Proyecto Fin de Carrera

API de servicios web REST extensible dinámicamente basada en una arquitectura Plug-in para el procesado de imágenes recibidas desde dispositivos móviles

Autor:
José Fortes Alayón

Director:
José Luis Sánchez de la Rosa

Titulación:
Ingeniero en Informática

Escuela Técnica Superior de Ingeniería Informática
Universidad de La Laguna

16 de mayo de 2016

D. José Luis Sánchez de la Rosa, con NIF 42.080.654–S, profesor del departamento Ingeniería Informática y de Sistemas de la Universidad de La Laguna

CERTIFICA

Que la presente memoria titulada:

API de Servicios Web REST Modular y Extensible basada en una Arquitectura de Plug In para el Procesado de Imágenes Recibidas desde Dispositivos Móviles

ha sido realizada bajo su dirección por D. José Fortes Alayón con NIF. 78.699.823-X.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firma la presente en La Laguna a 3 de Junio de 2016

Agradecimientos

A mi mujer, Ofelia, por entenderme, aconsejarme y apoyarme en todo momento, eso lo hace todo mucho más fácil.

A mi encantadora hija Sofía, que es la alegría de mi vida con su cariño, su gracia y su perspicacia.

A mi hijo César, al que pronto tendremos con nosotros para aumentar esa alegría en la familia.

A mi madre, a mi tía Delia y a mis hermanos, Carlos y Eduardo, por estar siempre ahí.

A José Luis Sánchez de la Rosa, el Director de este proyecto, que me lanzó un guante que no podía dejar pasar y que me ha acompañado con su ayuda en este trayecto.

Hay que tener en cuenta el hecho impactante de que vivimos en un mundo que gira. Después de considerar esta verdad, nada puede ser una sorpresa.

Thomas Ligotti

Como quien somos cumplimos.

José Zorrilla, *Don Juan Tenorio*

Venid, amigos míos.
No es demasiado tarde para buscar un mundo nuevo.
Zarpemos, y sentados en perfecto orden
golpeamos los resonantes surcos,
pues me propongo navegar más allá del poniente
y el lugar en que se bañan todos los astros del occidente,
hasta que muera.
Es posible que las corrientes nos hundan y destruyan;
es posible que demos con las Islas Venturosas,
y veamos al gran Aquiles, a quien conocimos.
A pesar de que mucho se ha perdido, queda mucho;
y, a pesar de que no tenemos ahora el vigor
que antaño movía la tierra y los cielos,
lo que somos, somos:
un espíritu ecuánime de corazones heroicos,
debilitados por el tiempo y el destino, pero con una
voluntad decidida
a esforzarse, buscar, encontrar y no ceder.

Alfred Lord Tennyson, *Ulises*

Índice de Contenido

Índice de Figuras

1 Resumen

El presente Proyecto de Fin de Carrera aborda una problemática no poco común en el escenario tecnológico actual que se origina al intentar resolver problemas que requieren cálculo intensivo y algoritmia compleja desde dispositivos móviles. Estos dispositivos, por sus propias características de hardware, no son idóneos para realizar procesamientos exigentes en términos de procesador y memoria pero, sin embargo, sí son ideales para proveer a las personas de aplicaciones de gran utilidad, aportando un valor añadido muy relacionado con el hábito de uso por el que las personas llevan prácticamente siempre el dispositivo consigo, así como debido a los periféricos integrados en este (cámara, GPS, etc.).

En el caso que atañe al proyecto, se ha diseñado e implementado una solución que actúa como contraparte de servidor de una aplicación móvil que permita a cualquier persona examinar el riesgo de melanoma en su piel mediante el análisis de una foto de una zona sospechosa tomada con el móvil. La conveniencia del dispositivo móvil y el hecho de estar equipado con cámara y conexión de datos hacen de este dispositivo el instrumento ideal para popularizar este tipo de análisis, constituyendo un paso previo a la visita a un especialista. Sin embargo, la complejidad del procesamiento de imágenes puede requerir más recursos que el que típicamente pueden proveer los dispositivos móviles. Esta situación aconseja, por tanto, en un esquema cliente – servidor en el que el cliente provee de la interacción con el usuario y el servidor de la disponibilidad del servicio y de la potencia de procesamiento.

Uno de los objetivos del proyecto ha sido la interoperatividad total entre sistemas y tecnologías, con lo que se ha elegido el protocolo estándar HTTP como medio de comunicación entre cliente y servidor. Por esta misma razón, el cliente móvil se ha simulado con una herramienta independiente de la tecnología y que trabaja sobre el estándar HTTP. Por su parte, la aplicación de servidor ofrece una API de servicios web REST para la comunicación asíncrona con el cliente móvil y, además, ofrece un diseño extensible dinámicamente, mediante una arquitectura modular apoyada sobre el patrón de diseño de software Plug-in, de manera que los algoritmos de procesamiento de imagen puedan ser conectados a la aplicación de servidor en tiempo de ejecución sin cambios en el código fuente.

2 Introducción

2.1 Motivación y Objetivos

Hoy en día la penetración de los dispositivos móviles en la sociedad es extremadamente alta [1], lo que hace que estos dispositivos se hayan convertido en fuente de aplicaciones muy útiles para sus propietarios. Además, las posibilidades de comunicación mediante Internet, así como los periféricos integrados en los mismos, tales como cámaras, GPS, acelerómetros, etc., hacen de estos dispositivos auténticos laboratorios caseros móviles si se complementan con el software adecuado.

La mayor parte de estas aplicaciones forman parte de un sistema más complejo formado por una solución cliente – servidor, siendo el cliente la aplicación móvil que ejecuta un usuario en su dispositivo y el servidor una aplicación que proporciona servicios clave para la aplicación como son: procesamiento y almacenamiento de datos en una base de datos central, comunicación con otros usuarios, mantenimiento de la sesión de un usuario dentro de la aplicación, autenticación y seguridad a la hora de acceder a los datos propios o de otros usuarios, etc. Por tanto, las aplicaciones móviles (cliente) constituyen la parte en contacto directo con el usuario y las aplicaciones de servidor la parte de funcionalidades y servicios horizontales centralizados. Estas dos partes del sistema se conocen en Ingeniería del Software como Front End y Back End [2] respectivamente.

Teniendo en cuenta este contexto y bajo la premisa de utilidad que proporcionan las aplicaciones móviles hoy en día, nace la idea de aportar a los usuarios, precisamente, una suerte de laboratorio móvil para el análisis y detección de riesgo de posibles dolencias de la piel. Esta utilidad se entregaría mediante una aplicación móvil que permitiría al usuario tomar una fotografía de determinada zona de la piel y enviarla a un servidor central para ser analizada mediante técnicas de procesamiento de imagen y algoritmia específica. Este análisis daría como resultado un diagnóstico precoz e inmediato, informando al usuario respecto a la detección de riesgo o no de dolencia en la piel.

La primera parte de este sistema ya fue abordado por el proyecto de fin de carrera “Sistema de Seguimiento de la Evolución de Lunares para Prevención de Melanomas Basado en Smartphones” [3], que dio como resultado el diseño y la implementación de una aplicación móvil para Android. Esta aplicación es capaz, grosso modo, de tomar una fotografía y enviarla a un servidor central para su

análisis, informando de nuevo al usuario del riesgo de dolencia detectado en la misma tras la recepción del diagnóstico realizado en el servidor.

El proyecto de fin de carrera descrito en esta memoria aborda el segundo componente de este sistema: la aplicación de servidor que recibe una imagen desde un cliente (aplicación móvil), la procesa mediante algoritmos específicos para la detección de dolencias en la piel y devuelve una respuesta al cliente indicando el resultado del diagnóstico.

Los requisitos clave que debe cumplir esta aplicación de servidor son:

- **Interoperatividad:** dado el ecosistema disperso de tecnologías móviles imperante, normalmente una aplicación móvil es desarrollada para varias plataformas, como iOS, Android, Windows Phone, BlackBerry OS, etc., de manera que pueda llegar al mayor número de usuarios posible. Esto obliga a que el Back End tenga que poder comunicarse con Front Ends desarrollados en diferentes tecnologías y, además, debe hacerlo de forma transparente. De esta manera se reutiliza la lógica del Back End y no se necesita desarrollar uno específico para cada Front End. La única manera de conseguir esta interoperatividad es trabajando con protocolos, tecnologías e interfaces estándar y no dependientes de una tecnología concreta.
- **Extensibilidad dinámica:** debe permitirse que la aplicación de servidor sea extendida por futuro proyectos mediante la adición de algoritmos específicos de análisis de imagen para la detección de riesgo de dolencias en la piel a partir de una fotografía. Esta capacidad de extensión idealmente será dinámica, no siendo necesario recompilar el código del Back End para poder extenderlo.
- **Optimización de las comunicaciones:** en el diseño de una solución como la que se plantea es fundamental minimizar la cantidad de datos que se intercambian entre el Front End y el Back End, ya que el consumo de datos está tarificado por los operadores móviles y comportan un gasto económico para los usuarios en la mayoría de los casos. Además, la optimización de las comunicaciones también redunda en una mayor rapidez en la interacción cliente – servidor, lo cual proporciona una mejor experiencia al usuario, siendo este uno de los factores críticos por los que se utilizan o se desechan aplicaciones.

El objetivo del proyecto es, por tanto, proveer de una aplicación de servidor totalmente interoperable, cuya arquitectura permita la extensión dinámica para incorporar diferentes algoritmos de procesamiento de imagen sin necesidad de recompilaciones de código fuente, así como optimice el intercambio de datos entre el cliente y el servidor.

Para que el sistema esté completo y constituya una solución extremo a extremo que dé respuesta al escenario de laboratorio móvil completo planteando, será necesario otro proyecto de fin de carrera cuyo objetivo sea, exclusivamente, el trabajo de la algoritmia específica para análisis de las imágenes. Dadas las posibilidades y extensión de esta temática específica, se ha considerado que este particular será mejor resuelto en un proyecto que se centre exclusivamente en ella.

A lo largo de esta memoria se mostrarán las decisiones de diseño y los detalles de implementación de la aplicación de servidor aportada a la solución global. Dicha solución persigue las capacidades mencionadas anteriormente. Para poner en contexto la solución aportada, se repasará el estado del arte actual de los conceptos y abstracciones en Ingeniería del Software, así como de los enfoques tecnológicos existentes que podrían ser útiles en la resolución del problema. Asimismo, también se analizará el estado del arte de las aplicaciones médicas basadas en un escenario cliente – servidor para conocer qué soluciones y decisiones de diseño se han adoptado ante problemáticas similares. Finalmente, se detallará el diseño y la implementación de la aplicación que se ha desarrollado.

2.2 Organización de la Memoria

La memoria está organizada en los siguientes capítulos:

- **Capítulo 1. Resumen:** Breve descripción del proyecto.
- **Capítulo 2. Introducción:** Introducción, motivación y objetivos del proyecto.
- **Capítulo 3. Conceptos Previos:** se introducen conceptos y abstracciones que se utilizarán en el proyecto.
- **Capítulo 4. Problema:** Se describe el problema a resolver.
- **Capítulo 5. Solución Implementada:** Describe el diseño e implementación de la solución al problema.

- **Capítulo 6. Evaluación y Resultados:** Se muestran los principales resultados obtenidos en la ejecución y pruebas realizadas.
- **Capítulo 7. Líneas Futuras:** Se apuntan líneas de extensión posibles del proyecto.
- **Capítulo 8. Conclusiones:** Tras el análisis de los resultados se detallan las conclusiones obtenidas del mismo y se reflexiona sobre la consecución del objetivo del proyecto.
- **Capítulo 9. Glosario.**
- **Capítulo 10. Referencias.**
- **Capítulo 11. Bibliografía.**

3 Conceptos Previos

En este capítulo se describen y analizan los conceptos relativos a la ingeniería del software que son clave para el diseño e implementación de la aplicación del proyecto. La aplicación de estos conceptos es lo que permite dotar al proyecto de las características deseadas para el mismo, tal y como se describió en la introducción.

3.1 Abstracciones y Conceptos de Ingeniería del Software

Para la toma de decisiones sobre el diseño de software y arquitectura del proyecto es necesario un análisis previo del estado del arte respecto a las abstracciones y conceptos disponibles en estas áreas.

Antes de comenzar, es conveniente indicar que el área de conocimiento dentro de la Ingeniería del Software a la que se circunscriben los conceptos analizados a continuación es el de los Patrones de Diseño dentro del paradigma de la Programación Orientada a Objetos.

3.1.1 Extensibilidad dinámica de aplicaciones

La extensibilidad de aplicaciones es un tópico muy importante en el desarrollo de aplicaciones profesionales, empresariales y/o de gran envergadura, en la que la necesidad de dejar puntos de extensión abiertos para futuras funcionalidades del sistema no previsibles en el momento del desarrollo del mismo; es algo que los buenos diseñadores y arquitectos intentan incluir en sus aplicaciones.

Los fundamentos de la extensibilidad se encuentran en las abstracciones de diseño de software que se analizarán a continuación, comenzando por los principios más generales y avanzando a través de ellos hasta analizar alternativas específicas para la extensión de aplicaciones.

3.1.2 Inversion of Control (IoC)

La inversión de control es un principio de diseño de software que permite que el control del flujo de ejecución de una aplicación se invierta y, en vez de ser controlado por la aplicación principal, como ocurre en la programación procedural

clásica en la que el código va realizando tareas específicas y se sirve de librerías externas para resolver problemas genéricos, llevando el programa principal siempre el control, lo que se consigue es que la aplicación principal se convierta realmente en un marco de ejecución (framework) reutilizable (e idealmente extensible) que aporta organización y funcionalidades genéricas, cediendo el control del flujo de ejecución en puntos determinados a componentes de software más específico que serán quienes realicen las tareas concretas que consiguen el verdadero objetivo de la aplicación.

Este principio también es conocido como el Principio de Hollywood: *"no nos llames, nosotros te llamaremos"*[4], haciendo alusión a que las partes específicas del código ya no llaman a las partes genéricas (típicamente librerías), sino que precisamente las partes genéricas del código (el framework o la base de la aplicación) son las que ahora llaman a los componentes específicos para que realicen su tarea cuando sea el momento adecuado.

La Inversión de Control consigue dotar de modularidad y extensibilidad a las aplicaciones, algo fundamental en el mundo del Desarrollo de Software actual y necesario para la aplicación objeto de este proyecto. La modularidad se consigue por el hecho de que las partes del código que se dedican a tareas específicas no pertenecen al programa principal, sino que son "módulos" que se "conectan" al código principal. Esto hace que la modificación de un módulo no repercuta en otro módulo, que se encarga a su vez de otras tareas específicas y que es independiente del primero. Con la programación procedural clásica esto no ocurre, ya que el código específico se encuentra en el programa principal, lo que implica que un cambio en la parte específica obligará a reescribir parte de dicho programa principal. A su vez, IoC facilita la extensibilidad por el hecho de que las implementaciones concretas de las funcionalidades específicas se conectan al programa principal mediante "contratos" bien definidos, que pueden tomar diversas formas en su implementación concreta, como herencia de clases abstractas, implementación de interfaces, inyección de dependencias, etc., como se analizará más adelante, pero que siempre permiten que podamos añadir funcionalidades a los módulos e incluso cambiar unos módulos por otros completamente diferentes con la única exigencia de que cumplamos el contrato de conexión definido.

Como se puede observar, IoC lleva implícita la asunción de que el código genérico y el específico de una aplicación se desarrollan por separado, aunque operan en conjunto.

Uno de los principales defensores y divulgadores del principio de IoC es Martin Fowler, quien traza la etimología del concepto de Inversion of Control hasta su aparición por primera vez en el artículo *"Designing Reusable Classes"* [5].

3.1.3 Open/Closed Principle (OCP)

Este principio de diseño de software, se basa en el concepto de que se debe permitir la extensión de una aplicación o componente, pero no requerirse la modificación de su código para ello. Por tanto, un componente debe estar abierto para la extensión y cerrado para la modificación.

Existen dos formas de conseguir este comportamiento: composición y herencia. A título de ejemplo de funcionamiento, el objetivo se consigue en el segundo caso mediante la herencia de una clase que no permite la modificación de uno de sus métodos, ya que ese método está cerrado a modificación externa. Al crearse la clase hija que hereda de la misma sí se permite sobrecargar la implementación concreta del mismo, lo que posibilita el cambio de comportamiento para un caso concreto dentro de una aplicación, pero impidiendo que la implementación del método de la clase original pueda modificarse. De esta forma se garantiza que en todas las partes de la aplicación en las que se usa la clase original el resultado va a ser consistente en el tiempo porque la implementación del método no cambia y, a la vez, se permite que para casos concretos, que posiblemente no se preveían al crearse la clase, se pueda crear una variación de la clase que sobrecarga un método concreto para conseguir un comportamiento diferente del original.

El término Open/Closed Principle se le atribuye a Bertrand Meyer [6] y se encuadra en el contexto de los principios SOLID [7], que representan un conjunto de buenas prácticas en el diseño de aplicaciones basadas en el paradigma de la Programación Orientada a Objetos.

3.1.4 Dependency Injection (DI)

Es un Patrón de Diseño que proporciona una implementación concreta de IoC para resolver las dependencias que necesita un objeto. Típicamente un objeto (una clase) es responsable de fabricar todas las dependencias que necesita para funcionar, esto es, todas las instancias de sus propias variables que necesita para realizar su tarea en tiempo de ejecución.

Con DI este enfoque se invierte, siguiendo el principio de IoC, de manera que las dependencias se "inyectan" en el objeto en tiempo de ejecución, no estando definidas en tiempo de compilación. Mediante este sistema la lógica del código que usa las dependencias se separa de la creación de las dependencias. Esto se puede conseguir mediante diferentes técnicas, como inyección de dependencias a través del Constructor de la clase, de sus Setters o de Interfaces que implementa la clase.

Para ilustrar el concepto de una manera más visual, imaginemos la clase Coche, que tiene las siguientes variables (dependencias): Neumático, Batería y Motor. Estas tres dependencias son Clases que dan significado al concepto de Coche y que son necesarias para que la clase Coche funcione en tiempo de ejecución cuando se convierta en una instancia concreta (un objeto). Con un enfoque clásico, el código de inicialización de Coche será responsable de crear las tres dependencias. Sin embargo, con DI, Coche no será responsable de crear e inicializar estas dependencias, sino que éstas vendrán de fuera a través de la actuación de una lógica externa de "inyección" de dependencias, que inyectará en tiempo de ejecución las variables que recibe el constructor, por ejemplo. De esta manera tan simple se está consiguiendo que sólo en tiempo de ejecución se conozca y asigne la dependencia concreta que necesita Coche para funcionar. Esto significa que quien use Coche en su programa concreto, podrá considerar inyectarle NeumáticoInvierno o NeumáticoVerano según sea más conveniente, sin venir esta decisión ya tomada en el diseño de la propia clase Coche, lo que la haría mucho menos flexible.

El único requisito que debe cumplir la lógica que construye e inyecta la dependencia en tiempo de ejecución en el objeto que la necesita es observar y cumplir el "contrato" que indica cómo deben ser esas dependencias. Esto se consigue típicamente mediante una Interfaz (entendida dentro del paradigma de la orientación a objetos) que debe implementarse para crear un objeto con el comportamiento que necesita la clase que lo va a utilizar. Es decir, esta clase define una Interfaz que representa el tipo de objeto que necesita recibir como dependencia y la lógica de inyección de dependencias creará dependencias que implementen dicha Interfaz, asegurándose así de que cumple con el comportamiento exigido.

Evidentemente, esto conlleva dos mejoras fundamentales frente a un enfoque clásico:

- **Gran desacoplamiento del código:** de manera que se pueden crear diferentes instancias de Coche en tiempo de ejecución a las que se les inyecten diferentes

tipos de Neumático, incluyendo algunos que no existían cuando se creó originalmente el código fuente, lo cual permite una gran extensibilidad y flexibilidad.

- **Facilidad para Tests Unitarios:** el código fuente resultante de aplicar DI es mucho más sencillo de testear mediante Tests Unitarios, ya que las dependencias de los objetos que se quieren testear no tienen que ser construidas e inicializadas para que el objeto funcione, sino que basta con inyectar dependencias que simulen a las reales, pero sin toda su complejidad, ya que no se las está testeando a ellas y, por tanto, no se necesita que tengan la complejidad de las reales. Estas dependencias simuladas se pueden implementar mediante Mocks o Stubs.

El desacoplamiento de código inherente al enfoque DI facilita la creación de aplicaciones extensibles en las que se puedan enchufar componentes de software a modo de plug-in, ya que los desarrolladores del plug-in conocen bien el contrato (la Interfaz) que tienen que cumplir los mismos para satisfacer las dependencias de las clases que las utilizan. Esto también permite que los equipos de desarrollo de estas clases y de los plug-in puedan ser completamente independientes.

Finalmente, y como precisamente la lógica de construcción e inyección de dependencias está separada de la construcción e inicialización de los objetos que las van a utilizar, es típico automatizar esta lógica mediante ficheros de configuración. Gracias a esto no es necesario programar ninguna línea de código para poder enlazar en tiempo de ejecución las dependencias concretas con los objetos que las van a utilizar. A través de un fichero de configuración se pueden cambiar estas dependencias sin modificar nunca el código fuente del programa principal.

Como alternativa a DI existe el Patrón de Diseño Service Locator [8], que también consigue desacoplar las dependencias del objeto que las utiliza a través del concepto de "localizador de servicio" (Service Locator), siendo "servicio" la dependencia. Service Locator funciona mediante el concepto de que el objeto que tiene dependencias contiene una variable Service Locator a la que él mismo le indica qué dependencia necesita, de manera que Service Locator la localiza y la proporciona al objeto. Como se puede ver, y aunque este enfoque aísla las dependencias del objeto que las utiliza, al igual que DI, no cumple sin embargo con el principio de IoC, ya que es el objeto que tiene las dependencias el que controla todo el proceso y además las pide a Service Locator. Por el contrario, en DI, el control del flujo de ejecución sí se invierte y el objeto que contiene las dependencias no llama a ningún método

para resolverlas ni controla el flujo de ejecución del programa, antes bien: es el programa el que inyecta las dependencias en el objeto y lo llama cuando lo necesita.

Es por esta razón que se ha elegido la alternativa de DI frente a Service Locator para el proyecto, ya que ambas proporcionan la modularidad y el desacoplamiento necesario para conseguir la extensibilidad requerida por el proyecto, pero sólo DI sigue el principio de IoC, lo cual proporciona un código más elegante y elimina realmente todo conocimiento concreto sobre las dependencias del objeto que las utiliza.

3.1.5 Patrón Plug-in

El Patrón Plug-in termina de cerrar un círculo que comienza con el principio de Inversion Of Control, se concreta con el Patrón de Dependency Injection y se termina de implementar con el Patrón Plug-in. Este Patrón aporta el hecho de que el enlace entre las clases que representan dependencias y las clases que las utilizan se realice mediante configuración y no mediante cambios en la lógica de inyección de dependencias en tiempo de compilación.

