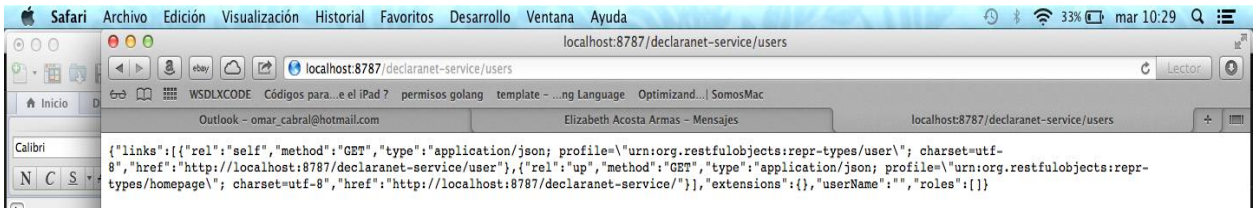

Figura A.4.12. Servicio “users”.

Después de ver el funcionamiento de AROW, haciendo reingeniería con los datos solicitados por el cliente AROW, se finalizó la codificación de los servicios solicitados por el cliente de AROW, por lo tanto ya no se mostraron tantos errores como cuando se indicaba solamente el servicio raíz de declaranet-service, sin los nuevos servicios relacionados con las cuatro estructuras principales (Users, Services, DomainTypes y Version) , pero seguía mostrando el siguiente error al momento de llegar a la parte de Version:

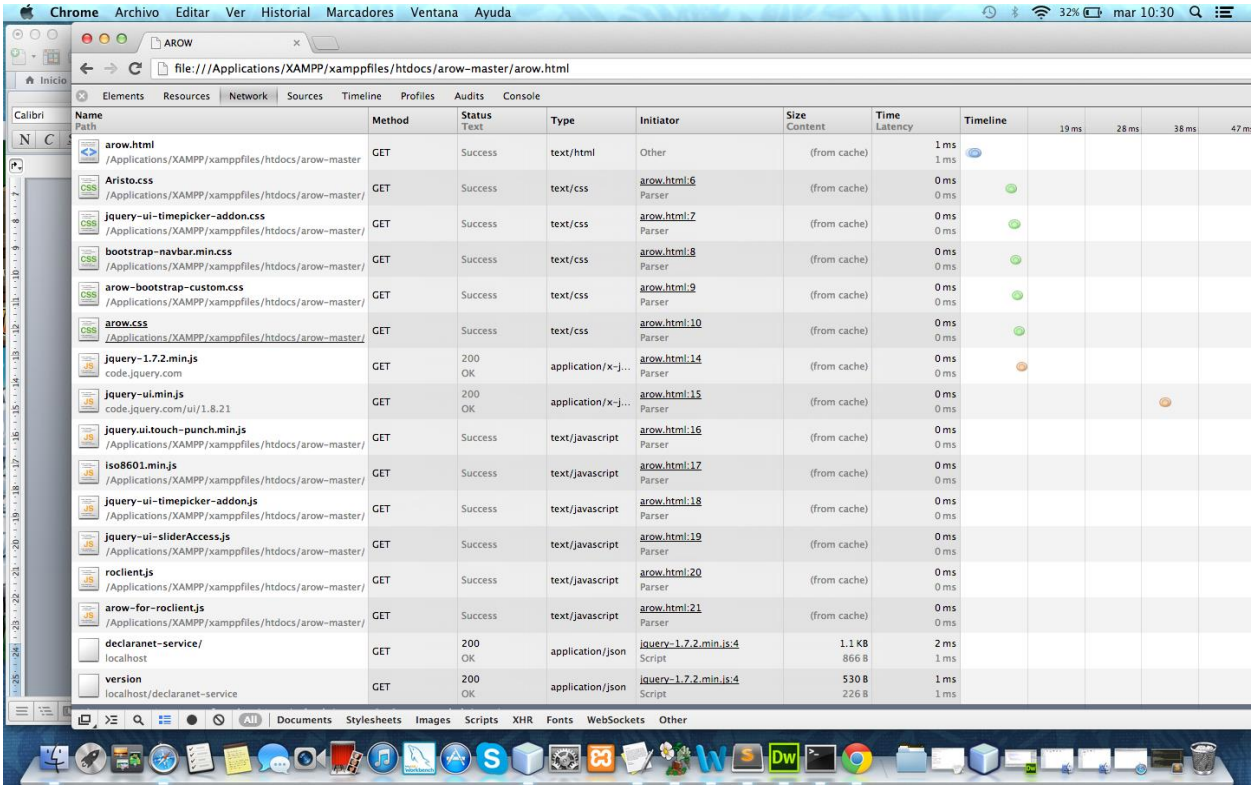

Figura A.4.13. Ejecución de AROW mostrando el seguimiento de operaciones ejecutadas

Después de varios intentos por seguir cumpliendo con los servicios solicitados, se encontró que la cantidad de servicios era muy grande, por ejemplo en la versión de

.NET se encontraron alrededor de 350 servicios específicamente solicitados por AROW.

De esta manera se puede concluir que el cliente de AROW, todavía se encuentra en desarrollo y no sigue un estándar universal de servicios REST, debido a que se tienen que agregar bastantes métodos y servicios. También existen dos versiones de AROW que son 0.55 y 1.0 pero no siguen el mismo estándar, incluso la versión de .NET (1.0) responde con más opciones y objetos o estructuras que la versión de Apache Isis (0.55).