Tal y como hemos visto, la inyección de dependencias dota de extensibilidad, modularidad y flexibilidad a un programa al desacoplar el código específico de un determinado objeto (el código que representa su comportamiento en un determinado contexto) del código de su construcción e inicialización. Esto es, además una concreción del principio de inversión de control.

Sin embargo, en algún lugar del código fuente es necesario incluir lógica que resuelve las dependencias en tiempo de ejecución y las inyecta en los objetos adecuados. Desde mi punto de vista, esta solución, aunque muy utilizada, no es especialmente elegante e higiénica como opción para enlazar dependencias con los objetos que las utilizan y, además, cuando existe un grafo de dependencias relativamente grande entre los objetos de una aplicación (típico en aplicaciones profesionales y/o de gran escala), esta lógica de enlace de dependencias puede convertirse en un problema y transferir la inconveniencia que se pretendía evitar mediante IoC y DI desde el código de los objetos concretos al código de inicialización del programa principal, donde existirá mucho código dedicado exclusivamente a la configuración. Además, si se mantiene la lógica de inicialización dentro del programa, en puridad, y aunque se hayan dado pasos en el camino de la extensibilidad, no se habrá conseguido la independencia total del código que evite su recompilación ante

cambios en las extensiones utilizadas, que se decidirán en tiempo de ejecución. El patrón Plug-in, sin embargo, sí completa esta total independencia.

El principio que rige el diseño del patrón Plug-in sigue siendo que se definen interfaces en un paquete de software y su implementación reside en otro paquete. De esta manera, el software que necesita la dependencia desconoce completamente la implementación de la misma. La clave de este enfoque es que permite enlazar dependencias en tiempo de configuración en vez de en tiempo de compilación, aportando, ahora sí, total dinamismo en tiempo de ejecución a una aplicación. Como se muestra en la figura 3.1, un objeto del dominio tiene una dependencia de una interfaz que podrá ser implementada por cualquier clase que pretenda satisfacer dicha dependencia.

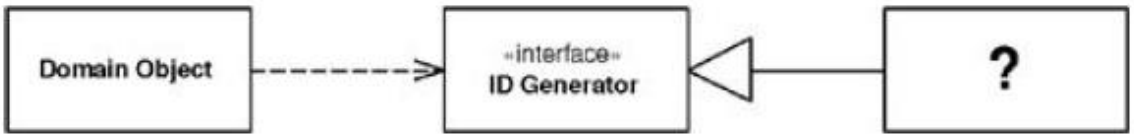

Figura 3.1. Diagrama de clases del patrón Plug-in

El patrón Plug-in termina de redondear, por tanto, un diseño extensible dinámicamente en el que las dependencias se convierten conceptualmente en Plug-ins, ya que se pueden enchufar "en caliente" en los puntos de extensión habilitados en el código, gracias al patrón DI que se ha aplicado en el diseño de las clases, indicándose en tiempo de ejecución qué dependencias concretas (Plug-ins) van a ser enchufadas en qué clases para resolver problemas concretos. Esta decisión se toma en tiempo de ejecución y se refleja típicamente en un fichero de configuración externo (normalmente XML) que se puede modificar en tiempo de ejecución con independencia del código del programa.

Por tanto, el patrón Plug-in proporciona la ventaja de la gestión centralizada de la configuración de todas las dependencias en tiempo de ejecución.

La solución global aportada por el Patrón Plug-in combinado con Dependency Injection como implementación concreta del principio de Inversion of Control es la que se ha elegido para conseguir la extensibilidad dinámica requerida por el proyecto con la mayor elegancia.

En concreto, en el proyecto se utiliza esta capacidad, la extensión, para permitir el acoplamiento de cualquier algoritmo de procesamiento de imágenes para indicar sospechas de problemas en la piel inspeccionada, sin necesidad de modificar el programa original que sirve de base y de aplicación de servidor que recoge los datos de los dispositivos móviles.

3.1.6 Programación Orientada a Atributos

La programación orientada a atributos se basa en la extensión de un programa mediante el marcaje con metadatos (atributos) de elementos del código fuente de manera que este se enriquece con la semántica que proporcionan dichos metadatos al compilador, preprocesador o incluso entorno de ejecución (esto depende del lenguaje).

El mérito conceptual de los atributos consiste en mejorar la precisión de lo que se pretende programar añadiendo semántica al código, a la misma vez que lo mantiene lo más simple e higiénico posible al esconder los detalles de implementación que esa semántica extra lleva asociada. De esta forma también se consigue que el código sea mucho más legible aunque se hayan enriquecido sus capacidades semánticas.

Aunque la utilización de atributos puede parecer similar a la utilización de macros que son luego expandidas posteriormente por un preprocesador, su funcionamiento es diferente: con las macros el código que se despliega por parte del preprocesador forma parte del código fuente final del programa, compilándose todo el conjunto. Sin embargo, los atributos se despliegan a un fichero separado del código fuente original, que siempre permanece limpio y legible. Es conocido por todos que las macros pueden correr el riesgo, si se utilizan inadecuadamente, de generar código ilegible e inmantenible. Los atributos suponen un enriquecimiento del código fuente original mucho más higiénico y mantenible.

Cabe reseñar que tanto Java como C# son dos lenguajes de programación modernos que carecen de preprocesador [9][10], tal vez precisamente por los problemas que pueden generar las macros.

El uso de los atributos en este proyecto aporta la capacidad de enlazado de dependencias en tiempo de ejecución, que es parte fundamental del patrón Plug-in. Se decoran clases e interfaces con atributos para indicar el tipo de sus dependencias,

que son resueltas, como se describirá en el capítulo dedicado a la implementación del código, en tiempo de ejecución.

3.2 Interoperatividad, API e Interfaces entre Sistemas

La interoperatividad es otro de los objetivos fundamentales que persigue este proyecto y para elegir los mejores métodos para hacerlo interoperable es necesario conocer las alternativas existentes, así como sus pros y sus contras. Eso es precisamente lo que se analiza en las siguientes subsecciones.

3.2.1 HTTP

Hypertext Transfer Protocol (HTTP) es un protocolo perteneciente a la capa de aplicación de la Suite de Protocolos de Internet que conforma la base de la comunicación de la World Wide Web (WWW). HTTP funciona mediante peticiones y respuestas que se realizan en un modelo cliente – servidor. El contenido de los mensajes HTTP se articula en forma de Hypertext, que es texto estructurado con enlaces embebidos a recursos de Internet. Estos enlaces se llaman Hyperlinks. Típicamente el Hypertext toma la forma de HTML (Hypertext Markup Language), que es el lenguaje con el que se crean las páginas webs de Internet y que los navegadores saben renderizar en pantalla. HTTP/1.1 es la especificación estándar del protocolo utilizada en la actualidad [11].

HTTP define una serie de métodos (también denominados *verbs*), de los cuales, los principales son:

- **HEAD:** se utiliza en peticiones que quieren conocer metadatos sobre un recurso determinado que se pide al servidor.
- **GET:** se utiliza para realizar peticiones de la representación de un recurso determinado del servidor. Es un método que sólo lee datos del servidor.
- **POST:** se utiliza para realizar peticiones al servidor que llevan datos en el cuerpo de la misma. Estos datos podrán ser utilizados por el servidor.
- **PUT:** se utiliza para realizar peticiones que envían datos en el cuerpo de la misma y que deben ser utilizados sobre un recurso determinado representado por su URI.

- **DELETE:** se utiliza en peticiones que eliminan un determinado recurso del servidor.

Estos métodos son los más relevantes para el proyecto, ya que conforman la base HTTP sobre la que se asientan los servicios web REST, que se analizarán más adelante.

3.2.2 JSON

JSON es un formato de intercambio de datos diseñado para ser utilizado en entornos donde el peso de las comunicaciones debe ser el más bajo posible y para poder ser entendido fácilmente por humanos, así como poder ser generado y parseado de manera muy simple por software.

JSON es un formato de texto independiente del lenguaje, aunque tiene características muy similares a las que se encuentran en la familia de lenguajes basados en C, como el propio C, C++, Java, C#, Python, Perl y Javascript. Esto hace que JSON sea especialmente útil como formato de intercambio de datos, ya que todos estos lenguajes son capaces de parsearlo de manera muy sencilla; en ocasiones incluso automáticamente con métodos incluidos en librerías estándar.

JSON está diseñado para almacenar dos tipos de datos:

- Colecciones de pares clave/valor, lo que es ideal para representar estructuras de datos estándar como objetos, structs, tablas hash, diccionarios, etc.
- Lista de valores ordenados, lo que es muy útil para representar arrays y listas.

Prácticamente todos los lenguajes de programación modernos utilizan estructuras de datos de este tipo, lo que hace que JSON sea un formato ideal para almacenar sus representaciones y transportarlas entre aplicaciones.

La figura 3.2 muestra un objeto JSON, que es un conjunto desordenado de pares nombre/valor. Cada objeto empieza con el símbolo “{“ y termina con el símbolo “}”, el nombre se separa del valor por el símbolo “:” y los pares nombre/valor se separan entre sí por el símbolo “,”.

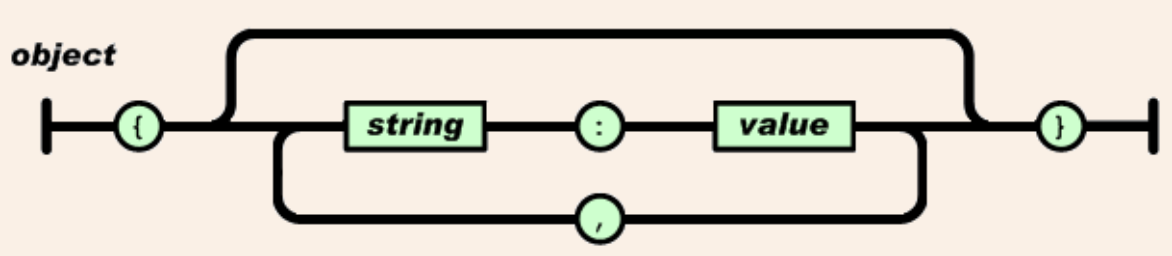

Figura 3.2. Objeto JSON.

La figura 3.3 muestra un array JSON, que es una colección ordenada de valores. El array comienza con el símbolo "[" y termina con el símbolo "]". Los valores se separan con el símbolo ",".

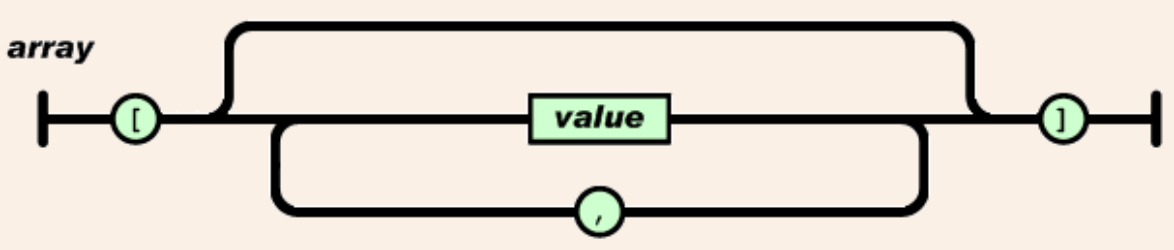

Figura 3.3. Array JSON.

La figura 3.4 muestra los valores que puede tomar un *value* JSON. Un *value* puede ser un string flanqueado por dobles comillas, un número, true o false, un array o null. Estas estructuras pueden ser anidadas.

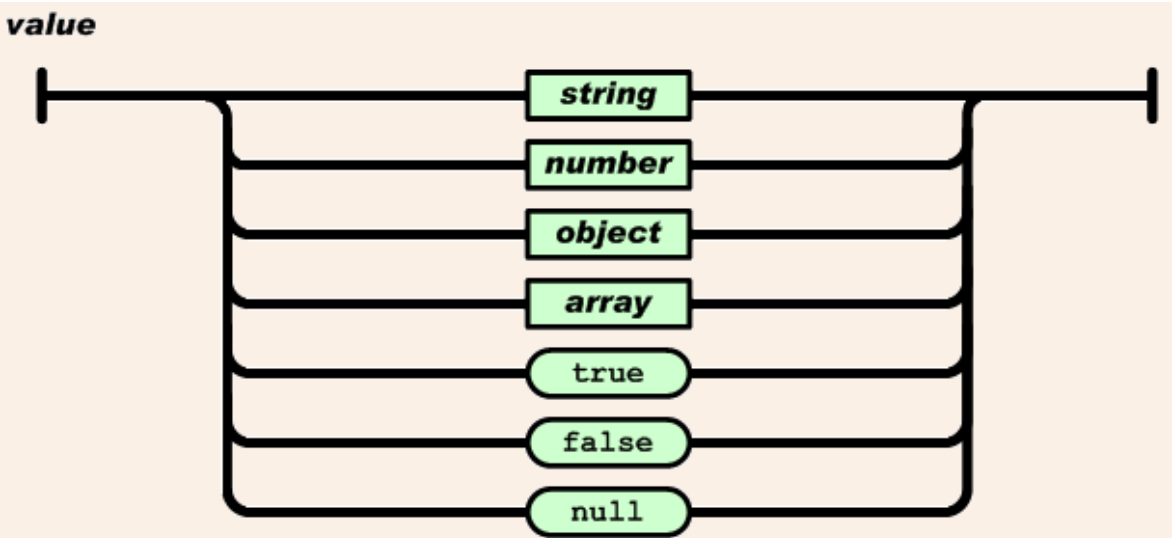

Figura 3.4. Valor JSON.

La figura 3.5 muestra los valores que puede tomar un número en formato JSON. Un número puede representar los mismos valores que representan los números en C, C# o Java, con excepción del formato octal y hexadecimal, que en JSON no se pueden representar.

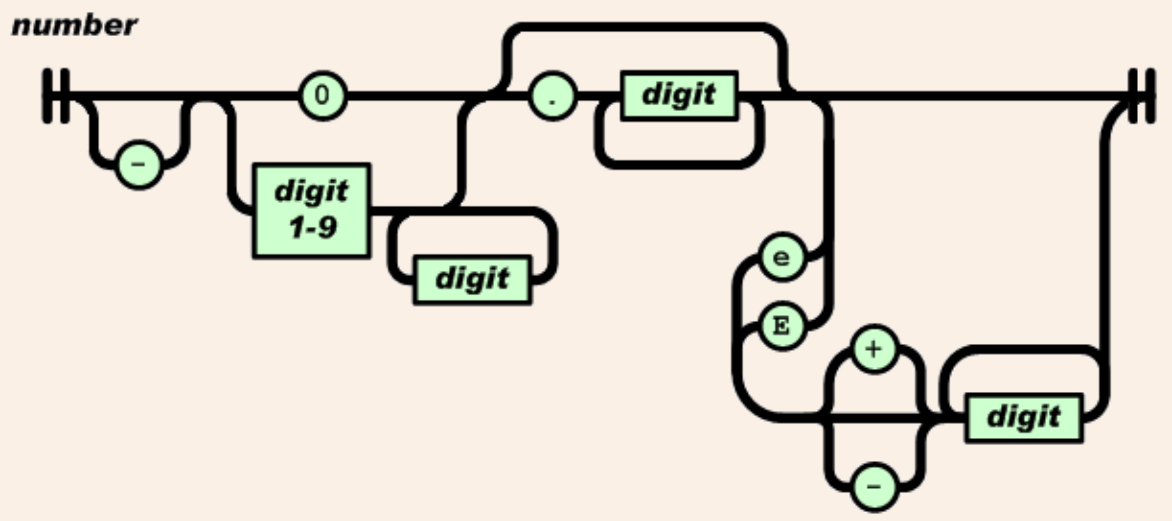

Figura 3.5. Número JSON.

La figura 3.6 muestra un string en formato JSON. Un string está compuesto por cero o más caracteres UNICODE flanqueados por dobles comillas. El carácter de escape es el símbolo "\". Un carácter individual se representa como una string de un solo carácter.

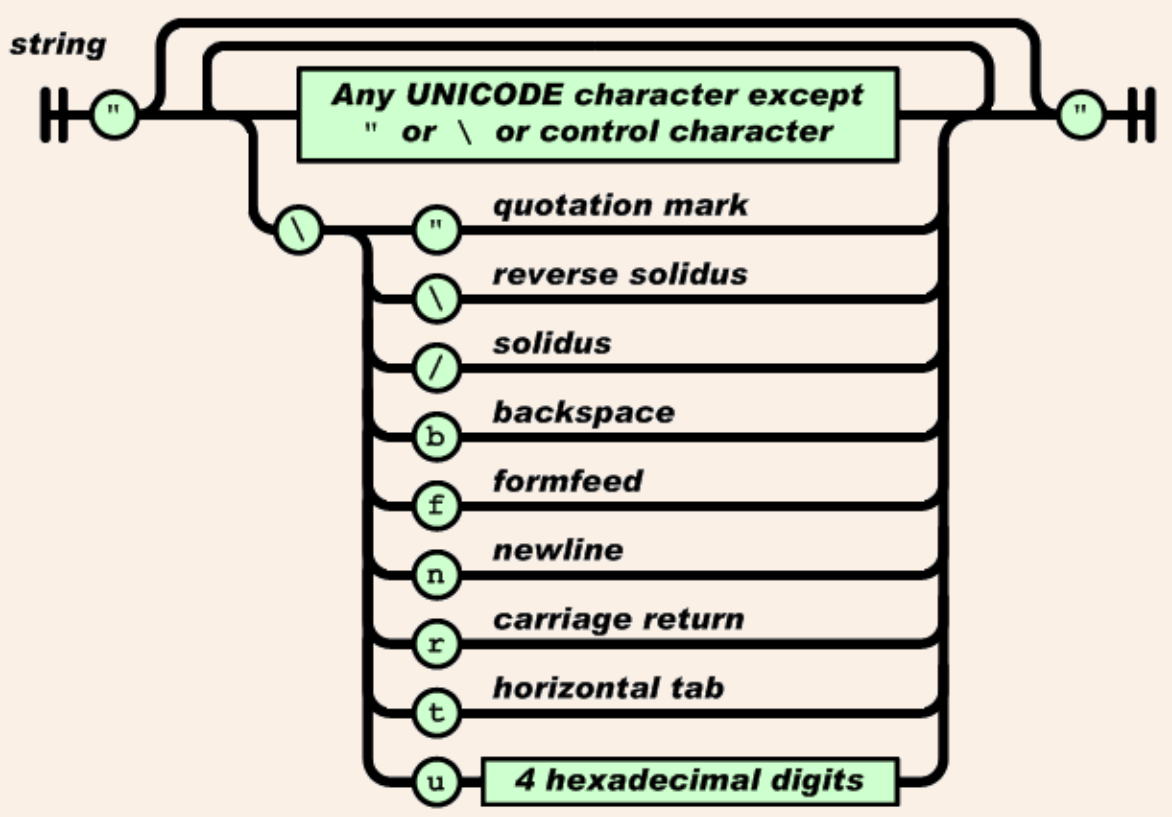

Figura 3.6. String JSON.

3.2.3 Web Services

Cada vez más, las aplicaciones web se comunican directamente unas con otras, además de comunicarse directamente con el usuario final. La interfaz programática que posibilita esa comunicación aplicación a aplicación son los servicios web. De esta manera, los servicios web constituyen un método de intercambio de información entre dos sistemas de software a través de una red de datos.

Los servicios web se apoyan en una serie de tecnologías, protocolos y formatos de datos, que enumeraremos a continuación, cuyo uso es universal y constituyen estándares de facto en la industria. Esto permite una interoperatividad total entre sistemas y hace que la implementación concreta que cada sistema hace de los servicios web sea transparente a efectos de dicha comunicación. Podemos concluir, por tanto, que los servicios web, conceptual y arquitecturalmente, constituyen uno de los mejores ejemplos de desarrollo de aplicaciones basadas en la reutilización de componentes, modularidad y flexibilidad aplicados a la Web.

Servicios Web SOAP y REST

Existen dos tipos de servicios web utilizados como estándares de facto en la actualidad: los servicios web SOAP, que toman su nombre del protocolo de intercambio de mensajes que implementan: Simple Object Access Protocol (también llamados a veces simplemente Web Services) y los servicios web REST, que es un acrónimo de REpresentational State Transfer. Este último término hace alusión a que este estilo arquitectural para servicios web se basa en la transferencia del estado de los recursos entre el servidor y el cliente. Recurso, en este contexto, se refiere a una entidad con un tipo, unos datos asociados, unos métodos asociados para manipular esos datos y unas relaciones con otras entidades. Como se puede ver, un recurso en el universo REST es similar a un objeto en Programación Orientada a Objetos, con la restricción de que los únicos métodos que expone el objeto REST son los que permiten las operaciones CRUD (Create, Read, Update y Delete) sobre sus datos. Los objetos en la Programación orientada a Objetos pueden tener cualquier número de métodos con los fines más diversos.

Los servicios web SOAP fueron los primeros en aparecer y constituyeron el único estándar de facto durante mucho tiempo. El avance de la web y, sobre todo, de las API entre sistemas de software a través de Internet, así como la necesidad de que estas API pudieran ser consumidas por dispositivos móviles ante el auge de los smartphones y las aplicaciones móviles, dieron lugar a la popularización de los servicios web REST. Analizaremos al final de este apartado las diferencias, ventajas y desventajas de cada uno de estos dos enfoques para construir servicios web, pero primero analizaremos las bases tecnológicas de cada uno de ellos.

Tanto los servicios web SOAP como REST tienen en común que se apoyan sobre una troncal de comunicación que provee de funciones de transporte a las peticiones y respuestas de los servicios web. Esta capa de transporte es, de facto, el protocolo HTTP, que es el estándar de comunicación en la Web.

A partir de aquí comienzan las diferencias: sobre esta capa de transporte se sitúan los mensajes que se intercambian los servicios web entre el cliente y el servidor en una interacción concreta. En SOAP Estos mensajes típicamente se codifican sobre el formato de datos basado en texto, eXtensible Markup Language, más conocido como XML. XML define una serie de reglas que permiten codificar de forma normalizada tanto estructuras de datos como mensajes que son fácilmente inteligibles tanto por humanos como por sistemas de software. Precisamente, uno de los objetivos de diseño de XML es la interoperatividad de sistemas de software a

través de Internet. Sobre la base de XML, los servicios web SOAP utilizan el protocolo Simple Object Access Protocol para intercambiar mensajes XML sobre el mencionado HTTP. SOAP provee de una capa de protocolo sobre HTTP y XML para normalizar la codificación de mensajes a nivel de cabeceras HTTP y de formato en el fichero XML para que tanto el cliente que consume el servicio web como el servidor que lo expone sepan cómo codificar y decodificar las sucesivas peticiones y respuestas intercambiadas.

El enfoque utilizado por los servicios web REST es más simple y no define un protocolo concreto para la codificación de mensajes. REST se apoya en cualquier formato de datos que sea capaz de viajar a través de HTTP, por ejemplo: JSON, XML, Atom. El protocolo de petición y respuesta se apoya simplemente sobre los siguientes cuatro verbs de HTTP: GET, PUT, POST y DELETE. Cada uno de estos verbs representa una acción que se quiere realizar utilizando los datos enviados en alguno de los formatos listados anteriormente. Como se puede ver, no existe protocolo que codifique y decodifique la intención de una petición concreta, sino que lo que existe realmente es una convención para leer las peticiones y las respuesta de una determinada manera, según el verb HTTP y el formato de datos utilizado. De hecho, REST se define como un estilo de arquitectura, no un protocolo. Este estilo arquitectural exige solamente que se cumplan una serie de restricciones y, cuando se cumplen, se define a la implementación concreta de servicios web REST que lo hacen como una implementación *RESTful*. Estas restricciones son las siguientes:

- **Client—Server:** la implementación de la lógica de la aplicación cliente y la aplicación servidor debe estar bien diferenciada. El cliente, por ejemplo, no debe conocer ni preocuparse de dónde se almacenan los datos, siendo esto tarea del servidor.
- **Stateless:** el servidor no almacena información de sesión (estado del cliente).
- **Cacheable:** las respuestas del servidor deben definirse como cacheables o no para que el cliente o los intermediarios puedan cachearlas o no, según sea ventajoso como medida de optimización o deba evitarse el cacheo en cada petición para no utilizarse información obsoleta.
- **Layered System:** un cliente no debe poder distinguir si está conectado directamente al servidor o a servidores intermedios, lo que hace que la arquitectura del sistema sea altamente escalable sin interferir en el resultado.

- **Code On Demand (Opcional):** el servidor debe poder extender las funcionalidades del cliente mediante el envío de código ejecutable, como por ejemplo Applets de Java.
- **Uniform Interface:** los recursos se representan mediante URIs:
  - Las representaciones de los estados de un recurso permiten al cliente tener la suficiente información como para modificar o eliminar el recurso.
  - Cada mensaje contiene información suficiente sobre cómo procesar el mensaje.
  - Las transiciones entre estados que efectúa un cliente se realizan sólo mediante hipermedia (invocación de hyperlinks en el hipertexto de respuesta que se recibe desde el servidor a cada petición). Po tanto, a priori, el cliente no conoce las opciones que ofrece el servidor, sino que va transitando por ellas dinámicamente a medida que va recibiendo respuestas del servidor con enlaces a otras acciones.

Hasta aquí hemos descrito las similitudes y diferencias, a muy alto nivel, de los servicios web SOAP y REST. Vamos a profundizar ahora en los detalles más relevantes de cada uno, así como en sus ventajas e inconvenientes.

Características de los servicios web SOAP

- Son independientes del lenguaje, plataforma y protocolo de transporte.
- Utilizan WSDL para describir los mensajes, bindings, operaciones y ubicaciones de un servicio web, representando una suerte de contrato que indica cómo utilizar dicho servicio web.
- Exponen lógica de aplicación que un cliente puede consumir, no sólo de datos que representan el estado de un recurso, como en el caso de REST. SOAP permite, por tanto, la construcción de APIs más ricas que permiten llamadas a procedimientos remotos (RPC).
- SOAP es un protocolo que introduce una capa de normalización en el paso de mensajes entre un servicio web y el cliente que lo consume.
- Cuentan con características de alto valor añadido que son útiles para la implementación de sistemas con un alto grado de complejidad que requieren

funcionalidades avanzadas en áreas como: seguridad, control de transacciones, envío confirmado de mensajes, gestión de errores, etc.

- Estandarización y numerosas especificaciones bajo el cuerpo de los WS-* para la implementación de características avanzadas como las mencionadas en el punto anterior.
- Son capaces de funcionar en entornos distribuidos empresariales. REST asume comunicaciones punto a punto.

Características de los servicios web REST

- Son independientes del lenguaje, plataforma, pero no del protocolo de transporte, ya que requieren HTTP.
- Carecen de información de estado entre peticiones. El estado de la interacción cliente – servidor debe ser mantenido por la aplicación cliente.
- Provee de una infraestructura de cacheo de mensajes, al estar basado puramente en HTTP y no tiene ninguna capa de transporte o paso de mensajes por encima de dicho protocolo.
- No existe información que defina la API generada con estilo REST, antes bien, las posibles acciones que ofrece la API serán descubiertas en tiempo de ejecución mediante los hipervínculos que devuelven las respuestas del servidor, que indican otras acciones disponibles.
- Exige que el cliente se preocupe y tenga que conocer información de contexto, así como del contenido que se intercambia entre cliente y servidor, al no existir metadatos que describan dicho contexto e información, como en el caso de SOAP.
- Siguen un principio arquitectural basado en un enfoque de mínimos, sin añadir mucho valor añadido, sino ofreciendo la simpleza de utilización como mayor valor, inspirándose en el enfoque de la propia World Wide Web.

Principales ventajas de SOAP

- Aporte de características de alto valor añadido para aplicaciones complejas empresariales.
- Funciona en entornos distribuidos.

- Los principales IDE y plataformas de desarrollo ocultan gran parte de la complejidad de este tipo de servicios web automatizando la creación de la mayor parte del código de "fontanería" necesario.

Principales ventajas de REST

- Curva de aprendizaje muy poco pronunciada.
- Diseño simple de implementar comparado a SOAP.
- Optimización de comunicaciones al intercambiarse significativamente menos datos que en SOAP, evitando el overhead producido por este dada toda la metainformación que introduce el protocolo y disminuyendo dramáticamente el tiempo de respuesta de una aplicación [12]. Se pueden utilizar formatos como JSON para minimizar aún más el tamaño de los mensajes, mientras que SOAP sólo puede utilizar XML. Esta es una de las principales razones de su adopción generalizada en el mundo de Internet.
- Rápido procesamiento: mensajes muy simples de parsear en el caso de utilizar formatos como JSON para el intercambio de información.
- Escalable mucho más fácilmente que SOAP dada la sencillez de su arquitectura, lo cual es otra de las razones por las que se está convirtiendo en el estándar de facto de APIs alojadas en la nube de sistemas con muy elevada demanda: Amazon, Facebook, Twitter, Google, etc.

En el caso de este proyecto, las ventajas de los servicios web REST, en concreto la optimización de comunicaciones y la facilidad de parseo de JSON como formato de referencia aconsejan que este sea el modelo utilizado, en detrimento del estilo más sobrecargado de SOAP, cuyos valores añadidos no son interesantes para este caso concreto.

3.2.4 Serialización

Uno de los principales puntos que el proyecto tiene que abordar es la transmisión de datos, principalmente imágenes (pero también datos con otro tipo) entre la aplicación cliente, el servidor y viceversa, a través de una conexión de red. Esto supone un problema bien conocido y con diferentes opciones para su resolución.

El problema radica en que al transmitirse datos entre un sistema y otro a través de la red (o al almacenarse datos en una base de datos, etc.) lo que realmente se necesita hacer es interpretar las estructuras de datos que realmente el programa está manejando en memoria y "aplanarlas", de manera que puedan ser transmitidas como una secuencia de bits a través de una red, ser guardadas en un fichero o en una base de datos, por ejemplo. Esto ocurre porque el formato que tienen ciertas estructuras complejas de datos, como por ejemplo un objeto en Programación Orientada a Objetos, no es posible transmitirlo a través de una red de datos por motivos obvios de diseño, ya que en las redes se transmiten secuencias planas de bits, no estructuras de datos. Esto también es válido para los supuestos antes mencionados: almacenar una estructura de datos compleja en base de datos o en un fichero. La serialización, por tanto, se ocupa de resolver el problema de mapear una estructura determinada de datos en una secuencia plana de bits que pueda ser transmitida o almacenada mediante protocolos estándar.

En el caso que atañe al proyecto, no es posible enviar una imagen, cuyo formato es binario puro, sin serializarla previamente a un formato compatible y que no genere problemas a lo largo del proceso de codificación, envío, transmisión, recepción y decodificación del mismo. El problema con la transmisión de datos binarios a través de la red es que, de manera estándar, y si no se utiliza el mismo codificador y decodificador en ambos extremos de la comunicación para asegurar que el formato es bien interpretado, puede ocurrir que algunos elementos de red o alguno de los extremos de la comunicación (el cliente o el servidor) introduzcan errores derivados de la interpretación que hacen ciertos lenguajes, como por ejemplo la interpretación que hacen C y C++ de las cadenas, en las que un NULL es interpretado como fin de la cadena [13]. Esto significaría que un cliente o servidor escrito en C++ dejaría de leer la secuencia de bits que está recibiendo cuando recibiera un NULL, aunque la secuencia de bits realmente no hubiera terminado aún. C y C++ consideran que un NULL (NUL en ASCII) se representa por un byte con todos sus bits a cero ('\0').

Por lo expuesto en el párrafo anterior, y dado que la interoperatividad es clave en el proyecto, ya que se debe permitir la comunicación con el servidor de cualquier cliente, desarrollado en cualquier tecnología, es clave que se elija un formato estándar y cuya adopción sea total entre todas las tecnologías. Para la serialización y envío de datos binarios a través de la red existen codificaciones muy interesantes como BSON [14] o Protocol Buffers [15], desarrollado por Google para la intercomunicación de todos sus sistemas a través de la red. Sin embargo, ninguno de

los dos garantiza ser un estándar de facto que todas las tecnologías implementen, lo cual los desaconseja para este proyecto.

La alternativa que emerge como estándar de facto en la transmisión de datos binarios a través de la red, garantizando total compatibilidad entre sistemas es la codificación en Base64 [16]. En la mayoría de los casos, además, las tecnologías de desarrollo cuentan con implementaciones estándar de la codificación y decodificación en Base64 en las librerías disponibles.

La razón para que Base64 se haya convertido en un estándar seguro de intercambio de información binaria serializada es que esta codificación se limita a generar solamente caracteres ASCII imprimibles (dejando fuera caracteres como BEL (Bell) o EOT (End Of Transmission), lo que, de facto, garantiza que todos los decodificadores que vayan a interpretar la información en Base64 serán capaces de recuperar la información sin corromperla. La única otra alternativa segura y completamente estándar a Base64 es la codificación Hexadecimal (que es realmente codificación en Base16), pero se ha decidido utilizar Base64 porque es capaz de codificar más información utilizando menos espacio. El estándar Base64 permite codificar tres bytes en cuatro caracteres, en lugar de los seis caracteres que necesita el formato hexadecimal. Como ejemplo, un fichero de 100 KB con caracteres en UTF-8 sería codificado en 200KB mediante Hexadecimal y en 133 KB mediante Base64. Como se puede ver, la eficiencia de Base64 es sensiblemente superior y este es un punto que impacta muy positivamente en el escenario del proyecto, siendo uno de los objetivos el optimizar el tamaño de las comunicaciones.

Como comentario final, es interesante apuntar que la misma premisa que lleva a elegir Base64 como alternativa (basada en que en esta codificación sólo se representan los caracteres imprimibles del código ASCII, lo que reduce los errores al pasar la información codificada por diferentes medios de comunicación y codificadores) llevaría a elegir codificaciones que maximicen el uso de los caracteres imprimibles del código ASCII, que son exactamente 94. Tales alternativas existen, como por ejemplo Base85 [17], Base91 [18] y Base94 [19], pero todos presentan problemas sobre Base64, ya que en todos los casos son codificaciones menos extendidas y soportadas, lo que reduce la interoperatividad. Esto los descarta para el proyecto.

3.3 Arquitectura y Diseño de Aplicaciones Multicapa

En la ingeniería de software se ha recorrido un gran camino desde los inicios de la programación, en los que la programación de sistemas batch se basaba en la manipulación de ficheros sin mayores abstracciones asociadas en el diseño de los programas, hasta las modernas aplicaciones multicapa, pasando por el advenimiento de varios hitos fundamentales en el desarrollo de software: la arquitectura cliente – servidor, la programación orientada a objetos y, sobre todo, la creación de la Web y el desarrollo para navegadores; todo esto ocurrió en los años noventa.

A lo largo de esta evolución una serie de requisitos fueron ganando en importancia a medida que las aplicaciones de software (sobre todo empresarial) comenzaban a ganar envergadura y complejidad:

- **Mantenibilidad:** la facilidad con la que un sistema de software puede ser modificado para corregir errores, mejorar su rendimiento u otros atributos, así como para adaptarlo a nuevos entornos.
- **Reusabilidad:** el grado en el que un componente o programa puede ser utilizado en varios programas distintos sin necesidad de ser modificado.
- **Escalabilidad:** la facilidad con la que un componente o programa puede adaptarse a cambios en la demanda por parte de los usuarios.
- **Flexibilidad:** la facilidad con la que un componente o programa puede ser modificado para ser utilizado en un sistema o entorno para los que no fue específicamente diseñado.
- **Robustez:** el grado de fiabilidad y consistencia de un sistema de software.

La forma en la que la comunidad de software fue dando respuesta a la hora de proponer una forma de diseñar la arquitectura de un programa de software fue a través de la creación conceptual de las aplicaciones con varias capas. Las capas no son otra cosa que agrupaciones de componentes que se sitúan en el mismo nivel de abstracción, delimitándose así diferentes niveles en la construcción de una aplicación. De la misma forma, la construcción de una aplicación por capas se basa en la definición de una jerarquía entre las mismas, como se muestra en la figura 3.7.

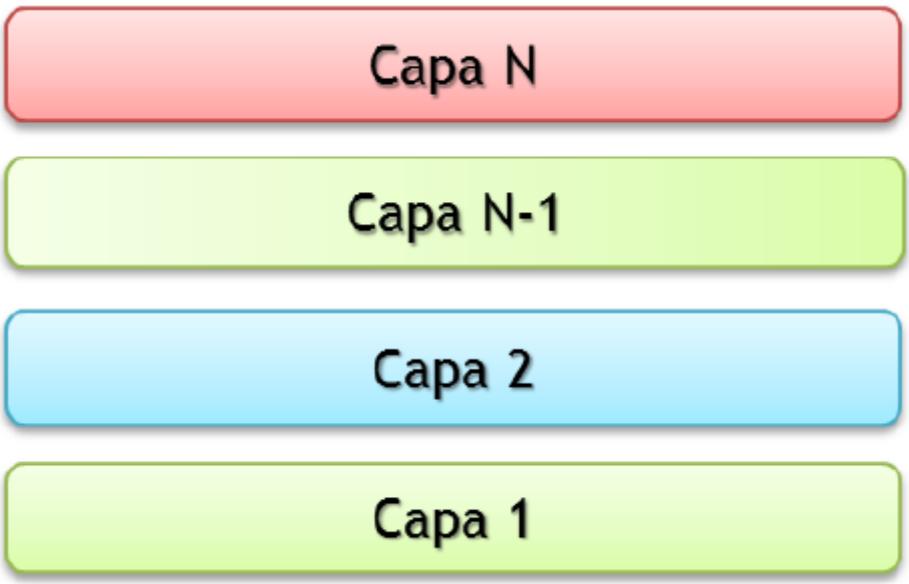

Figura 3.7. Jerarquía de capas.

Típicamente una capa conoce y utiliza los servicios ofrecidos por la capa inmediatamente inferior a ella y desconoce la existencia de la capa situada inmediatamente encima de ella, aunque expone funcionalidad para que esta pueda consumirla. Los componentes que se encuentran en la misma capa se pueden encontrar ligados fuertemente mediante dependencias entre ellos, etc., mientras que los componentes que se encuentran en capas distintas deben estar muy poco ligados, de manera que no existan dependencias fuertes entre ellos, lo que proporciona precisamente la independencia entre las capas. De hecho la gestión de dependencias entre capas es una de las claves de una arquitectura basada en capas.

El diseño de arquitecturas de software basadas en capas es un tópico extremadamente extenso, existiendo abundantes recursos especializados, como algunos de los que se relacionan en la bibliografía de esta memoria, pero conviene resaltar como mínimo algunas de las cualidades que derivan en los beneficios de la utilización de capas al dar solución a los requisitos de aplicaciones complejas enumerados en párrafos anteriores:

- **Separation of Concerns (SoC):** la división de una aplicación en capas proporciona una manera elegante para separar responsabilidades, de manera que cada capa es responsable de unas determinadas funcionalidades, que no se solapan con las responsabilidades de otras capas. El principio de SoC es la base del concepto de modularidad, ya que esta última se consigue aislando

componentes de software de otros componentes y haciéndolos cooperar mediante interfaces bien definidas y conocidos. Precisamente esto es lo que se consigue dividiendo una aplicación en capas con responsabilidades bien delimitadas.

- Derivada de la cualidad anterior, al localizarse las responsabilidades de un tipo determinado en una única capa, los cambios que se deben efectuar en el código cuando cambia una determinada responsabilidad se limitan a una capa y no afectan al resto; esto hace más fácil la corrección de errores, facilita el mantenimiento e incrementa la flexibilidad de una aplicación.
- Diferentes equipos de desarrollo deben poder programar a la misma vez en diferentes capas, e incluso componentes, con mínima o ninguna dependencia entre ellos. Esto es posible a que pueden hacer uso de las dependencias que tienen de otras capas/componentes mediante el uso de interfaces bien definidas, pero ignorando siempre las implementaciones internas concretas.
- Diferentes componentes pertenecientes a diferentes capas de una aplicación deben poder ser desplegados, mantenidos y/o actualizados en diferentes momentos en el tiempo, sin afectar al resto de capas o al resto de componentes de la misma capa.

Una aplicación desarrollada en capas puede incluir en su diseño diferente número de capas, pero las tres capas principales que presentan, como mínimo, la mayoría de aplicaciones multicapa son:

- **Presentación:** presentan una interfaz al usuario o exponen servicios como una forma de interfaz. Algunos ejemplos de funciones o componentes típicos de una capa de presentación son: una aplicación web que se ejecuta en un navegador, la gestión de los clics de ratón o teclado, la gestión de peticiones HTTP en un servidor y la exposición de una API de servicios web.
- **Lógica de Negocio (o Dominio):** se encarga de proveer de la lógica, en forma de algoritmia, objetos y datos (en un paradigma orientado a objetos) que resuelve los problemas de un dominio determinado. Es aquí donde aparecen reglas, validaciones, cálculos, etc., que tienen sentido en el dominio de aplicación concreto para el que se está intentado construir una solución.

Esta capa es, por tanto, la capa principal a la hora de la resolución de un problema determinado del dominio.

- **Datos:** se encarga de almacenar los datos, ya sea en una base de datos, un sistema de ficheros o cualquier otro soporte de persistencia.

Aunque estas tres capas son las más comunes y prácticamente imprescindibles en cualquier diseño correctamente realizado, es muy frecuente encontrar una capa intermedia entre la capa de lógica de negocio y la de datos; se trata de la DAL (Data Access Layer) que se encarga de gestionar las recuperaciones y almacenamientos de datos solicitados por la capa de lógica de negocio sobre la capa de datos. La inclusión de esta capa propicia que los componentes de la capa de lógica de negocio ignoren totalmente en qué estructuras y tecnologías se almacenan los datos, ya que la DAL los accede por ellos. Esto incrementa muchísimo la modularidad de una aplicación, propiciando que la capa de datos pueda cambiar sin tener que cambiar la capa de lógica que da solución al problema ni ninguna que esté por encima de esta en la jerarquía.

En la figura 3.8 se muestra un diseño multicapa más realista para una aplicación profesional compleja. Dicho diseño especifica una arquitectura precisamente compuesto por cuatro capas jerárquicas (datos, acceso a datos, lógica de negocio e interfaz de servicios y, finalmente, interfaz de usuario), así como por otras tres capaz transversales (comunicación, gestión operativa y seguridad). La existencia de estas capas transversales es común, ya que existen responsabilidades que deben intersectar a todas las capas de una aplicación, como la gestión de excepciones (que se situaría en la capa de gestión operativa) o la autenticación y autorización (seguridad).

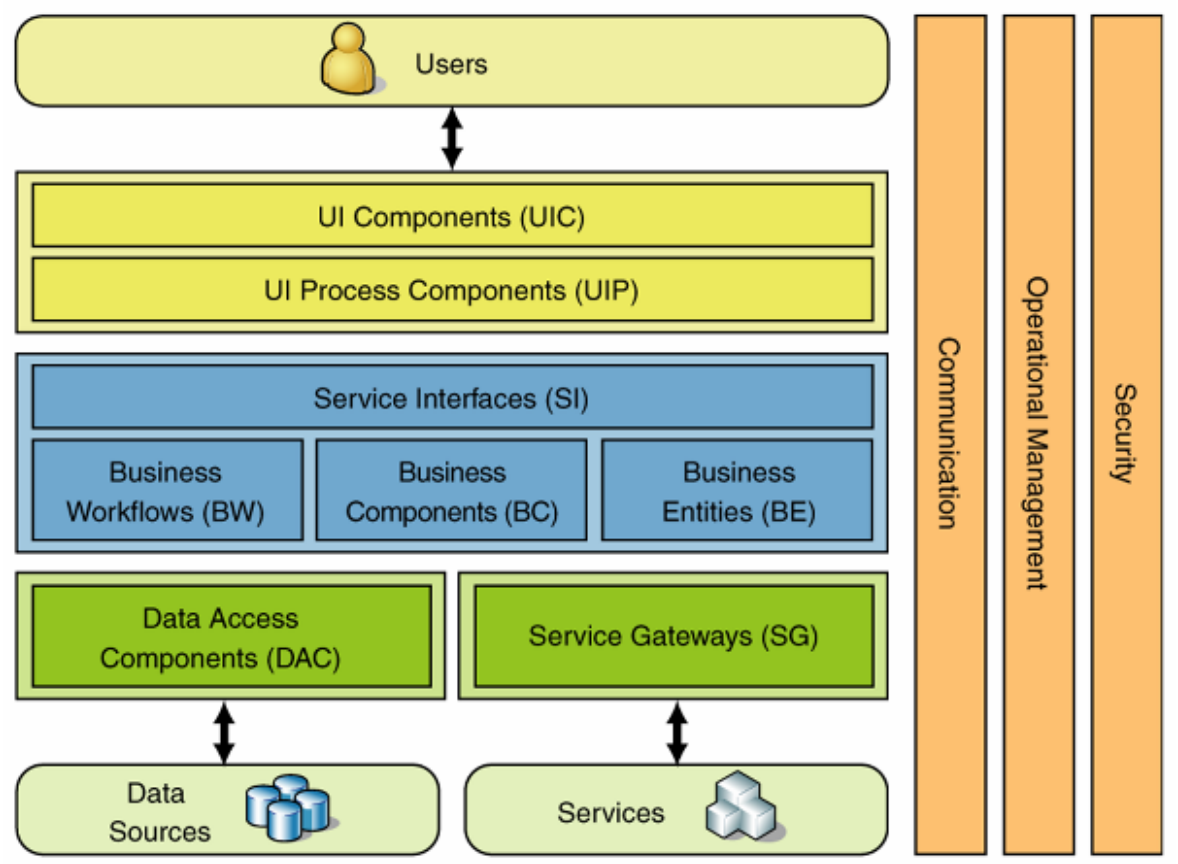

Figura 3.8. Diseño profesional de una aplicación multicapa compleja.

3.4 Patrón Model View Controller

El patrón de diseño MVC (Model View Controller) es muy popular en el desarrollo de aplicaciones actualmente y desde hace unos años, dados los beneficios que presenta al separar tres ámbitos que son diferentes entre sí, pero que colaboran para hacer que funcione una típica aplicación con interacción por parte del usuario (o cliente, en nuestro caso, pudiendo ser este un software y no una persona).