Por lo que se requiere de demasiado tiempo para poder adaptar servicios básicos, para que puedan ser consumidos por el cliente AROW, cuya función principal es adaptarse a un servicio y mostrar de manera gráfica y amigable los servicios disponibles.

Al entrar a las páginas oficiales de las dos implementaciones de Servidor de Restful Objects, se puede ver que el funcionamiento es correcto en los clientes AROW (Figura A.4.14). Sin embargo, el principal problema es que no hay documentación completa de los requerimientos necesarios para crear un nuevo Servidor Restful Objects.

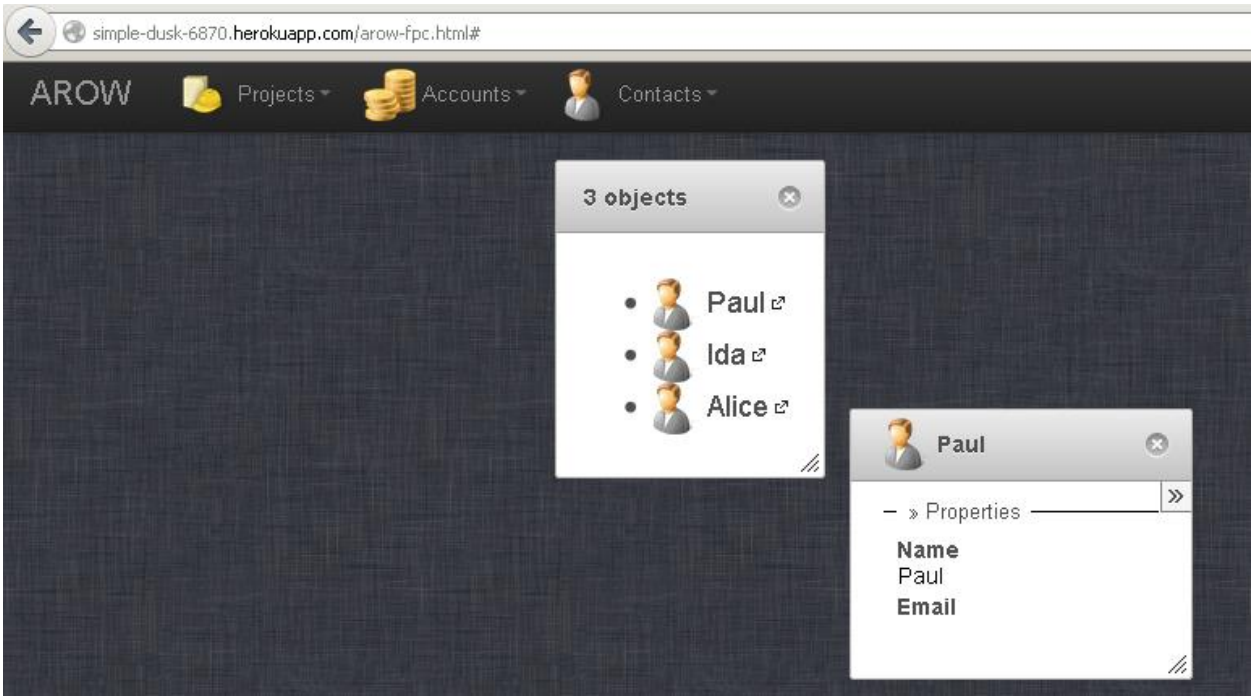

Figura A.4.14. Funcionamiento correcto de AROW con Apache ISIS.

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS, A.C.

BIBLIOTECA

AUTORIZACION
PUBLICACION EN FORMATO ELECTRONICO DE TESIS

El que suscribe

Autor(s) de la tesis: Octavio Reyes Pinedo

Título de la tesis: Caso de implementación de un servidor Restful Objects en GO

Institución y Lugar:

Grado Académico: Licenciatura ( ) Maestría (X) Doctorado ( ) Otro ( )

Año de presentación: 2014

Área de Especialidad: Maestría en Ingeniería de Software

Director(es) de Tesis: Alejandro García Fernández - José Guadalupe Hernández Reveles

Correo electrónico: octavioreyes1@hotmail.com

Domicilio: C. Zacatecas #9, Colonia Centro
Morelos, Zac. CP 98100

Palabra(s) Clave(s): NaKed Objects, Restful Objects, Lenguaje GO, Reflection, ARow

Por medio del presente documento autorizo en forma gratuita a que la Tesis arriba citada sea divulgada y reproducida para publicarla mediante almacenamiento electrónico que permita acceso al público a leerla y conocerla visualmente, así como a comunicarla públicamente en la Página WEB del CIMAT.

La vigencia de la presente autorización es por un periodo de 3 años a partir de la firma de presente instrumento, quedando en el entendido de que dicho plazo podrá prorrogar automáticamente por periodos iguales, si durante dicho tiempo no se revoca la autorización por escrito con acuse de recibo de parte de alguna autoridad del CIMAT

La única contraprestación que condiciona la presente autorización es la del reconocimiento del nombre del autor en la publicación que se haga de la misma.

Atentamente

Octavio Reyes Pinedo

Nombre y firma del tesista

CALLE JALISCO S/N MINERAL DE VALENCIANA APDO. POSTAL 402
C.P. 36240 GUANAJUATO, GTO., MÉXICO
TELÉFONOS (473) 732 7155, (473) 735 0800 EXT. 49609 FAX. (473) 732 5749
E-mail: biblioteca@cimat.mx