El objetivo del patrón MVC es separar la representación de la información y la lógica de negocio que opera sobre ella, la representación de esta información de cara al usuario y la coordinación de las acciones que operan tanto sobre la información como sobre su representación. Es decir, la principal ventaja del patrón MVC es que aísla los datos de una aplicación de su representación en una interfaz de usuario y de la interacción con esa interfaz que realiza un usuario. Esto proporciona mucha flexibilidad y facilidad de mantenimiento a cualquier aplicación con interfaz.

La figura 3.9 muestra un diagrama que representa el flujo típico que sigue un patrón MVC. El usuario interactúa con una interfaz (una aplicación web o móvil, por ejemplo) y sus acciones son enviadas a la controladora, que es quien se encarga de ejecutar los cambios en los datos de la aplicación (el modelo), mediante la lógica de negocio que sea necesario. Tras haberse producido la acción pedida desde la interfaz y ejecutada por la controladora, las interfaces suscritas al modelo (vistas) se actualizan y representan los datos sobre los que se ha operado.

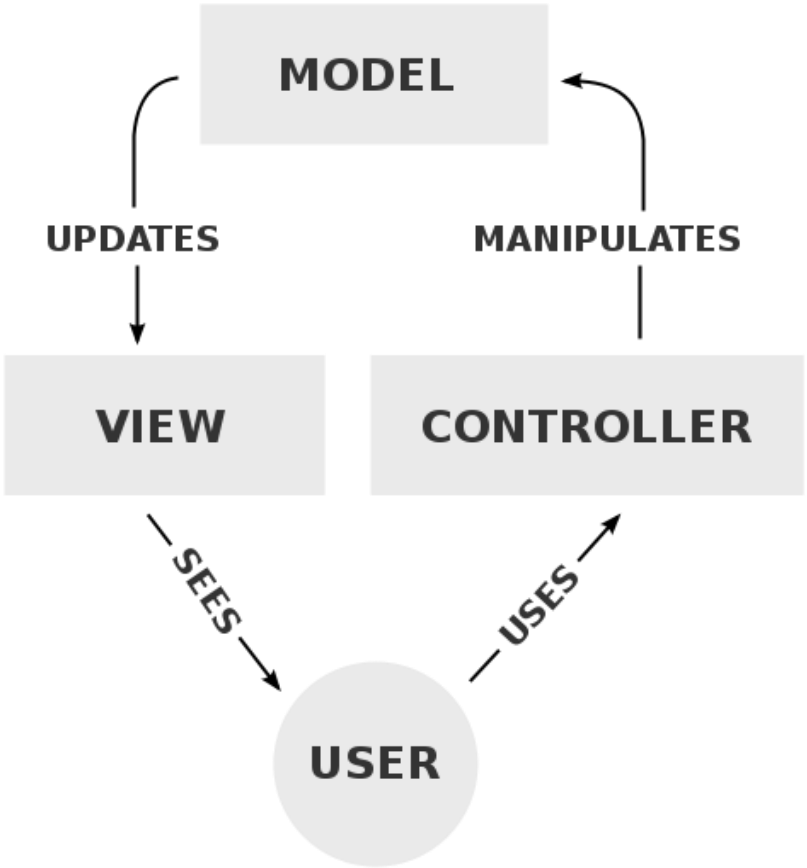

Figura 3.9. Esquema del patrón de diseño MVC.

El impacto de utilizar un patrón MVC se refleja en el código de un programa en el hecho de que no existirá código en la interfaz de usuario que manipule directamente los datos ni ejecute algoritmos sobre ellos. Esta función la desempeña la controladora, que coordina las acciones que afectan tanto a los datos y su lógica (modelo) como a su representación (vista). Esto propicia que un mismo modelo pueda ser representado por diferentes vistas, sin tener que multiplicar el código de manipulación de esta representación, que se escribe una única vez en el modelo. Por

ejemplo, el patrón MVC hace posible de manera sencilla el desarrollo de una aplicación donde los datos pueden representarse mediante una página web, exponerse a través de una capa de servicios web y, a la vez, salir a través de una aplicación móvil. Incluso posibilita no tener que reescribir ningún código en el modelo si se quiere añadir una nueva interfaz, por ejemplo, añadir una aplicación de escritorio como interfaz de una aplicación que ya cuenta con interfaces web y móvil. Esto es posible porque se pueden crear tres vistas diferentes que representen los datos del mismo modelo.

Si no se utilizara el patrón MVC existiría lógica que controlaría tanto la interacción del usuario como las modificaciones sobre los datos en cada una de las tres interfaces mencionadas.

4 Problema

4.1 Objetivo del Proyecto

El proyecto abordado tiene como objetivo proveer de una aplicación de Back End que se ejecuta en un servidor para la recepción de imágenes desde dispositivos móviles (aplicaciones cliente), su procesamiento mediante plug-ins de análisis de imagen que detectarán riesgos asociados a posibles problemas en la piel y, finalmente, el envío de una comunicación de vuelta a los dispositivos móviles indicando el resultado del diagnóstico.

El procesamiento en sí de las imágenes es un punto de extensión del proyecto, con lo que en el mismo se implementará una prueba de concepto de procesamiento de imágenes, pero no los algoritmos concretos de detección de anomalías sobre imágenes de la piel, que podrán ser enchufados en la aplicación servidor del proyecto más adelante en forma de extensiones o plug-ins.

Por tanto, la aplicación realizada en el proyecto debe presentar, una vez concluido su diseño e implementación, las siguientes características:

- **Interoperatividad:** la aplicación de servidor debe ser capaz de comunicarse e interactuar con cualquier aplicación cliente (móvil, web, escritorio, etc.) realizada en cualquier tecnología.
- **Arquitectura extensible dinámicamente:** la arquitectura de la aplicación debe permitir que diferentes algoritmos de procesamiento de imagen puedan ser acoplados a la misma en tiempo de ejecución y sin necesidad de recompilación del código fuente.
- **Optimización de las comunicaciones:** el intercambio de información entre la aplicación móvil (Front End) y la aplicación de servidor (Back End) debe minimizarse debido a los cargos económicos que produce el tráfico de datos a través de operadores móviles, así como para mejorar la interacción e inmediatez del feedback con el usuario.

4.2 Estado del Arte

En esta sección se explora el estado del arte de las aplicaciones médicas basadas en el paradigma cliente/servidor, haciéndose un análisis de las alternativas de diseño de software más interesantes.

4.2.1 MobiHealth [20]

MobiHealth es un proyecto de monitorización médica de pacientes en movilidad financiado por la Comisión Europea y realizado por 14 organizaciones de 5 países, incluyendo hospitales, proveedores de servicios médicos, proveedores de aplicaciones móviles, proveedores de hardware e infraestructura móvil, así como operadores de telefonía móvil.

La base de MobiHealth es que los pacientes visten una prenda que incluye sensores para monitorizar diferentes aspectos de su salud, lo que se conoce como BAN (Body Area Network), y estos a su vez se conectan a un dispositivo móvil que hace de Gateway con un servidor remoto donde se realiza la monitorización por parte de especialistas. Las aplicaciones de valor añadido y monitorización pueden también conectarse en local sobre el dispositivo móvil que recoge las lecturas de todos los sensores del BAN. Por ejemplo para que una enfermera que visita al paciente pueda ejecutar una aplicación específica y realizar determinados diagnósticos. En la figura 4.1 se muestra la arquitectura funcional mencionada, en la que la red de sensores (BAN) se conecta al dispositivo móvil (MBU) que hace de Gateway y de plataforma para ejecutar aplicaciones en local. A su vez, otras aplicaciones pueden ejecutarse en remoto en la infraestructura del proveedor de servicios médicos y alimentarse de los datos obtenidos de los sensores de la BAN.

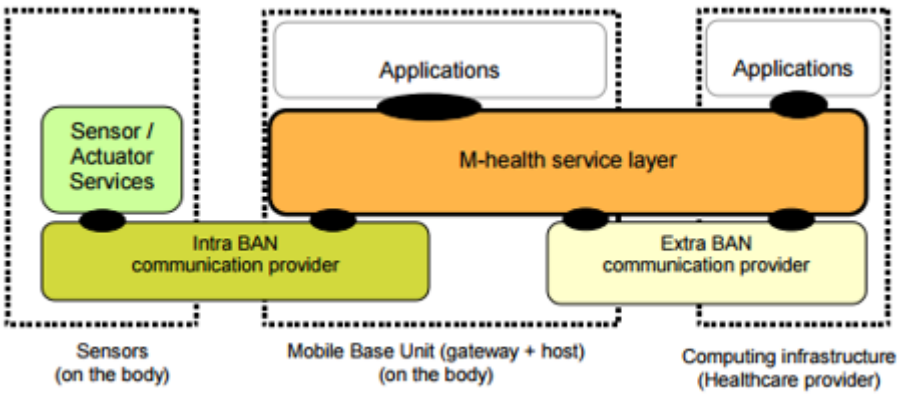

Figura 4.1. Arquitectura funcional de la plataforma MobiHealth

La arquitectura de MobiHealth se basa en que un "sustituto" que representa a cada sensor del BAN se ejecuta como componente de software en una localización remota. Es de ese "sustituto" de dónde se permite hacer lecturas y escrituras a las aplicaciones de software que se activen en el sistema. Por ejemplo, en un call center se levanta un "sustituto" de un sensor del BAN de un paciente y en un hospital situado en otro edificio se accede a ese sustituto y varias aplicaciones operan con sus datos.

Vemos, por tanto, que existe un escenario de comunicación remota y de envío de información médica a diferentes máquinas y aplicaciones, lo que se ha resuelto arquitecturalmente de la siguiente manera:

- **Remote Method Invocation (RMI):** la ejecución de aplicaciones que operan remotamente sobre el "sustituto", que no deja de ser un componente de software que representa a un sensor, se realiza mediante la invocación de métodos remotos de Java. Esta tecnología permite el intercambio de información y la ejecución de código remota través de una red.
- **HTTP POST:** la comunicación entre un sensor de la BAN y su "sustituto" en la ubicación remota se realiza a través de una red móvil 2.5G/3G mediante una petición HTTP POST en la que se incluyen los datos del sensor como payload de la misma.
- **Java 2 Micro Edition (J2ME):** la programación del dispositivo móvil que recolecta los datos de los sensores y envía las peticiones HTTP POST al servidor se realizó utilizando esta versión de Java para dispositivos móviles.

En la figura 4.2 se representa detalladamente lo descrito anteriormente, distinguiéndose en qué elementos de la arquitectura se utiliza como comunicación entre componentes Java RMI y en cuáles peticiones HTTP POST (representadas como BANip).

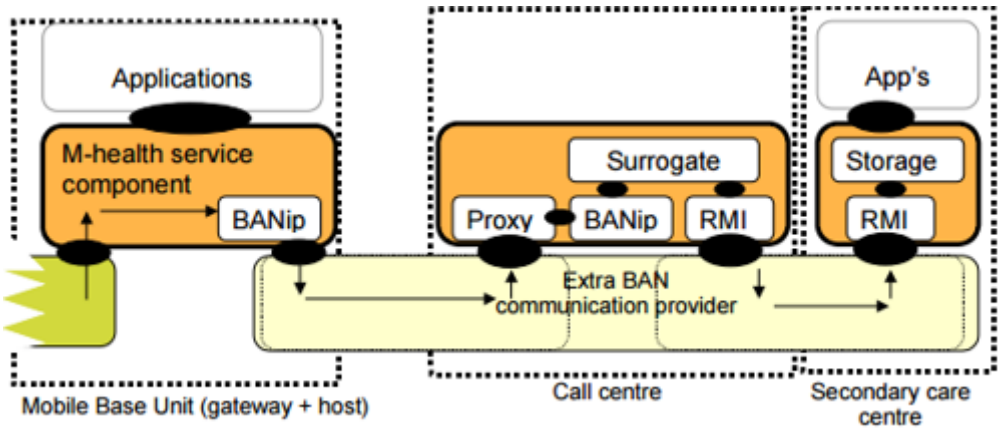

Figura 4.2. Detalle de la arquitectura de MobiHealth

4.2.2 Wireless Remote Healthcare Monitoring with Motes [21]

El objetivo de este trabajo es crear una arquitectura funcional para la monitorización remota de pacientes utilizando hardware y software commodity ya existente en el mercado en la medida de lo posible. Se utiliza una red de sensores, un servidor remoto donde se reciben los datos de los sensores y un dispositivo móvil que actúa de Gateway entre los sensores y el servidor remoto. A nivel funcional esta arquitectura es similar a la de MobiHealth, descrita en la subsección 4.2.1, sin embargo, técnicamente es muy diferente.

En la figura 4.3 se muestra el principio general del problema a resolver.

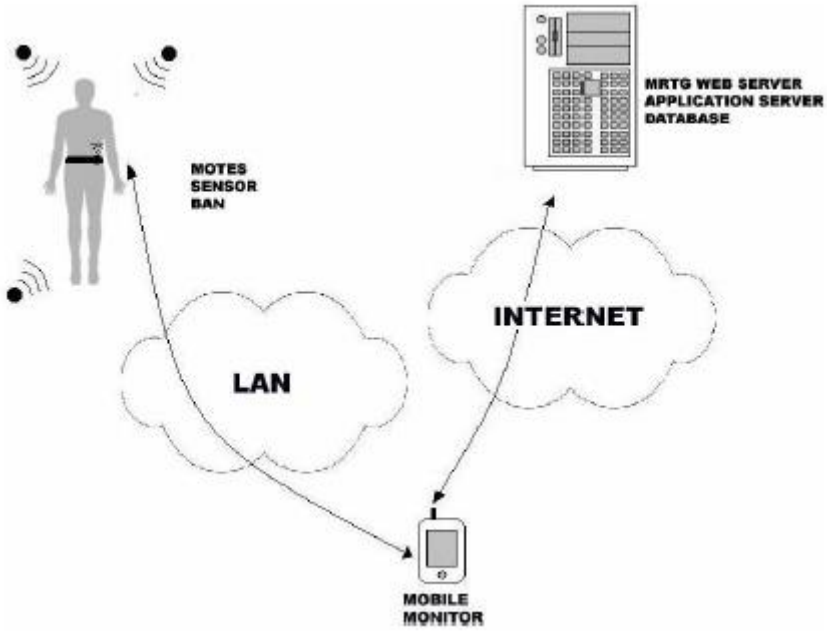

Figura 4.3. Escenario del problema y arquitectura básica

En este proyecto se realizó la aplicación cliente que se ejecuta en el dispositivo móvil en lenguaje C# sobre .NET Compact Framework. Los autores justifican la elección de esta tecnología y plataforma sobre J2ME por la simplicidad y mayor compatibilidad de la misma frente a J2ME.

La comunicación entre el dispositivo móvil y el servidor remoto se realizó mediante sockets TCP/IP, descartándose la otra alternativa que se analizó (servicios web con XML como formato de intercambio) debido al overhead que generaba en las comunicaciones frente a los sockets. La información se envía en ficheros de texto plano.

Una vez en el servidor, un script desarrollado en Perl se encarga de cargar los ficheros en el servidor Multi Router Traffic Grapher (MRTG), una aplicación de monitorización de tráfico de red que los autores utilizan en este caso para representar las señales de los sensores, como se muestra en la figura 4.4.

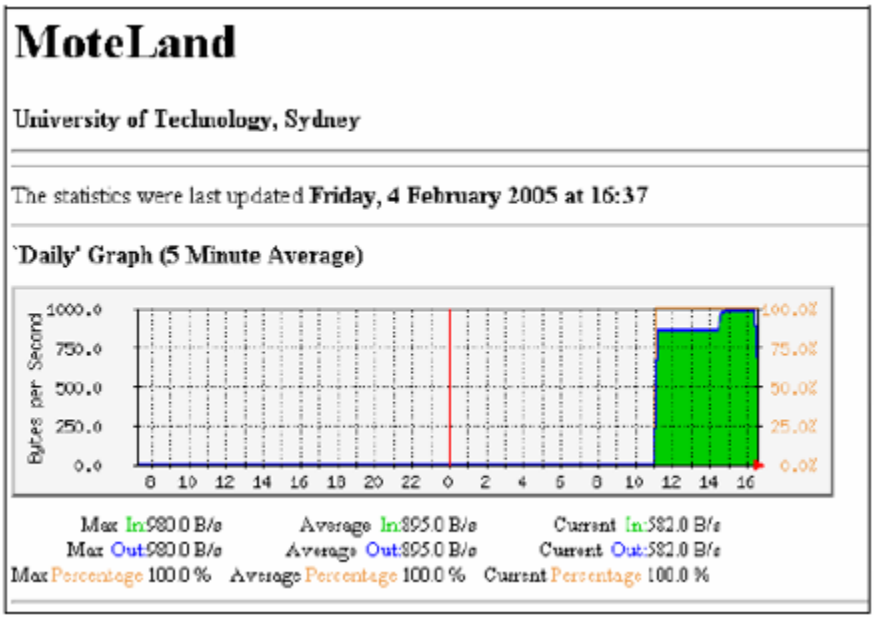

Figura 4.4. Display principal de Multi Router Traffic Grapher

4.2.3 Distributed Image Processing Application Considering CORBA and XML Technology [22]

El objeto de este proyecto es la creación de un sistema distribuido de análisis avanzado de imagen utilizando exclusivamente estándares abiertos y herramientas gratuitas, que pueda utilizarse para aplicaciones médicas, industriales, etc. El cliente es una aplicación de escritorio que es capaz de enviar y recoger imágenes que se han de procesar en remoto para utilizar la mayor potencia de procesamiento de un servidor.

La arquitectura de este sistema cliente/servidor se caracteriza por lo siguiente:

- **Common Object Request Broker Architecture (CORBA):** se trata de una tecnología que posibilita acceder a objetos en cualquier lugar de una red sin importar en qué lenguaje hayan sido implementados o sobre qué sistema operativo estén corriendo. El autor ha utilizado CORBA para garantizar la posibilidad de integrar de nuevos componentes distribuidos en el futuro sin importar su lenguaje de programación o sistema operativo. CORBA actúa como tecnología para establecer la comunicación entre el cliente y el servidor a través de una red de datos.

- **C++ y Magic++:** es el lenguaje de programación utilizado para la aplicación que corre en el servidor y que realiza el análisis de las imágenes. El autor ha elegido C++ porque ha considerado que el procesamiento de imagen es una tarea intensiva en cálculo y ha buscado el mejor rendimiento para el mismo, siendo C++ ideal en este sentido sobre otros lenguajes. Magik++ es la librería Open Source de procesamiento de imagen utilizada.
- **Python:** es el lenguaje de programación utilizado para la aplicación cliente y ha sido elegido por su facilidad de uso, lo que proporciona velocidad en el desarrollo. La aplicación cliente tiene que ser sencilla, con lo que Python cubre los requisitos desde el punto de vista del autor.
- **XML:** es el formato utilizado para el intercambio de datos entre el cliente y el servidor. Las imágenes se almacenan en XML y, dado que una imagen son datos binarios, el autor utiliza Base64 para serializarlas y almacenarlas en los ficheros XML.
- **File Transfer Protocol (FTP):** se utiliza para transferir los ficheros entre el cliente y el servidor.

En la figura 4.5 se muestra un esquema de esta arquitectura.

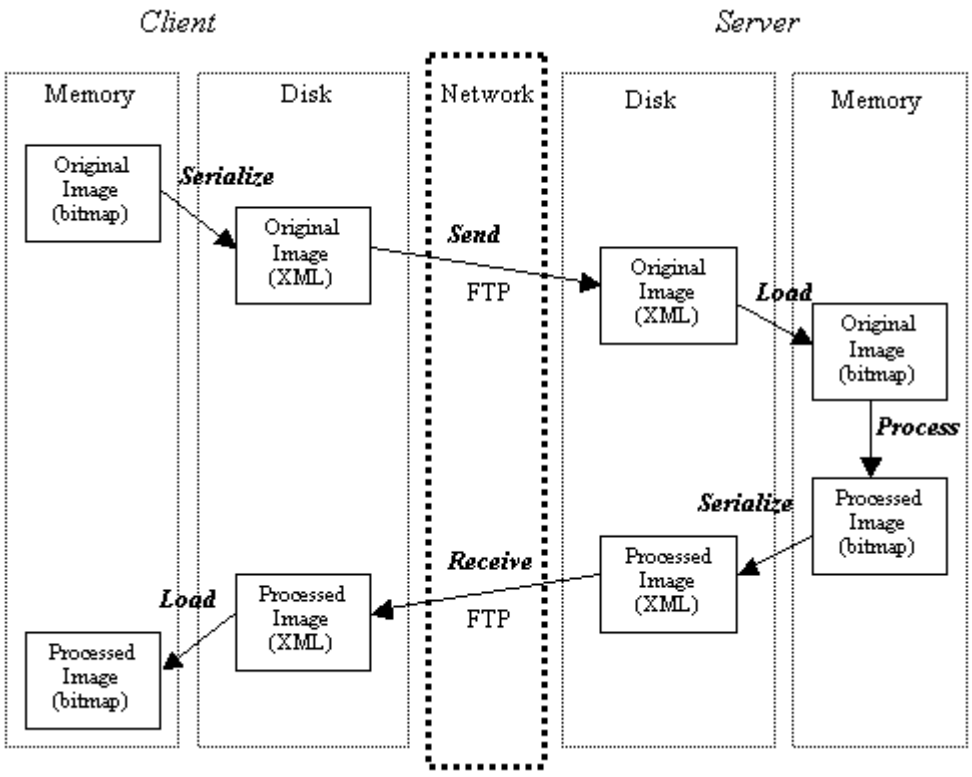

Figura 4.5. Esquema de la arquitectura de procesamiento distribuido.

4.2.4 Soluciones Propietarias

En el mercado existen múltiples aplicaciones propietarias que se han enfrentado al problema del diseño de una arquitectura de software que, siguiendo un esquema cliente/servidor, den solución a los requerimientos de las aplicaciones de salud en movilidad; estas aplicaciones van conformado el novedoso y cada vez más interesante campo de la mobile health (mHealth).

A continuación se reseñan tres de estas aplicaciones a modo de muestra representativa de las mismas. No se especifican en grandes detalles sus arquitecturas ya que al ser productos comerciales propietarios no están disponibles los pormenores de las mismas.

- **iHealth** [23]**:** esta empresa comercializa una suite de aplicaciones médicas para medir, entre otras cosas, la presión sanguínea, los niveles de glucosa o capacidades asociadas al estado de forma del usuario. Los resultados se presentan en el propio dispositivo móvil del usuario, a través de aplicaciones para Android e iOS, así como existe la posibilidad de almacenar los datos en un servidor remoto en Cloud y compartir las medidas tomadas con un médico o familiares.

  La figura 4.6 muestra un glucómetro que se interfasea con la aplicación móvil de iHealth mediante Bluetooth y la figura 4.7 muestra el esquema de conexión de varios dispositivos a través de un Gateway y hacia el Cloud de iHealth.

Figura 4.6. Aplicación móvil y sensor de iHealth

Figura 4.7. Esquema de iHealth Cloud

- **Mobile Assay (lab-on-device mobile platform)** [24]**:** Mobile Assay comercializa una solución para el análisis y diagnóstico móvil aplicable a diversos ámbitos, como la salud, agricultura, salud animal, etc. La solución se basa en un cliente (app móvil: Mobile Rapid Diagnostic Test Reader) y una aplicación de servidor (Mobile Cloud).

  El funcionamiento se basa en que con el dispositivo móvil se toman imágenes de la tira reactiva con la que se hace cualquier test de flujo lateral basado en colores y su graduación y la propia app móvil procesa la imagen y devuelve resultados cuantitativos al usuario. Adicionalmente se puede enviar una imagen al servidor en la nube para almacenar y compartir los datos, hacer análisis en mayor profundidad, generar informes con datos agregados, etc.

  La figura 4.8 muestra la interfaz de la aplicación móvil de Mobile Assay.

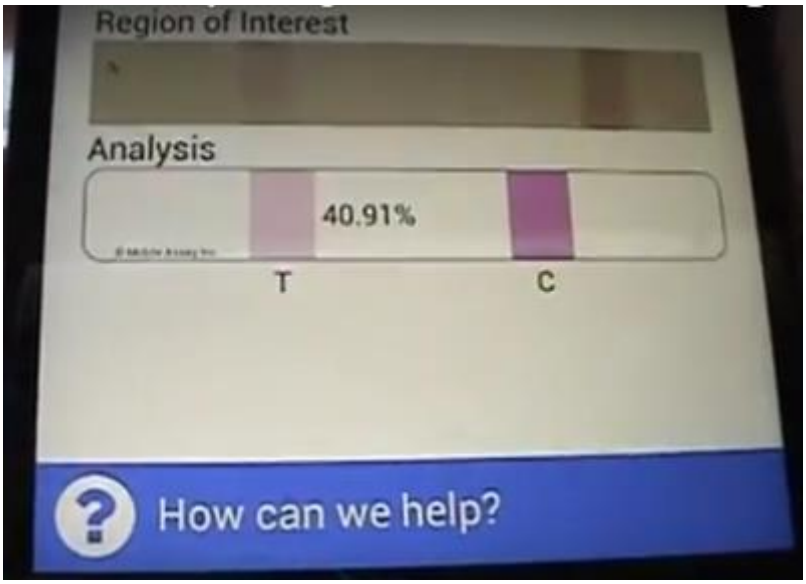

Figura 4.8. Aplicación móvil de Mobile Assay analizando una tira reactiva.

En [25] se enlaza un vídeo donde se puede ver la aplicación móvil realizando un test completo de una tira reactiva.

5 Solución Implementada

5.1 Arquitectura de la Solución

Este proyecto de fin de carrera implementa una aplicación web que reside en un servidor y que expone una API de servicios web REST totalmente interoperable a través de la cual es capaz de recibir imágenes de una aplicación cliente que se comunique a través de Internet, procesarlas haciendo uso de algunos de los algoritmos de procesamiento de imágenes que cargará en tiempo de ejecución como Plug-ins y devolver el resultado de este procesamiento indicando el riesgo de enfermedad de la piel al cliente.

Para conseguir la extensibilidad dinámica deseada en la aplicación de servidor se implementa el patrón de diseño de software Plug-in, como solución concreta dentro del marco conceptual de IoC y DI descrito en profundidad en las subsección 3.1.2 y 3.1.4 respectivamente. La implementación de este patrón se apoya, además, en Programación Orientada a Atributos, cuyos principios se describen, a su vez, en la subsección 3.1.6.

Toda la comunicación entre el cliente y el servidor se establece a través del protocolo HTTP, en formato JSON. El intercambio de las imágenes entre ambos extremos se realiza mediante la serialización de esta a Base64 y el empaquetado de la misma en JSON.

En la figura 4.1 se muestra un esquema de la arquitectura de esta solución con el objetivo de aclarar el flujo que se produce entre cliente y servidor y los principales componentes que intervienen. El cliente envía la imagen serializada en formato JSON al servidor comunicándose a través de la API que este expone. El servidor procesa la imagen con uno de los plugins disponibles en tiempo de ejecución y devuelve la respuesta en JSON al cliente.

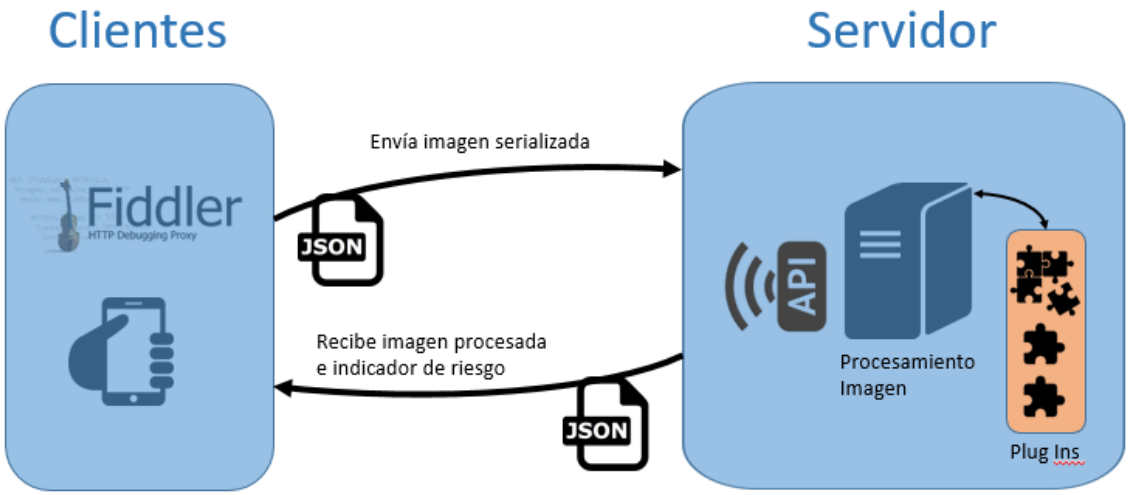

Figura 5.1. Esquema cliente – servidor de la solución implementada.

En el proyecto se ha utilizado Fiddler como cliente. Cualquier aplicación (una aplicación móvil, por ejemplo) que pueda interactuar con una API REST via HTTP podrá hacer de cliente con la aplicación web del servidor.

5.1.1 Solución Multicapa

La aplicación del proyecto ha sido desarrollada utilizando varias capas lógicas, lo que es imprescindible para poder lograr la separación de responsabilidades (SoC) necesaria para poder gestionar un escenario de dependencias tan poco acoplado como requiere el patrón Plug-in, utilizado para dotar de extensibilidad dinámica a la solución.

La figura 4.2 muestra el diagrama de capas lógicas del proyecto, en el que se representan tanto las capas conceptuales que componen la aplicación como el lugar o capas físicas en donde se ejecutan (cliente o servidor).

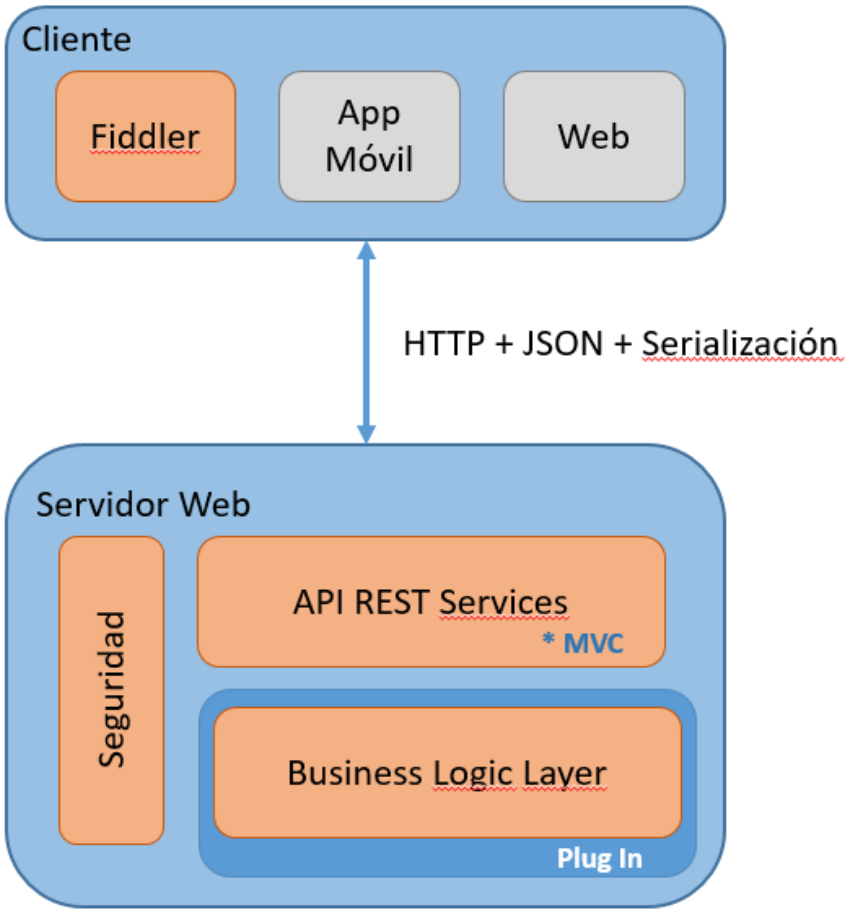

Figura 5.2. Diagrama de capas lógicas de la solución implementada.

En el lado del cliente cualquier aplicación (web, móvil, de escritorio, etc.) capaz de consumir una API REST estándar, funcionando sobre protocolo HTTP y utilizando JSON como formato de intercambio de datos podrá comunicarse con la aplicación que se ejecuta en el servidor. En el caso de este proyecto se utiliza la aplicación Fiddler [26], una herramienta estándar y bien conocida en el mundo del Desarrollo de software y que se describe en la subsección 5.3.6 como cliente. Precisamente Fiddler garantiza que la comunicación con la API del servidor es completamente estándar, al ser una herramienta agnóstica tecnológicamente.

Por su parte, en el servidor se encuentra precisamente la capa con la API de servicios web REST, la capa de lógica de negocio (en este caso formada por los analizadores de imágenes) y una capa de seguridad transversal a ambas, que en este proyecto se encarga básicamente de la autenticación de las peticiones que recibe la API. Sin embargo, esta capa básica de seguridad crea también la base para poder implementar políticas de autorización a nivel de lógica de negocio al asignar un

usuario, con su rol, al thread principal de ejecución una vez que se autentica correctamente una petición.

Además de esto, es interesante conocer que la implementación de la capa de la API REST sigue un patrón MVC, que se describe en la sección 3.4.

Asimismo, la particularidad de la capa de lógica de negocio es que se trata de una capa que se incorpora a la aplicación en tiempo de ejecución en la forma de un Plug-in. Es decir, esta capa no está compilada en la aplicación principal de servidor (que contiene la API REST), sino que de enlaza en tiempo de ejecución a través del patrón de Plug-in que es uno de los principales elementos diferenciales del diseño de software de este proyecto y que se describe en detalle en la subsección 3.1.5.

5.1.2 Implementación del Patrón Model View Controller

En este proyecto la API REST está construida con el patrón MVC, siendo la vista realmente los servicios web REST que están publicados vía HTTP para que un cliente pueda interactuar con ellos. En este caso, por tanto, la interfaz no es visual, sino que son servicios web. Será el cliente que los consuma quien realmente cree una interfaz gráfica (aplicación móvil, por ejemplo) para mostrar los datos a un usuario final.

Las controladoras recogen las peticiones web que entran a través de los servicios web REST y ejecuta acciones sobre el modelo, que en este caso son los algoritmos que se aplican a las imágenes para su procesamiento. Al terminarse este procesamiento se genera la respuesta del servicio web con la imagen modificada y si se ha encontrado riesgo en ella o no. Esta es la actualización de la vista, es decir, la interfaz que expone la aplicación.

El enrutamiento de una petición HTTP hacia una controladora lo realiza el motor de ASP.NET de manera automática según las reglas de enrutamiento definidas en el fichero de configuración "./App_Start/RouteConfig.cs", lo que permite exponer una API con URL bien construidas para gestionar con las controladoras adecuadas las peticiones que se reciban por cada URL de forma unívoca.

5.1.3 Seguridad

La capa de seguridad es transversal, intersectando a las otras dos capas lógicas de la aplicación que se ejecutan en el servidor, y entra en acción tan pronto como se recibe la petición HTTP a través de los servicios webs expuestos por la API REST. Es en este momento cuando se autentica la petición y se le concede permiso para continuar avanzando a través de la lógica de la aplicación o no. Para esto, se inspeccionan las credenciales del cliente que ha realizado la petición y se comprueba si están permitidas.

A partir de aquí, se genera un usuario con un rol y se asigna el mismo al thread que ejecuta la aplicación, pudiéndose, a partir de ese momento, establecer políticas de autorización concretas para acceder a recursos dentro del resto de la aplicación.

En la implementación realizada para este proyecto, y dado que no está en el alcance del mismo el implementar un sistema completo de seguridad, sólo es realmente utilizada la parte de autenticación dentro de las capacidades de seguridad disponibles; se sientan, eso sí, las bases para establecer una seguridad basada en usuarios y roles. Dependerá de futuras modificaciones y/o enlaces con otros proyectos la definición e implementación de políticas concretas de autorización.

El funcionamiento de la autenticación y su implementación concreta se describen en detalle en la subsección 5.3.5.

5.2 Tecnologías Utilizadas

En esta sección se describen las tecnologías de desarrollo utilizadas en el proyecto, así como la justificación de su elección.

5.2.1 .NET

.NET [27] es una Plataforma de Desarrollo fabricada por Microsoft que proporciona los elementos para desarrollar aplicaciones modernas de extremo a extremo y capaces de cubrir cualquier categoría de proyectos: aplicaciones de escritorio, web, servicios web, etc. .NET se compone de los siguientes elementos:

- **Common Language Runtime (CLR):** es una máquina virtual en la que se ejecuta el código objeto de las aplicaciones .NET. En esta ejecución, a su vez, el código CLR se transforma, mediante un proceso de compilación Just-In-Time

[28], en código máquina nativo del procesador. Por tanto, el código .NET no es nativo, sino que se interpreta en el CLR, quien lo compila finalmente a nativo.

Este enfoque tiene ventajas e inconvenientes. La principal ventaja se basa en que el código es gestionado (Managed Code) [29]. El CLR proporciona gestión automática de memoria, seguridad de tipos, manejo de excepciones y recolección de basura. Todas estas funcionalidades suponen un valor añadido importante que hacen que las labores de "fontanería" de la programación sean transparentes al desarrollador, pudiendo concentrarse este en código lo más relacionado posible con el problema que intenta resolver. A modo de ejemplo, la gestión de memoria en C/C++, mediante peticiones de memoria (malloc, calloc, etc.) y sus correspondientes liberaciones (free) es una de las labores que no aportan valor añadido en la resolución de un problema concreto y que, sin embargo, más errores introduce en tiempo de ejecución, siendo muy frecuentes los memory leaks. Esto no sucede con el modelo de programación de .NET, ya que el CLR se encarga de esta gestión automáticamente, disparándose así la productividad.

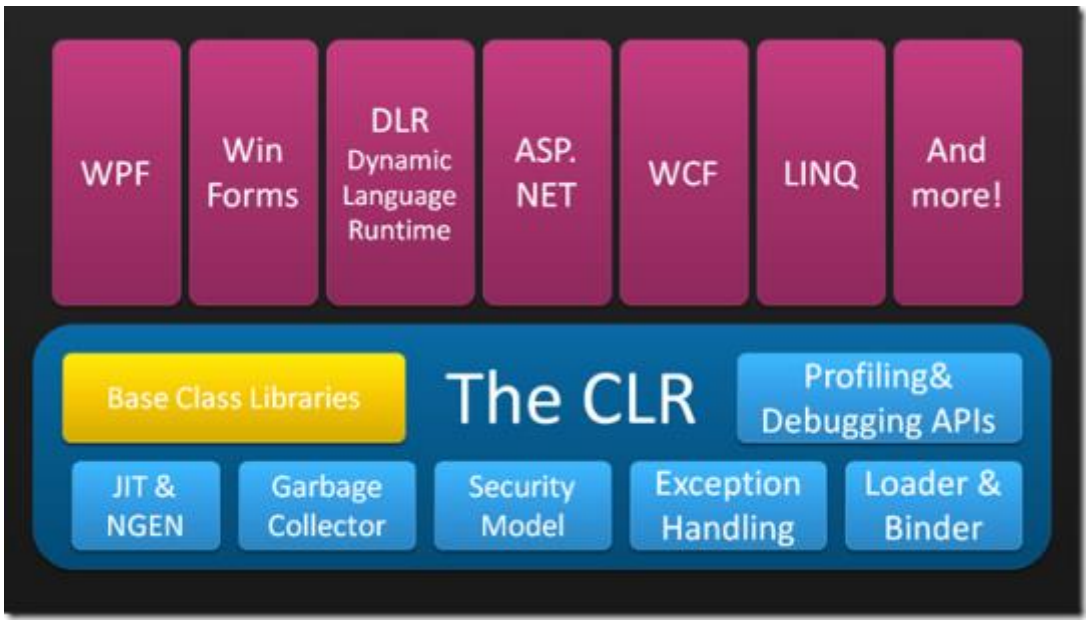

Figura 5.3. CLR y sus principales componentes.

- **Common Intermediate Language (CIL):** el CLR ejecuta un código objeto intermedio que se denomina CIL [30] y que proviene de la compilación del código fuente generado por cualquiera de los lenguajes que están soportados en la plataforma .NET.

- **Lenguajes:** .NET soporta actualmente más de veinte lenguajes [31], entre los que destacan C# (descendiente directo de C++ y muy similar a Java; es el lenguaje más utilizado en .NET), Visual Basic .NET (VB.NET), F# (lenguaje multiparadigma que soporta programación funcional y orientada a objetos), L# (implementación de Lisp), A# (implementación de Ada), C++/CLI (implementación de C++ que permite la invocación de clases de .NET y se puede compilar de manera mixta a código gestionado y nativo del procesador), Eiffel, etc.

  Common Language Infrastructure (CLI) y Common Languaje Specification (CLS): todos los lenguajes soportados por la plataforma .NET cumplen la especificación CLS, que define un subconjunto de tipos básicos y las reglas para operar con ellos. De esta manera, todos los lenguajes que cumplen CLS comparten una intersección de tipos y reglas. A su vez, todos los lenguajes se deben adherir también a lo especificado en la CLI, que es una especificación abierta diseñada por Microsoft que define el formato del código ejecutable y del entorno de ejecución (CLR). Todo esto da como resultado que el CLR puede ejecutar código CIL generado por la compilación de cualquiera de estos lenguajes. Esto significa, a su vez, que es posible desarrollar aplicaciones que mezclen librerías de distintos lenguajes, según convenga, ya que al compilarse se generará CIL, que será interpretado por el CLR independientemente de su lenguaje de origen.

  La figura 5.4 muestra el flujo de compilación de una aplicación .NET, en el que, como ya se ha mencionado, cualquier aplicación escrita con cualquier lenguaje soportado por la plataforma .NET es compilada al lenguaje intermedio común, CIL, y este a su vez es compilado por el CLR específico de la arquitectura sobre la que se ejecuta en código máquina de dicha arquitectura

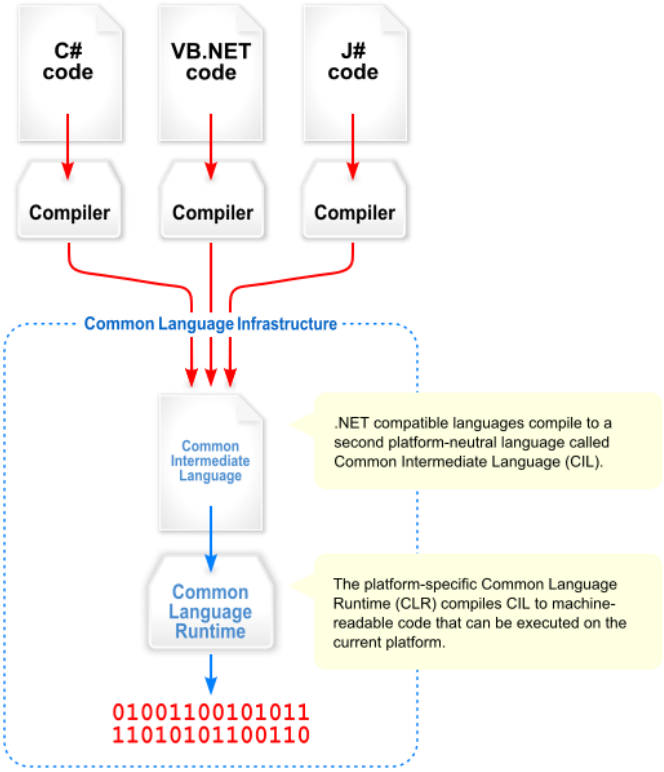

Figura 5.4. Esquema de compilación de .NET.

Framework Class Library (FCL) y Base Class Library (BCL): entre el entorno de ejecución (CLR) y los lenguajes CLI se encuentran la FCL [32], la BCL [33] y otros conjuntos de clases agrupados en conjuntos que definen funcionalidades determinadas (programación gráfica, acceso a base de datos, colecciones, etc.), que son las librerías que contienen todas las clases disponibles de manera estándar al desarrollar en .NET. La figura 5.5 muestra la Base Class Library y los espacios de nombres en los que se organiza según las funcionalidades que aportan las clases.

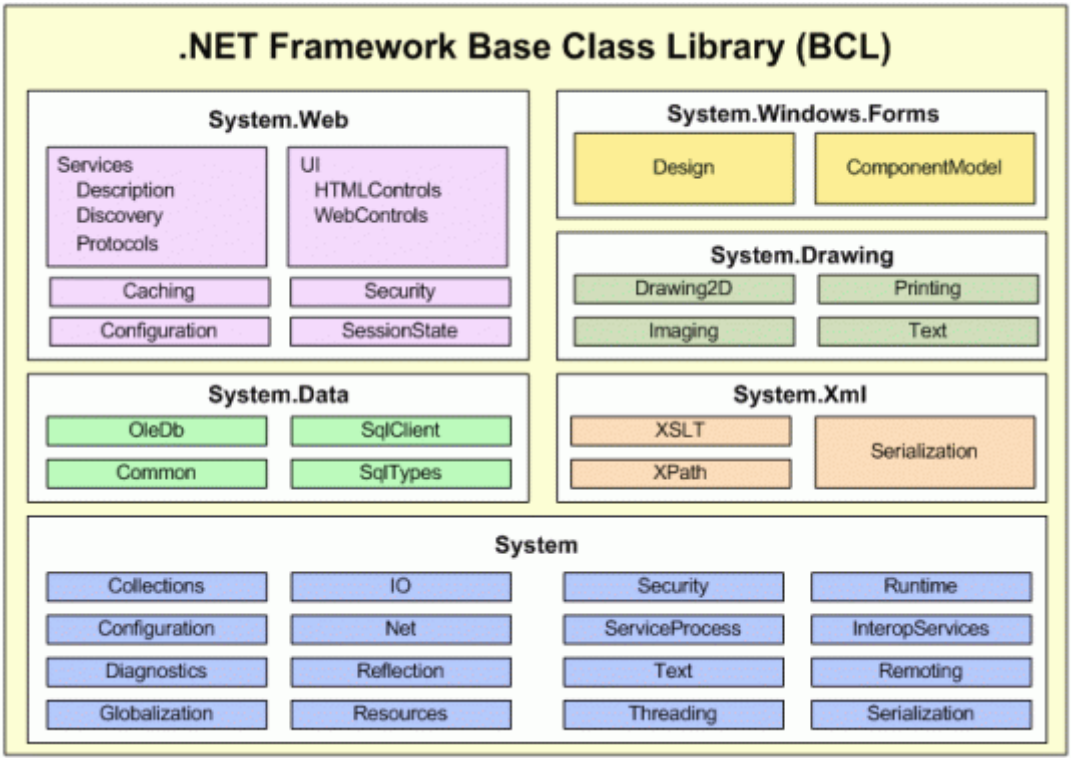

Figura 5.5. Base Class Library de .NET.

Visual Studio: es el Integrated Development Environment (IDE) que sirve de herramienta de desarrollo para codificar, depurar, compilar y desplegar aplicaciones desarrolladas en .NET con cualquiera de los lenguajes disponibles. Adicionalmente, en algunas versiones, incluye un gestor de versiones de código propio y también se integra con repositorios basados en Git como GitHub [34]. Visual Studio también permite el diseño y ejecución de tests unitarios. Visual Studio se describirá en detalle más adelante en esta sección.

Otras características de .NET: el 12 de noviembre de 2014 Microsoft anunció .NET Core, un subconjunto de librerías del framework .NET, así como un entorno de ejecución (CLR) que permite ejecutar aplicaciones desarrolladas sobre este subconjunto en Mac OS y Linux, además de en Windows. Adicionalmente, el código fuente de .NET Core se liberó bajo licencia MIT, haciendo que .NET se convirtiera en un proyecto Open Source bajo el patronazgo de la .NET Foundation [35]. Esto, sumado a que Visual Studio cuenta con versiones gratuitas completamente funcionales hace que no existan barreras de entrada para desarrollar en .NET, ni por licenciamiento y precio ni por la imposibilidad de portar el código a otros sistemas operativos que no sean Windows, dos inconvenientes de esta tecnología en el pasado.

Alternativas a .NET: Java Platform, Enterprise Edition (Java EE). Java EE es la otra alternativa principal a nivel de plataforma de desarrollo completa que permita la fabricación de software profesional de todo tipo.

Se ha elegido .NET porque la destreza profesional del autor del proyecto es mayor con esta plataforma que con Java EE, si bien ambas son similares tanto en su conceptualización como en las capacidades de desarrollo que ofrecen, teniendo ambas innumerables proyectos de software, de todas las escalas, desarrollados en su plataforma.

5.2.2 C#

El lenguaje de programación C# es un lenguaje orientado a objetos puro, que pertenece a la familia de lenguajes C, C++ y Java, siendo su sintaxis muy similar a la de este último, aunque posiblemente algo más clara y directa. C# se utiliza para desarrollar aplicaciones para la plataforma de desarrollo .NET. En esta plataforma, la compilación de C# se lleva a cabo cuando se necesita el código en la ejecución del programa, lo que se conoce como Just-In-Time Compiling (JIT). Esta política de diseño aumenta el rendimiento de C# frente a lenguajes interpretados.

C# se crea en el año 2000, siendo por tanto 5 años más moderno que Java, lo que propició que pudiera incorporar características heredadas de C y C++, así como conceptos novedosos que debutaron en el mundo de la programación con Java.

Las principales características de C# son la orientación a objetos pura (absolutamente todo en C# debe estar contenido dentro de una clase), la gestión automatizada de memoria (Garbage Collection), detección y gestión de errores estructurada y extensible mediante la Gestión de Excepciones, así como es un lenguaje fuertemente tipado, lo que hace imposible utilizar variables no inicializadas, acceder a posiciones de un array más allá de su dimensión definida o hacer typcasts sin comprobaciones. Por tanto, como se puede ver, uno de los objetivos principales de C# es la robustez y el disminuir la probabilidad de cometer errores tan presente en C y C++.

La especificación de C# se puede descargar en [36].

5.2.3 ASP.NET WEB API

HTTP es actualmente un estándar de facto en cuanto a las comunicaciones sobre redes se refiere. Su uso está absolutamente extendido y cualquier plataforma de desarrollo tendrá una liberaría que permita gestionar peticiones HTTP. Por tanto, y más allá de uso universal como protocolo que entrega los datos que finalmente acaban siendo páginas web HTML en los navegadores de los usuarios, HTTP también es la pieza clave de las API de servicios y datos que las aplicaciones web modernas exponen para ser consumidos por cualquier tipo de aplicación cliente, ya sea el mencionado navegador web, un smartphone, una aplicación de escritorio tradicional o una aplicación de terceros que embeba la funcionalidad expuesta por la API en su propio código.

ASP.NET Web API [37] es un framework para desarrollar APIs de servicios basadas en el intercambio de información a través del protocolo HTTP. Web API se encuentra integrado en la plataforma de desarrollo .NET y provee al desarrollador de un modelo de programación HTTP-céntrico, dando todo el control sobre una petición HTTP, así como sobre los tipos en los que leer y devolver los datos que se piden a la API. De esta manera, Web API permite la creación de completas APIs de forma sencilla, tanto bajo un enfoque RESTful como bajo un enfoque clásico y utilizando un intercambio de datos estandarizado mediante formatos como JSON o XML.

5.2.4 Managed Extensibility Framework (MEF)

MEF es una librería integrada en .NET desde su versión 4.0 (primero existió compro liberaría especializada e independiente de las clases de .NET) diseñada para crear aplicaciones extensibles, proporcionando a los desarrolladores la capacidad de descubrir y cargar extensiones de código, que proporcionarán nuevas funcionalidades al código principal, sin requerir programación ad hoc. De la misma manera, permite a los desarrolladores de extensiones encapsular su código de una manera robusta y reutilizable que evita las dependencias fuertes de código y la rigidez que esto introduce en un programa.

MEF ayuda a implementar de manera elegante el patrón Plug-in con el que se resuelve en este proyecto el problema de la extensibilidad de aplicaciones. El funcionamiento de MEF se basa en dos conceptos fundamentales, la composición,

que es la manera en la que el motor de MEF es capaz de descubrir los componentes que representan extensiones válidas para el programa en el que se ejecuta, y los propios componentes, denominados "parts" en la jerga de MEF. Estos componentes encapsulan y empaquetan las funcionalidades con las que se va a extender el programa. Adicionalmente, estos componentes pueden definir puntos de extensión para sí mismos, que MEF se encargará también de resolver descubriendo extensiones disponibles que casen con los requisitos necesarios. De esta manera, la extensión de un programa base con funcionalidades que se conectarán al mismo en tiempo de ejecución se puede hacer de manera dinámica sin requerir cambios en el código ni configuraciones extra, permitiendo a un programa crecer y añadir funcionalidades en un futuro que no se conocen o no se definen hoy. Esto es posible simplemente definiendo puntos de extensión que serán satisfechos por componentes desarrollados en el futuro por los mismos desarrolladores de una aplicación o por terceros, por ejemplo: expertos en determinados tipos de problemas cuyo conocimiento se quiere integrar en un programa que realiza además otras funciones.

Entrando algo más en detalle, en MEF, los componentes definen sus dependencias (imports) y las capacidades que ofrecen (exports). Con estos conceptos, concretados como metadatos en el código fuente a través de un enfoque de Programación Orientada a Atributos, MEF es capaz de descubrir y casar dependencias con componentes que las satisfacen.

5.2.5 Visual Studio

Visual Studio es el entorno integrado de desarrollo (Integrated Development Environmet, IDE) creado por Microsoft para desarrollar aplicaciones para su plataforma .NET.

En 2015 Microsoft ha liberado una versión gratuita de Visual Studio, Visual Studio Community 2015, completamente funcional con todas las funcionalidades clásicas de Visual Studio y con capacidad para desarrollar aplicaciones para .NET (incluyendo toda la variedad de aplicaciones que permite la plataforma), Android e iOS. Visual Studio Community está orientado principalmente a su uso por parte de desarrolladores individuales, centros educativos, startups y empresas en general de menos de 1 MM de euros de facturación y/o 250 estaciones de trabajo.

Adicionalmente, permite la utilización y compilación de múltiples lenguajes, desde C# a Python pasando por Javascript. Esta capacidad multiplataforma viene

propiciada por la adquisición por parte de Microsoft de Xamarin [38], empresa fundada por el líder del proyecto Mono (.NET para Linux) y antiguo CTO de Novell, Miguel de Icaza, que contaba con tecnología para desarrollar aplicaciones en .NET y C# que luego se compilaban a código nativo de iOS y Android, eliminando así la necesidad de escribir tres versiones de una misma aplicación para poder desplegar en los tres principales sistemas operativos de consumo de la actualidad. Con esta adquisición, las capacidades de Xamarin han quedado integradas nativamente dentro del IDE Visual Studio.

Adicionalmente, y como otros IDE avanzados, Visual Studio permite la utilización de gran número de extensiones para mejorar la productividad de los desarrolladores, como por ejemplo las extensiones para refactorización avanzada de código con mínimo trabajo por parte del desarrollador.

5.3 Implementación

En esta sección se describe en detalle la implementación de la solución propuesta anteriormente, que deberá cumplir con los principios de diseño expuestos, especialmente el de extensibilidad dinámica vía la implementación de un patrón Plug-in como caso específico de Inyección de Dependencias (DI), que se a su vez se encuadra en el marco más amplio del principio de Inversión de Control (IoC).

En concreto, se utilizan las clases dentro de .NET situadas en el namespace System.ComponentModel.Composition, que aglutinan el core de Managed Extensibility Framework (MEF), que servirá como tecnología base para la implementación del patrón Plug-in. Como se ha descrito anteriormente, en MEF los componentes definen sus dependencias (imports) y las capacidades que ofrecen (exports). Con estos conceptos, especificados como metadatos en el código fuente, MEF es capaz de descubrir y casar dependencias con componentes que las satisfacen.

5.3.1 Estructura de la solución

La solución se ha organizado en dos proyectos: uno que contiene la API Web REST que corre en el servidor y que está disponible para que los dispositivos móviles que envíen imágenes de la piel consuman sus servicios, así como un proyecto que contiene las extensiones que se acoplan en tiempo de ejecución al primer proyecto, validando la prueba de concepto de que, efectivamente, se ha logrado la extensibilidad del proyecto principal (el de la API que se ejecuta en el servidor).

Para esto en el segundo proyecto, el de los plug-ins, se ha decidido implementar algoritmos muy básicos de procesamiento de imágenes con el único objetivo de que se pueda elegir uno u otro en tiempo de ejecución y se vea cómo la extensión acoplada a la API varía. Para mostrar esto, se ha elegido procesar una imagen de entrada añadiéndole una marca de agua en un color determinado. Así, estas extensiones procesadoras de imagen reciben una imagen de entrada y devuelven la misma imagen con una marca de agua de cierto color, así como un booleano que simula el resultado de riesgo detectado en el análisis de la piel representada en la imagen. De esta forma sabemos que la imagen se ha procesado y se deja la puerta abierta a la extensión de este proyecto de fin de carrera para incorporar algoritmos complejos de análisis de anomalías en la piel representada en las imágenes como fruto de futuros proyectos de fin de carrera.

La figura 5.6 muestra la vista en Visual Studio con los dos proyectos que componen la solución.

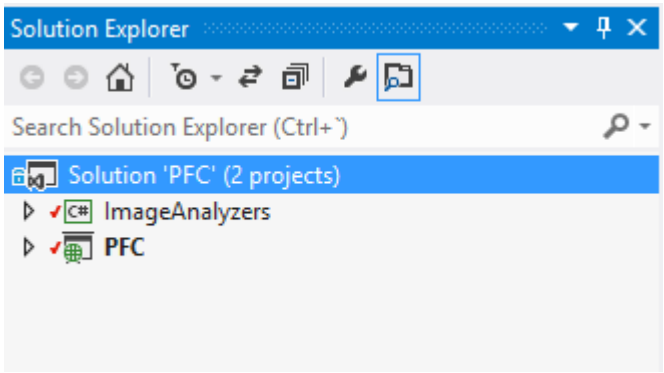

Figura 5.6. Solución en Visual Studio.

5.3.2 API Web REST

El código de la API REST reside en el proyecto PFC, que es un proyecto que sigue la estructura del patrón MVC, Model View Controller o Modelo Vista Controladora, utilizado ya por defecto y por diseño para todas las aplicaciones web, así como servicios web y, por tanto, API, desarrolladas en .NET con Visual Studio.

Como la aplicación del proyecto se trata de una API de servicios web REST, el proyecto simplemente contiene controladoras que reciben las peticiones y las enrutan a la extensión de procesamiento que corresponda, pero no vistas, ya que no existe interfaz gráfica que mostrar al usuario. Del mismo modo tampoco existen modelos típicos, ya que no se resuelve un problema de negocio que tenga un correlato real en un dominio determinado, sino que la API debe proveer de un servicio de analizado de imágenes a modo de utilidad. El modelo estaría compuesto en este caso por la lógica de los algoritmos de procesamiento de imágenes. No estamos, por tanto, ante una aplicación de negocio que resuelva, por ejemplo, la gestión de un concesionario de coches, donde encontraríamos clases definidas como modelos dentro de MVC que representarían entidades del dominio como “coche”, “operario”, “parte de trabajo”, “lista de averías”, “turno”, etc.

La figura 5.7 muestra el detalle de los ficheros que componen el proyecto principal de la solución: la API REST.

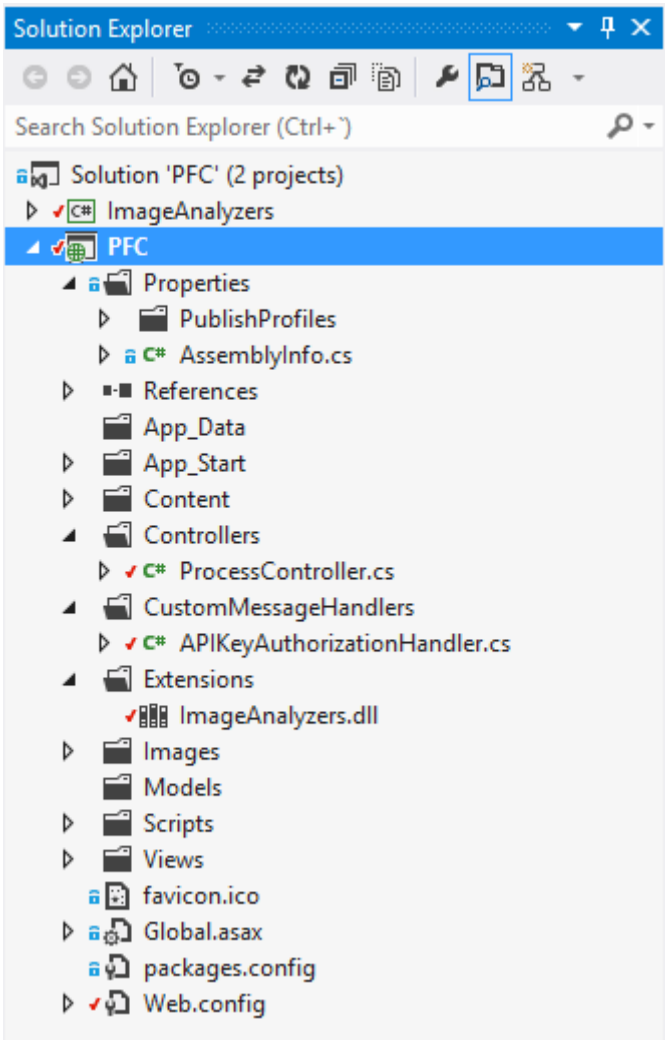

Figura 5.7. Vista detalle del proyecto de la API REST.

5.3.3 Flujo principal del programa

A continuación se explica el código del flujo principal del programa, donde se recibe mediante la API REST la petición enviada por un cliente móvil que contiene una imagen para analizar, se realiza el análisis y se devuelve la imagen procesada y un indicador sobre si existe riesgo o no para la salud como conclusión de dicho análisis.

La clase *ProcessController*, situada dentro de la carpeta *Controllers* se encarga de recibir la petición HTTP que contiene la imagen serializada en Base64 en el cuerpo de dicha petición y procesarla para averiguar si hay riesgo de enfermedad en la piel (en el caso del proyecto, se simula este análisis añadiendo una marca de agua a la imagen, como se ha comentado anteriormente).

La clase *ProcessController* hereda de *ApiController*, la clase específica dentro de ASP.NET Web API para las controladoras de una API de servicios web, no para una aplicación web estándar.

```
public class ProcessController : ApiController
{
```

Código 5.1. Declaración de la clase ProcessController

Dentro de esta clase se encuentra el método que gestiona la petición HTTP realizada con el *verb* POST. Este método se anota con el atributo [HttpPost] para indicar al motor de ASP.NET que sólo está disponible para peticiones POST. La combinación unívoca de la URI que identifica a la controladora dentro de la API, del verb de HTTP utilizado (GET, POST, PUT, DELETE, etc.) y de los parámetros pasados, en caso de utilizarse, hace que el enrutamiento de una petición al método correcto para procesarla dentro de la controladora sea único.

Método de la controladora	URI	HTTP Verb	Parámetro
AnalyzeImage	PFC/api/process	POST	-

Adicionalmente, el método también se anota con el atributo [Authorize], ya que vamos a autenticar y filtrar las peticiones HTTP que se realicen a la API desde los dispositivos móviles y sólo daremos acceso al servicio web de procesamiento de imágenes si el usuario que lo pide es un usuario autorizado. Anotando el método *AnalyzeImage* de esta manera podemos inyectar código de seguridad que realizará la autenticación de la petición HTTP justo en la recepción y procesado de la misma y ni siquiera la dejará pasar del primer paso en el pipeline de ASP.NET Web API si no se autentica correctamente. Con esto no sólo se consigue autenticar la petición, sino optimizar el uso de recursos del servidor, ya que la petición no llegará a la controladora ni se ejecutará su código si la autenticación realizada sobre el análisis de las cabeceras de la petición HTTP no arroja un resultado satisfactorio.

```
        /// <summary>
        /// Recibe la imagen a analizar como cuerpo de la HTTP Request y devuelve
        /// la misma procesada.
        /// Sólo es axccesible mediante una HTTP Request POST.
        /// Exige que la petición sea filtrada primero por el sistema de
        /// seguridad de ASP.NET, donde se autenticará.
        /// </summary>
        /// <param name="value">Contenido del body de la petición</param>
        /// <param name="request">Petición HTTP POST</param>
        /// <returns>
        /// Devuelve un objeto JSON compuesto por el campo Riesgo, que indica si
        /// el análisis ha revelado anomalías
        /// en la piel e Imagen, que devuelve la imagen procesada con marca de
        /// agua como prueba de concepto.
        /// </returns>
        [Authorize]
        [HttpPost]
        public dynamic AnalyzeImage(dynamic value, HttpRequestMessage request)
        {
```

Código 5.2. Definición del método AnalyzeImage()

Una vez autenticada la petición HTTP, esta se enruta hacia el método *AnalyzeImage*, como se ha comentado anteriormente. En la firma de este método se puede ver que recibe como parámetros un valor (con tipo dinámico, lo que permite que el objeto que envíen los dispositivos móviles al servidor pueda variar a lo largo del tiempo y futuras implementaciones, siempre y cuando contenga como mínimo las propiedades necesarias para el procesado de la imagen, como se verá a continuación), así como el mensaje que se ha recogido de la petición HTTP. Devuelve también un tipo dinámico.

```
        [Authorize]
        [HttpPost]
        public dynamic AnalyzeImage(dynamic value, HttpRequestMessage request)
        {
            string base64ImagenEntrada = value.imagen;

            string virtualPath =
                   System.Web.HttpContext.Current.Server.MapPath("../");

            // Catalog that can store a collection of catalogs
            var catalog = new AggregateCatalog();

            // Loads all the parts available in this assembley
            catalog.Catalogs.Add(new
            AssemblyCatalog(typeof(ProcessController).Assembly));
            catalog.Catalogs.Add(new DirectoryCatalog(virtualPath +
            "/Extensions"));

            CompositionContainer cc = new CompositionContainer(catalog);

            try
            {
                cc.ComposeParts(this);
            }
            catch (TypeLoadException tle)
            {
                throw;
            }
            catch (CompositionException ce)
            {
                throw;
            }
```

Código 5.3. Cuerpo del método AnalyzeImage()

Lo primero que se hace dentro del método es recuperar la imagen que viene en el cuerpo del mensaje HTTP dentro de la propiedad “imagen” del objeto “value” recibido. Para hacer esta recuperación se deserializa la imagen desde Base64 y se asigna a una variable de tipo string para poder manipularla a partir de ahora cómodamente en el servidor. Tal y como se detalle en la subsección 3.2.4 se ha elegido Base64 para la serialización porque es un estándar de facto que permite la total interoperatividad de sistemas, sea cual sea el cliente que consuma la API, ya que en prácticamente cualquier tecnología existen librerías para la serialización y deserialización en Base64. Conviene recordar, como también se detalla en la subsección 3.2.4 que se recibe la imagen serializada debido a que no se puede transmitir por Internet un formato binario puro, como es el caso de una imagen, y hacerla atravesar números elementos de red y sistemas desde el origen al destino garantizando su integridad.

A partir de aquí se entra directamente en el código de composición que da vida al patrón Plug-in. Se crea el catálogo de extensiones (parts) y se añaden al catálogo las que se encuentran dentro del ensamblado donde se encuentra la controladora de la API, *ProcessController*, así como del ensamblado que contiene las extensiones que serán "enchufadas" a *ProcessController* para extenderla. El proyecto que genera el ensamblado con dichas extensiones se describe más adelante.

Por tanto, se incluyen en el catálogo todas las parts que participan en el proceso de extensión tipo Plug-in, esto es: las parts que tienen dependencias (las del código de *ProcessController*, que necesitan que una extensión específica realice un trabajo concreto para completar el código) y las parts que exportan funcionalidad para satisfacer dependencias (las del proyecto de procesado de imágenes que se describirá más adelante). Finalmente se crea una instancia de *CompositionContainer*, quien se encargará de descubrir las dependencias entre todas las parts que se han cargado en el catálogo que recibe al instanciarse y satisfacerlas, casando dependencias con extensiones que las satisfacen. Este proceso se realiza apoyándose en la ayuda semántica proporcionada por la Programación Orientada a Objetos, tal y como se ha comentado anteriormente, mediante la cual se decora el código de un programa con atributos. En este caso, la semántica adicional añadida al código se circunscribe al contexto de la implementación del patrón Plug-in, añadiéndose así significado específico para que la tecnología subyacente, MEF, sea capaz de resolver las dependencias que plantea el programa.

Tras ejecutar la composición de parts, ya se han "enchufado" las extensiones correspondientes en las dependencias expresadas en el código principal de *ProcessCotroller*. En este caso concreto se habrá satisfecho la dependencia de un algoritmo concreto de procesado de imagen en la controladora *ProcessController* que expone la API. Este algoritmo se ha encontrado entre las extensiones o plugins detectados en el momento de la composición como parts que satisfacen la necesidad de esta dependencia de algoritmo de procesado de imagen indicada por *ProcessController* en su variable analyzer, de tipo *IAnalizer*. Tal y como se ha comentado, la dependencia concreta que tiene analyzer se especifica decorando la variable con el atributo que especifica el tipo concreto de la misma.

```
public class ProcessController : ApiController
{
    // Indica que tiene una dependencia de tipo IAnalyzer
    [Import(typeof(IAnalyzer))]
    public IAnalyzer analyzer;
```

Código 5.4. Variable (dependencia) del tipo IAnalyzer

Una vez satisfecha la dependencia, analyzer contiene un algoritmo de procesamiento de imagen válido y añadido en tiempo de ejecución, tal y como era el objetivo del proyecto. Es importante recalcar que el código principal no conoce nada acerca de dicho algoritmo, sólo sabe que cumple el contrato especificado por IAnalizer, lo que le garantiza que es un analizador válido. De esta forma se consigue poder "enchufar" al código principal del proyecto cualquier algoritmo concreto que se desarrolle en el futuro sin afectar al código del servidor.

Como paso final del flujo principal del programa, y teniendo ya un algoritmo válido de procesado de imagen en analyzer, se procede precisamente a pasar al algoritmo la imagen recibida por la API mediante la petición HTTP anteriormente autenticada y procesada, dando como resultado un string que representa la imagen recibida y codificada en Base64.

```
            string algorythm =
            System.Configuration.ConfigurationManager.AppSettings["Algoritmo"];

            string base64ImagenSalida = string.Empty;
            bool riesgo = false;

            analyzer.Analyze(base64ImagenEntrada, out base64ImagenSalida,
            algorythm, out riesgo);

            // Aquí se podría escribir una imagen en el disco duro
            //File.WriteAllBytes(virtualPath + "imagenDisco.jpg", binaryImage);

            var objeto = new { Riesgo = riesgo, Imagen = base64ImagenSalida };

            return objeto;
        }
```

Código 5.5. Recogida del algoritmo de procesado según lo configurado en el fichero externo XML. Invoca el método Analyze() que lo procesará con el plug-in que case con el algoritmo especificado, si está disponible. Tras el procesamiento se devuelve un objeto con el resultado que Web API formateará como JSON en la salida.

Este algoritmo de procesado se va a elegir, de entre todos los disponibles y válidos entre las extensiones cargadas en tiempo de ejecución, a través de un

parámetro de configuración que se almacena en un fichero XML de la solución. Este fichero es modificable en cualquier momento, no siendo necesario parar y volver a poner en marcha el servidor web ni de aplicación que está sirviendo la API, ya que al detectar este último cualquier cambio en el ese XML de configuración, recicla la aplicación y todas las peticiones que reciba la API a partir de ese momento serán enviadas al código de servidor que ya ha tenido en cuenta el cambio en el XML. El parámetro concreto que se define en el XML para elegir algoritmo de procesado es, precisamente "Algoritmo". En el XML se almacenan parámetros de configuración como este de una manera muy simple, en formato [clave, valor], de manera que junto con la clave "Algoritmo" se define el valor que se quiere para la misma. En el caso de este proyecto, será aquí donde dinámicamente se permita cambiar de algoritmo sin ni si quiera tener que resetear la aplicación.

Por tanto, se pone en marcha un esquema en el que se tienen a disposición del programa principal varias extensiones, cargadas en la fase de composición que da vida al patrón Plug-in y, además, se puede seleccionar específicamente cuál de esas extensiones modificar mediante la modificación de un parámetro en un XML de configuración. De esta manera se dota de agilidad a la solución a la hora de probar nuevos algoritmos de procesado de imagen.

Tal y como se ve en el código, se invoca el método dentro de analyzer que efectúa el procesado y análisis de la imagen en busca de riesgos. Este analizador será descrito más adelante en detalle junto con el proyecto que genera las extensiones (plugins) para cubrir las dependencias del proyecto principal.

Como paso final, se recoge la imagen devuelta por el método de procesado, que contiene la marca de agua como prueba de concepto de que efectivamente ha sido procesada, así como el booleano que indica si hay riesgo para la salud detectado en el análisis de la imagen y se devuelven por HTTP mediante un objeto que contiene los datos "riesgo" e "imagen". Este objeto, como se ve, se crea ah hoc para aprovechar en formato JSON, descrito en detalle en la subsección 3.2.2, en el envío de la respuesta al cliente.

Adicionalmente, en este método principal de la controladora *ProcessController* figura un pequeño fragmento de código comentado que indica el mejor sitio donde podría almacenarse la imagen, así como los datos de riesgo sobre la misma, bien en disco, en base de datos o en cualquier otro medio de persistencia que se eligiera.

5.3.4 Extensiones: plug-ins de algoritmos de procesado

Una vez descrito el programa principal, situada en el proyecto PFC, se explica a continuación el código del proyecto *ImageAnalyzers*, que contiene la creación de algoritmos de procesamiento de imagen que se exportan como plugins válidos para satisfacer las dependencias del proyecto principal.

El proyecto tiene la estructura de la figura 5.8, en la que se pueden ver las clases dentro del directorio "PluginArchitecture", que son las que se utilizan para implementar el patrón Plugin. Las del directorio "Analyzers" son los algoritmos de procesamiento de imagen, los plugins propiamente dichos.

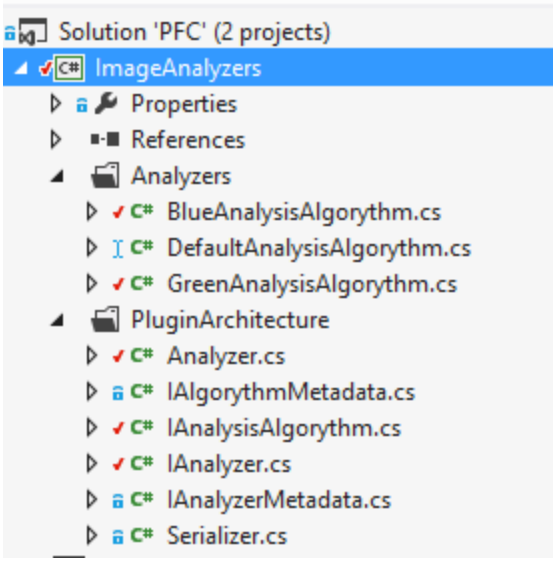

Figura 5.8. Vista detalle del proyecto ImageAnalyzers.

Dentro de la carpeta "PluginArchitecture" es donde residen todas las clases propias del patrón Plug-in que hacen posible la extensibilidad del proyecto, a través de la tecnología concreta de MEF para satisfacer dependencias en tiempo de ejecución. Se describen a continuación las clases que posibilitan dicha arquitectura y el rol que juegan:

IAnalyzer Es la interface que debe implementar un analizador para ser utilizado por el programa principal. Define el contrato

	que debe cumplir el analizador para ser válido. En concreto, define un método que debe ser implementado por un analizador válido, cumpliendo con la definición de su firma. En el programa, se define un analizador, que es utilizado por el programa principal y el analizador es quien cargará los diferentes algoritmos de procesado de imagen.
IAlgorithmMetadata	Interface que define los metadatos que debe tener una clase para definirse a sí misma como un Plug-in válido. En este caso se define que todas las clases que implementen algoritmos deben tener un metadato llamado "Algorithm" para ser analizado por el motor de composición en tiempo de ejecución y realizarse la inyección de dependencias con el Plug-in adecuado.
IAnalysisAlgorithm	Interface que define los métodos que debe tener una clase que implemente un algoritmo de análisis para ser válido. En concreto, cualquier clase para analizar imágenes deberá implementar el método Process(), según se define en la firma, para ser un Plug-in válido.
Analyzer	Clase que representa el analizador que contiene al algoritmo concreto que analizará las imágenes en tiempo de ejecución.
Serializer	Clase que funciona como utilidad para serializar y deserializar una imagen en Base64.

La figura 5.9 muestra el diagrama de clases UML que describe las relaciones entre las clases e interfaces que componen el proyecto de los analizadores de imagen, es decir, de los plugins.

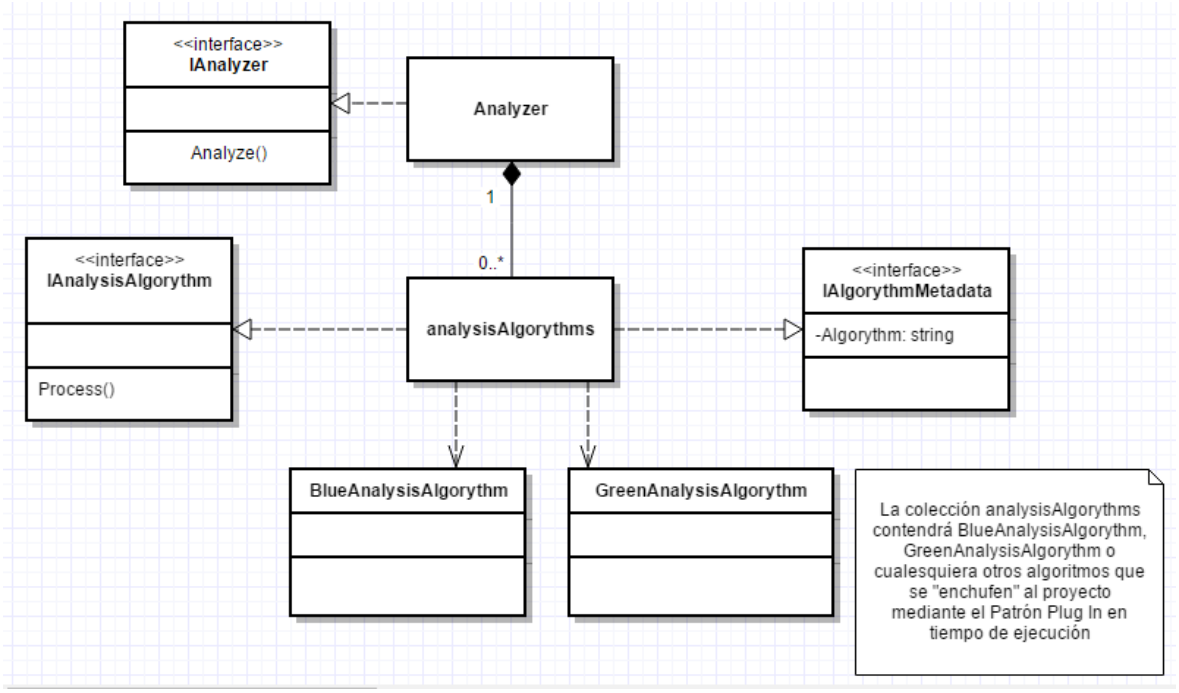

Figura 5.9. Diagrama de clases UML de ImageAnalyzers.

A continuación se muestran las figuras con el código más relevante de las clases e interfaces reseñadas en la figura 5.9. Todas estas clases e interfaces se encuentran en el espacio de nombres *ImageAnalyzers*.

```
namespace ImageAnalyzers
{
    public interface IAnalyzer
    {
        void Analyze(string base64ImagenEntrada, out string base64ImagenSalida,
        string algorythm, out bool riesgo);
    }
}
```

Código 5.6. Interfaz IAnalyzer. Define el contrato que debe cumplir la clase que vaya a definirse como la analizadora de las imágenes. Sólo debe cumplir tener un método Analyze() con la firma que se especifica en el código.

```
namespace ImageAnalyzers
{
    public interface IAnalysisAlgorythm
    {
        void Process(string base64ImagenEntrada, out string base64ImagenSalida,
        out bool riesgo);
    }
}
```

Código 5.7. Interfaz IAnalysisAlgorithm. Define el contrato que deben cumplir las clases que pretendan ser algoritmos de procesado de imágenes válidos. Obliga a que exista un método Process() con la firma especificada.

```
namespace ImageAnalyzers
{
    public interface IAlgorythmMetadata
    {
        string Algorythm { get; }
    }
}
```

Código 5.8. Interfaz IAlgorithmMetadata. Define el contrato que debe cumplir una clase que vaya a representar los metadatos de un algoritmo. En este caso se trata de contener simplemente un string de sólo lectura, ya que su función será aportar metadatos sobre una determinada clase.

```
namespace ImageAnalyzers
{
    // Indica que puede satisfacer una dependencia de tipo IAnalyzer
    [Export(typeof(IAnalyzer))]
    public class Analyzer: IAnalyzer
    {
        // Indica que tiene una dependencia que debe ser resuelta por MEF. Los
        algoritmos de análisis disponibles serán asignados por MEF a esta
        colección
        [ImportMany]
        IEnumerable<Lazy<IAnalysisAlgorythm, IAlgorythmMetadata>>
        analysisAlgorythms;


        /// <summary>
        /// Recibe una imagen, añade una marca de agua como prueba de concepto de
            que ha sido procesada y analizada, así como indica si hay riesgo para
            la salud en la misma.
        /// </summary>
        /// <param name="algorythm">Algoritmo que se va a utilizar para procesar
            la imagen y analizar si existe riesgo. Se cargará de entre las
            extensiones disponibles.</param>
        /// <param name="base64ImagenEntrada">Imagen que será procesada y
            analizada.</param>
        /// <param name="base64ImagenSalida">Imagen devuelta con marca de
            agua.</param>
        /// <param name="riesgo">Booleano que indica si se ha detectado riesgo
            para la salud en la imagen procesada y analizada.</param>
        public void Analyze(string base64ImagenEntrada, out string
        base64ImagenSalida, string algorythm, out bool riesgo)
        {
            bool procesado = false;
            base64ImagenSalida = string.Empty;
            riesgo = false;

            foreach (Lazy<IAnalysisAlgorythm, IAlgorythmMetadata>
            analysisAlgorythm in analysisAlgorythms)
            {
                if (analysisAlgorythm.Metadata.Algorythm.Equals(algorythm))
                {
                    analysisAlgorythm.Value.Process(base64ImagenEntrada, out
                    base64ImagenSalida, out riesgo);

                    procesado = true;
                }
            }

            if (!procesado)
            {
                DefaultAnalysisAlgorythm defaultAnalysisAlgorythm = new
                DefaultAnalysisAlgorythm();

                defaultAnalysisAlgorythm.Process(base64ImagenEntrada, out
                base64ImagenSalida, out riesgo);
            }
        }
    }
}
```

Código 5.9. Código de Analyzer. Se define una colección, analysisAlgorithms, con la particularidad de que se utiliza la técnica Lazy Loading, para no instanciarla hasta que sea referenciada por primera vez, ahorrando así memoria. El método Analyze() casa el algoritmo de procesamiento definido en el XML con el plug-in adecuado y procesa la imagen.

La clase *Analyzer* es la base del patrón Plug-in, donde se van a enchufar los distintos algoritmos de procesamiento en tiempo de ejecución, según se defina en el XML de configuración. Esta clase implementa la interfaz *IAnalyzer*, que como se puede observar en su código, obliga a incluir un método *Analyze()* que reciba una imagen serializada en Base64, el nombre de un algoritmo para elegirlo de entre los disponibles, así como devuelve una imagen serializada en Base64 tras haberla procesado y un booleano que indica si hay riesgo o no.

Como se puede ver, la clase *Analyzer* está decorada por el atributo *Export(typeof[IAnalyzer])* lo que indica que esta clase es una "part", una extensión que promete satisfacer dependencias del tipo *IAnlyzer*. En el código principal que ya se ha explicado anteriormente, se instancia un analizador de imagen precisamente de tipo *IAnalyzer*. Si aquella era la dependencia a satisfacer, ahora estamos viendo el otro lado del espejo, siendo esta la extensión que la va a satisfacer. Evidentemente puede ser esta implementación concreta o cualquiera otra que implemente correctamente *IAnalyzer*.

A continuación se declara una colección, analysisAlgorityms, de tipo *Lazy< IAnalysisAlgorithm, IAlgorithmMetadata>* y que cargará una dupla con los algoritmos que se encuentren en tiempo de ejecución. Los valores de la dupla son el propio código del algoritmo, de tipo *IAnalysisAlgorithm* y su metadato de tipo *IAlgorithmMetadata*. Esta colección se itera a través del iterador *IEnumerable<T>*, siendo T: *Lazy< IAnalysisAlgorithm, IAlgorithmMetadata>*. Cabe destacar que el uso de Lazy agrega un enfoque Lazy Loading al código que optimiza los recursos, en concreto memoria y tiempo de ejecución, ya que la colección no se va a cargar hasta que realmente se acceda la primera vez a ella. Es decir, aun cuando el código se ejecute y se pase por la instrucción que carga la colección, esto no se producirá, sino que se mantendrá vacía, sin incurrir en ningún coste de carga de datos y almacenamiento en RAM hasta que se produzca la primera referencia en el código que utilice realmente estos datos.

La clase *Analyzer* continúa con la definición del método *Process*(), tal y como exige la interfaz *IAnalyzer* que implementa. Lo principal de esta clase es que se recorre la colección descrita en el párrafo anterior, que contiene los algoritmos que se han encontrado en tiempo de ejecución que cumplen con *IAnalysisAlgorithm* y de los que se tiene también su metadato descriptivo *IAlgorytmMetadata*, y se selecciona el algoritmo cuyo metadato coincida con el parámetro algorithm que recibe el método *Process()*. De esta manera, y como hemos dicho que en última instancia ese

parámetro algorithm se ha leído previamente de un fichero de configuración XML, se tiene control sobre qué algoritmo, de entre los disponibles, se quiere seleccionar. Si no se eligiera ninguno el código del programa elige un algoritmo de análisis por defecto: defaultAnalysisAlgorithm. Así se asegura que siempre existe un algoritmo de procesamiento.

Una vez seleccionado el algoritmo que se buscaba, o bien una vez seleccionado el algoritmo por defecto, se procesa la imagen y se devuelve la imagen procesada y el riesgo como salida.

Dentro de la carpeta *Analyzers* encontramos las implementaciones concretas de varios algoritmos de procesamiento y análisis de las imágenes recibidas. Como se ha dicho anteriormente, el procesado en el caso de este proyecto consiste en añadir una marca de agua a la imagen de entrada para probar su procesamiento.

GreenAnalysisAlgorithm, *BlueAnalysisAlgorithm* y *DefaultAnalysisAlgorithm* son los tres algoritmos implementados. Se describe el código de uno de ellos, ya que los otros dos tienen un código prácticamente idéntico.

```
namespace ImageAnalyzers
{
    [Export(typeof(ImageAnalyzers.IAnalysisAlgorythm))]
    [ExportMetadata("Algorythm", "Blue")]
    class BlueAnalysisAlgorythm : IAnalysisAlgorythm
    {
        public void Process(string base64ImagenEntrada, out string
        base64ImagenSalida, out bool riesgo)
        {
            Image imagen = Serializer.StringToImage(base64ImagenEntrada);

            Graphics g = Graphics.FromImage(imagen);

            Font f = new Font("Arial", (float)14);

            Color c = Color.Blue;

            g.DrawString("procesado - Blue", f, new SolidBrush(c), new
            Point(imagen.Width / 5, imagen.Height / 5));

            MemoryStream ms = new MemoryStream();

            imagen.Save(ms, imagen.RawFormat);

            base64ImagenSalida = Convert.ToBase64String(ms.ToArray());

            riesgo = false;
        }
    }
}
```

Código 5.10. Código de BlueAnalysisAlgorithm, un algoritmo concreto de procesamiento de imagen. Se decora con atributos que asignan los metadatos "Algorithm" y "Blue" a la clase para que pueda participar como plugin en la arquitectura del proyecto. Implementa IAnalysisAlgorithm para cumplir con el contrato de un plug-in válido. El cuerpo del código simplemente añade una marca de agua a la imagen.

El código 5.10 muestra el código de *BlueAnalysisAlgorithm*, que implementa *IAnalysisAlgorithm* porque pretende ser un algoritmo de procesamiento de imagen a la manera que exige dicha interfaz, la clase se decora con los atributos *Export* y *ExportMetadata*. Con el primer atributo se define el tipo de dependencia que es capaz de satisfacer esta part *BlueAnalysisAlgorithm*. Obviamente este tipo es *IAnalysisAlgorithm*, que es el tipo de dependencia que tiene un *IAnalyzer* para conseguir un algoritmo de procesamiento de imagen, como ya se ha visto anteriormente. Por otro lado, el atributo que exporta los metadatos recibe los valores "Algorithm" y "Blue", lo que indica que esta part es el algoritmo azul. Esto servirá para buscar y descubrir específicamente esta extensión en el momento en el que se estén resolviendo las dependencias, como ya se ha visto justamente en el código del método *Process()* de la clase *Analyzer* anteriormente explicada.

Se está viendo ya el corazón de la resolución de dependencias y el mecanismo de enganche de las extensiones a las mismas.

El código que sigue a continuación el método no es de demasiado interés, ya que se implementa un método *Process()* en el que se manipula la imagen para añadir una marca de agua de color azul en una región determinada como prueba de concepto. Será en una clase como esta y en la implementación de un método *Process()* como este donde se podrían implementar algoritmos especializados de detección de anomalías en la piel.

5.3.5 Autenticación de la petición HTTP y acceso a la API

Antes de acceder a las funcionalidades descritas anteriormente para análisis de una imagen, una petición realizada por un cliente hacia la API debe autenticarse primero, como no podría ser de otra manera. El sistema de seguridad implementado busca la mayor simplicidad posible y, sobre todo, la manera óptima a la hora de autenticar la petición consumiendo el menor número de recursos posibles y dejando entrar a la petición lo mínimamente posible en el código de la API para autenticarla. De hecho, la petición realmente nunca llega a tocar el código de la API si no se autentica correctamente.

Para esto se ha decidido implementar el código de autenticación en un manejador de procesamiento de la petición HTTP propio que se añadirá al pipeline de procesamiento de la petición HTTP por parte de ASP.NET WEB API [39]. De esta forma se inyecta en este punto la autenticación a nivel de petición HTTP, evitando tener que hacerlo en el propio código de la controladora y, por tanto, evitando el progreso de la petición hacia el core del código de la API si no es necesario. Este manejador es *APIKeyAuthorizationHandler*, que implementa *DelegatingHandler*, la interfaz que deben implementar los manejadores personalizados que se quieran intercalar en el pipeline estándar de procesamiento de una petición HTTP en ASP.NET Web API.

El pipeline estándar de ASP.NET Web API cuenta con varios manejadores que procesan la petición. Estos son: *HttpServer*, que recoge la petición del servidor web, *HttpRoutingDispatcher*, que enruta la petición y *HttpControllerDispatcher*, que envía la petición a la controladora que corresponda. Tal y como se ha hecho en este proyecto con el objeto de optimizar la autenticación, se pueden añadir más manejadores al pipeline en algún punto del mismo como muestra la figura 5.10. En

esta figura se describe el pipeline de procesamiento de una petición HTTP por parte del motor de ASP.NET Web API.

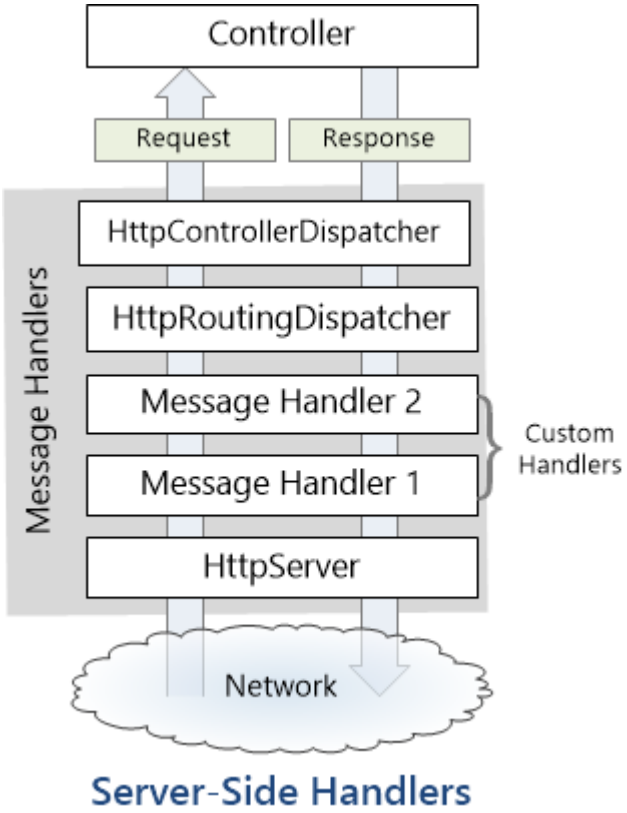

Figura 5.10. Pipeline de procesamiento de una petición HTTP.

La técnica que se ha utilizado para mejorar el proceso de autenticación se basa en intercalar un manejador personalizado en este pipeline, como se muestra en figura 5.11.

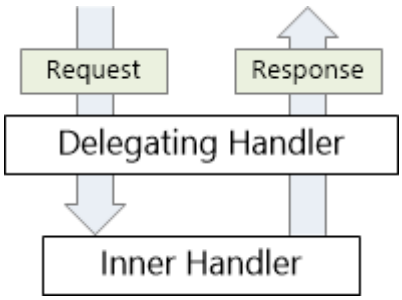

Figura 5.11. Intercalación de un manejador personalizado.

Para inyectar el manejador personalizado en el lugar adecuado se utiliza la clase WebApiConfig, situada dentro de la carpeta App_Start, donde se registra el manejador, como se muestra en el código 4.10.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        // Añade el custom Message Handler creado para la autenticación
        config.MessageHandlers.Add(new APIKeyAuthorizationHandler());
    }
}
```

Código 5.11. Se registra en el pipeline de procesamiento de las peticiones HTTP de ASP.NET Web API el manejador personalizado (Custom Message Handler): APIKeyAuthorizationhandler(), que se encargará de comprobar que la petición HTTP trae consigo la clave que le permitirá usar la API.

En el código de *APIKeyAuthenticationHandler,* que se muestra en el código 4.12, se sobrecarga el método *SendAsync()*, tal y como obliga el contrato definido por *IDelegatingHandler* y en él se distinguen claramente tres secciones de código. En la primera se inspecciona la petición para comprobar que se incluye la cabecera de autenticación API-KEY. Si no es así, inmediatamente se corta la ejecución y devuelve una respuesta al cliente con un código HTTP 403 Forbidden, indicando que el acceso a la API está prohibido, ya que no se tiene clave para accederla. Este tipo de petición podría realizarla cualquier cliente que tenga acceso a Internet (cualquier usuario que sea capaz de crear una petición HTTP a la URI de la API que se ha creado). Con este filtro descartamos de la forma más temprana posible el procesamiento de esta petición.

La segunda sección sigue analizando la petición sabiendo que sí contiene la cabecera API-KEY e inquiere en este punto sobre su contenido. Lo compara con el valor que se guarda en el fichero de configuración XML web.config y si no coinciden corta la ejecución devolviendo un código HTTP 401 Unauthorized.

Finalmente, si ha pasado los dos filtros anteriores, se asigna un *IPrincipal* al contexto de ejecución actual asignando un usuario y rol determinado que, en este caso se lee también del fichero web.config. En una versión de la aplicación en

producción en un entorno de negocio podría recogerse el usuario con su contraseña correspondiente de otra cabecera HTTP ad hoc y contrastarlo contra un sistema de seguridad que podría residir, típicamente, en una base de datos. A partir de este punto se permite a la petición seguir escalando por el pipeline de procesamiento de ASP.NET Web API de manera autenticada. Esto se hace invocando a *base.SendAsync()* para acceder al método *SendAsync()* del manejador padre del actual en el pipeline. El último manejador enviará la petición a la controladora adecuada, según la URI de la misma, como ya se explicó anteriormente.

```
public class APIKeyAuthorizationHandler : DelegatingHandler
{
    /// <summary>
    /// Analiza la petición HTTP y la autentica o cancela su ejecución sin
        dejarla llegar al código de la API.
    /// </summary>
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage
    request, CancellationToken cancellationToken)
    {
        // Inspecciona la petición y si tiene una cabecera "API-KEY" la
           procesa, si no devuelve un 403 Forbidden
        if (!request.Headers.Contains("API-Key"))
        {
            var response = new
            HttpResponseMessage(System.Net.HttpStatusCode.Forbidden);

            var tsc = new TaskCompletionSource<HttpResponseMessage>();
            tsc.SetResult(response);

            return tsc.Task;
        }

        // Lee el password del fichero XML de configuracion, web.config
        string apiKey = ConfigurationManager.AppSettings["API-KEY"];

        var valorAPIKey = request.Headers.GetValues("API-
                          Key").FirstOrDefault();

        // Si el valor de la API Key coincide deja pasar la Request al
           handler más interior y hasta la controladora, si no devuelve un
           401 Unauthorized
        if (valorAPIKey != apiKey)
        {
            var response = new
            HttpResponseMessage(HttpStatusCode.Unauthorized);

            var tsc = new TaskCompletionSource<HttpResponseMessage>();
            tsc.SetResult(response);

            return tsc.Task;
        }

        string role = ConfigurationManager.AppSettings["Role"];
        string user = ConfigurationManager.AppSettings["User"];

        // Asigna un Iprincipal al contexto de ejecución, con un usuario y un
           rol leido también del XML de configuración web.config
        IPrincipal principal = new GenericPrincipal(new
                               GenericIdentity(user), new string[] { role });
        Thread.CurrentPrincipal = principal;
        HttpContext.Current.User = principal;

        // Envía la request al handler que está más hacia el interior en el
           pipeline, que lo enviará a su vez a la controladora
        return base.SendAsync(request, cancellationToken);
    }
}
```

Código 5.12. Implementación del manejador personalizado. Inspecciona la petición HTTP entrante y comprueba que tiene la clave “1234” en su cabecera “Api-Key”. Si la tiene autentica la petición y asigna un usuario y rol genérico al thread principal. Si no corta la ejecución y envía una respuesta HTTP con los valores 401 Unautjorized ó 403 Forbidden.

5.3.6 Herramientas Utilizadas

En esta subsección se especifican y describen las herramientas utilizadas para poner en marcha el proyecto, así como la forma en la que se utilizan en el mismo.

Fiddler

Fiddler es una herramienta gratuita desarrollada por Eric Lawrence y adquirida por la famosa compañía de utilidades para desarrolladores Telerik. Eric Lawrence era un antiguo Program Manager del equipo de Internet Explorer, en Microsoft, incorporándose más tarde a Telerik tras la adquisición de Fiddler por esta compañía. Actualmente trabaja en el equipo de seguridad de Chrome, en Google.

Fiddler es una herramienta muy popular en el mundo del desarrollo de software profesional para Internet. Se trata de una aplicación utilizada para depurar y probar aplicaciones HTTP, para lo cual actúa como generador de peticiones HTTP de forma que las mismas se pueden configurar en detalle, tal y como se quiera antes de ser enviada por el método que se seleccione a un determinado servidor web mediante la especificación de una URI concreta. De igual manera, Fiddler es capaz también de recibir las respuestas HTTP del, permitiendo esto el que sea utilizado a modo de cliente web que envía peticiones a un servidor web, recibe las respuestas y las muestra desglosadas y en detalle de diversos formatos para facilitar la depuración.

El uso de Fiddler en este proyecto es fundamental porque es la herramienta agnóstica a cualquier tecnología concreta (.NET, Java, Android, iOS, etc.) que actúa como cliente de la API Web REST desarrollada y que, por tanto, actúa como prueba de concepto de que cualquier cliente que sea capaz de generar peticiones HTTP correctamente formateadas podrá utilizar la API objeto del proyecto. Por tanto, su uso garantiza un nivel de compatibilidad y adecuación al estándar superior a haber elegido el desarrollo de un pequeño cliente de prueba en una tecnología concreta para probar el funcionamiento de la API.

La figura 5.12 muestra la captura de una respuesta HTTP de servidor con código 200 (OK). El panel de la derecha arriba muestra las cabeceras de la respuesta y el panel inferior la respuesta en JSON.

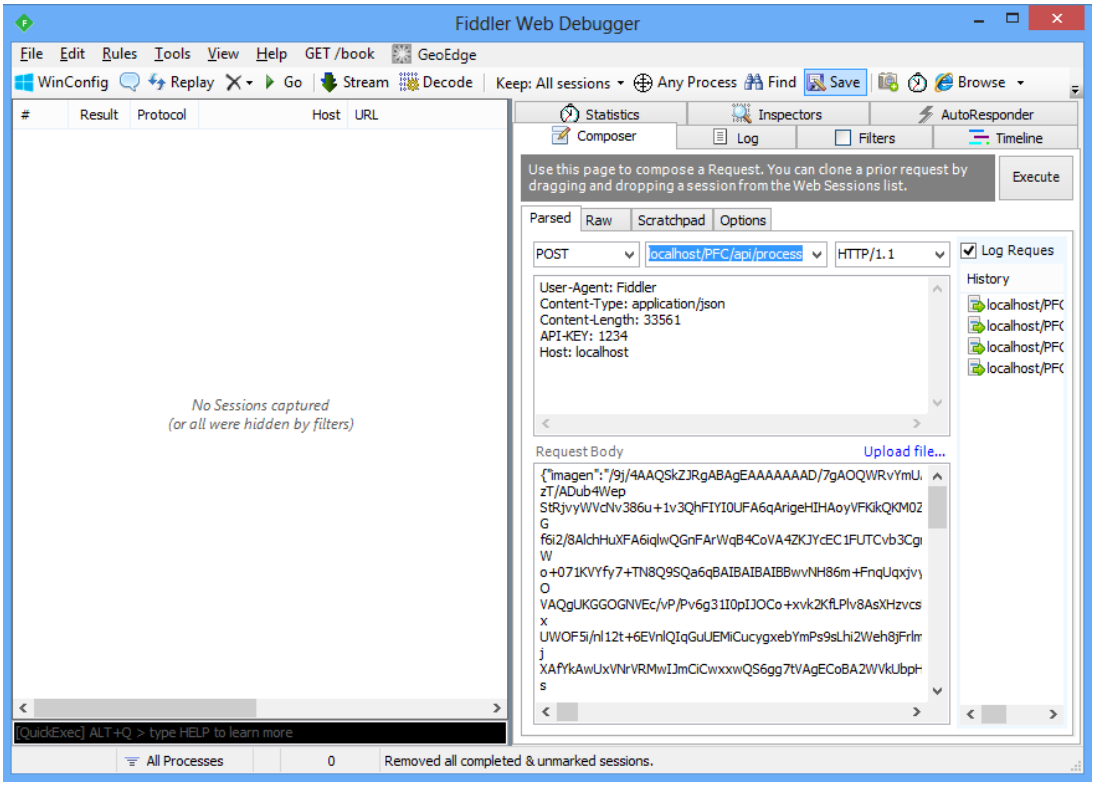

Figura 5.12. Captura de la respuesta del servidor en Fiddler.

La figura 5.13 muestra el Composer de Fiddler donde se crea una petición HTTP tanto a nivel de cabecera como de cuerpo de la petición y se especifica la URI a la que se va a enviar.

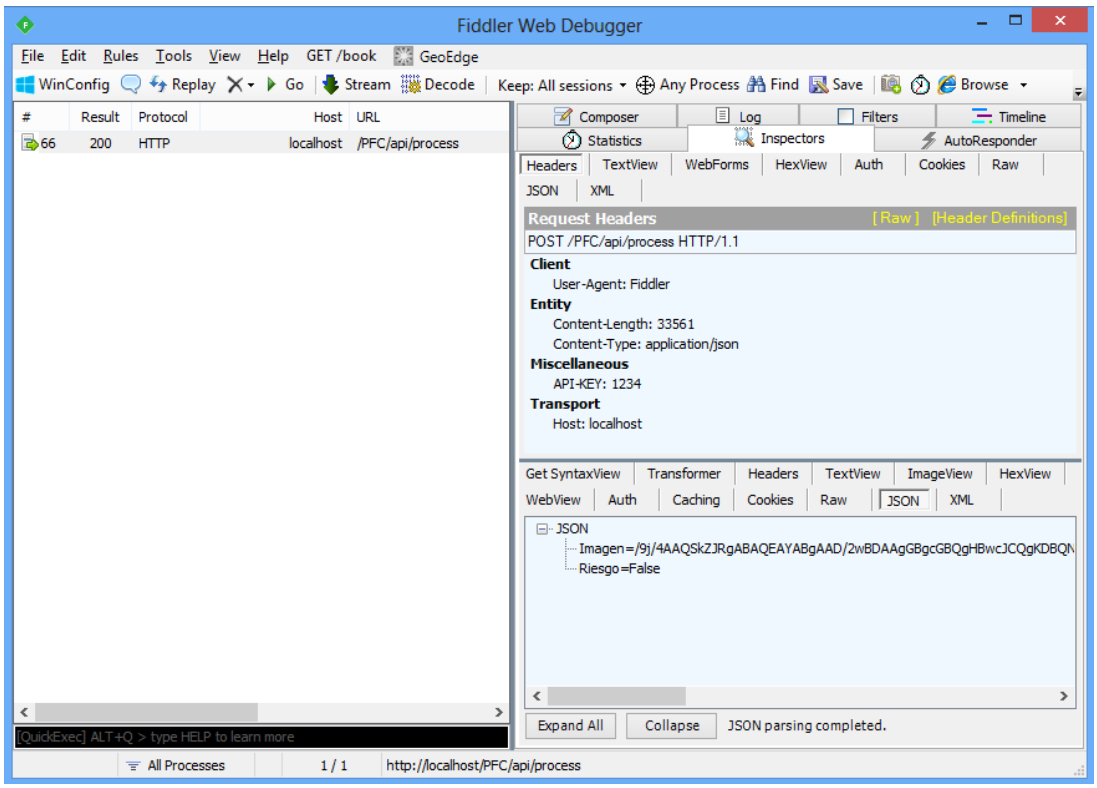

Figura 5.13. Composer de Fiddler.

Freeformatter.com

Freeformatter.com es una página web de Internet que pone a disposición de sus usuarios, de manera gratuita, conversores de formato de distinto tipo y online. En el caso de este proyecto, Freeformatter.com ha sido utilizado con profusión debido a que el formato en el que se serializan las imágenes para enviarse a la API, así como el formato de salida de las mismas tras su análisis y procesado es Base64. Por tanto, el codificador y decodificador a Base64 de Freeformatter.com ha sido de mucha utilidad.

La manera en que se ha utilizado esta herramienta ha sido introducir la imagen de salida procesada por la API del proyecto y codificada en Base64 en *freeformatter.com* para decodificarla y guardarla como fichero binario para poder abrirla posteriormente como imagen y comprobar el resultado del procesamiento.

En la figura 5.14 se muestra la ventana del codificador/decodificador online de *freeformatter.com*.

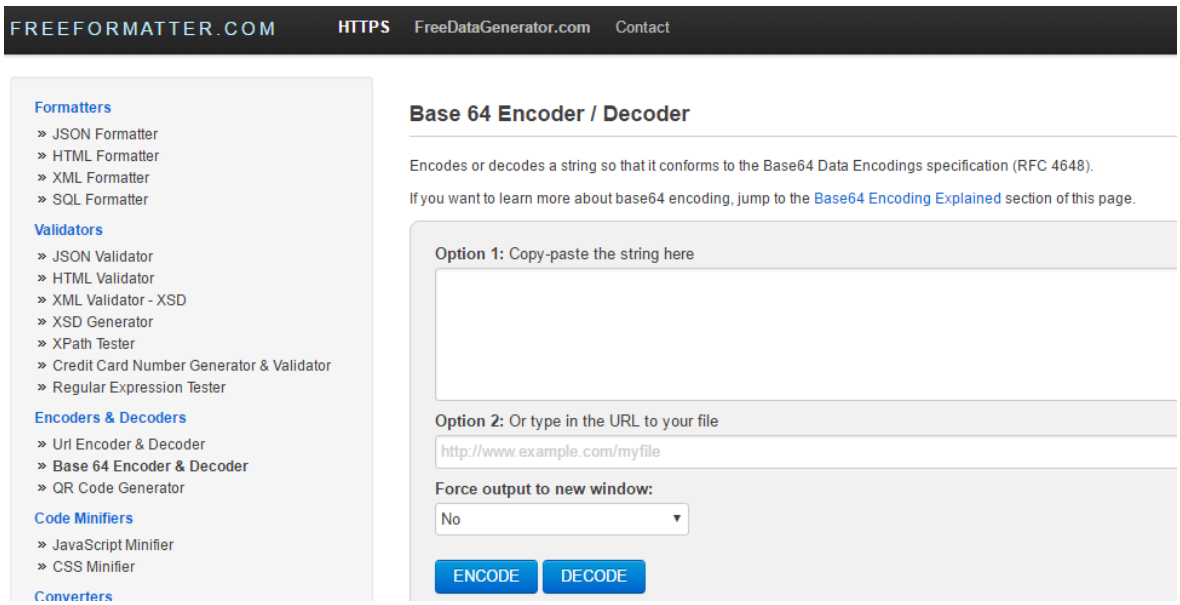

Figura 5.14. Codificador/Decodificador en Base64.

BASE64IMAGE.com

BASE64IMAGE es una herramienta online gratuita que permite obtener la codificación Base64 de una imagen binaria de una manera muy sencilla. Por tanto, ha sido la otra cara de la moneda respecto a la funcionalidad que aporta freeformatter.com, ya que en este caso se ha utilizado para generar la representación en Base64 de la imagen original que se envía al servidor para su procesamiento.

El funcionamiento de BASE64IMAGE.com es tan sencillo como subir a la web la imagen deseada, acción que se puede hacer mediante drang & drop, como se muestra en la interfaz de la figura 5.15 y se obtiene su representación en Base64, pudiendo copiarla para su uso posterior.

Figura 5.15. Interfaz web principal de BASE64IMAGE.

6 Evaluación y Resultados

En este capítulo se reflejan los experimentos realizados con el objetivo de evaluar el comportamiento de la implementación realizada al ejecutarla en varios escenarios reales. Se comparan los resultados obtenidos con los esperados respecto a que la solución cumpla su función y, a su vez, consiga las capacidades de interoperatividad, extensibilidad dinámica y optimización del intercambio de datos deseadas.

Todos los experimentos se han llevado a cabo en un entorno contenido en una máquina virtual sobre VMWare con Sistema Operativo Windows 8 Professional, Visual Studio 2012 como IDE para el desarrollo y la depuración, .NET Framework 4.5 como plataforma de desarrollo y ejecución, Internet Information Services (IIS) 8.0 como servidor web y Fiddler Web Debugger 4.6.2.0 como cliente REST.

6.1 Experimentos

6.1.1 Experimento principal

Experimento

El primer experimento consistió en probar el flujo básico de la solución. Se envió una imagen desde la aplicación cliente (Fiddler) al servidor web a través de la API REST; en el servidor se procesó la imagen añadiéndosele una marca de agua como prueba de concepto de que, efectivamente, se ha aplicado un algoritmo de procesamiento sobre la misma. La respuesta se envía de vuelta al cliente, donde se recupera, se deserializa para poder verla y se muestra comprobando que contiene la marca de agua. Adicionalmente, en el paquete de respuesta se obtiene también un valor para el indicador de riesgo de que el lunar de la imagen pueda ser un melanoma.

En la figura 6.1 se muestra, a modo de recordatorio, el flujo principal de la aplicación y su arquitectura: el cliente utilizado en este proyecto es Fiddler, pero podería ser una aplicación sobre cualquier plataforma móvil. El cliente envía la imagen serializada en formato JSON al servidor comunicándose a través de la API que este expone. El servidor procesa la imagen con uno de los plug-in disponibles en tiempo de ejecución y devuelve la respuesta en JSON al cliente.

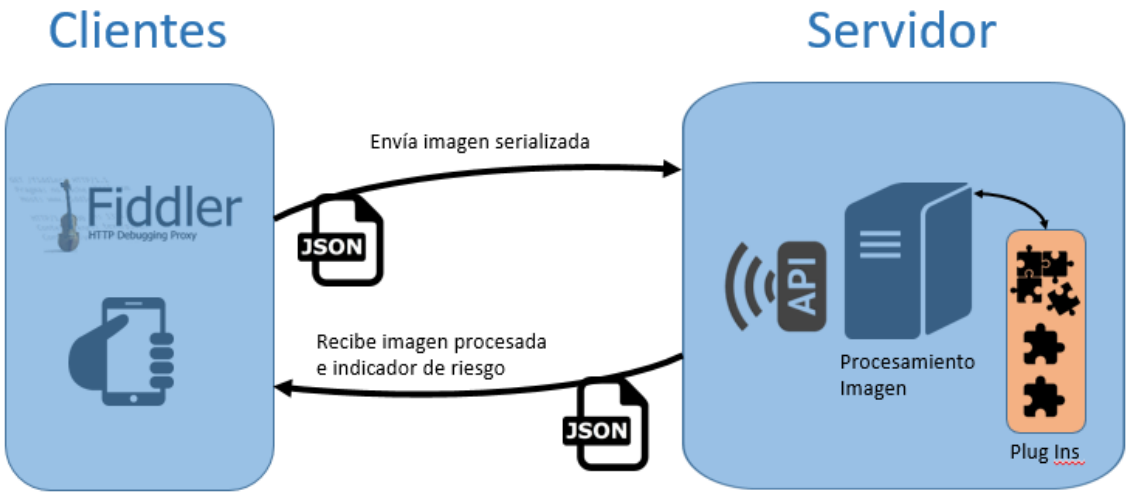

Figura 6.1. Esquema cliente – servidor de la solución.

La figura 6.2 muestra la imagen original sin procesar que se envía desde el cliente al servidor.

Figura 6.2. Imagen original sin procesar.

Para enviar la imagen al servidor a través de una petición HTTP POST primero es necesario serializarla al tratarse de un fichero binario, como se explicó en la subsección 3.2.4. Se utiliza la serialización en Base64, siendo un pequeño fragmento de la representación de la imagen en este formato la siguiente, que se añade como contenido del cuerpo de la petición HTTP POST:

/9j/4AAQSkZJRgABAgEAAAAAAAD/7gAOQWRvYmUAZAAAAAAB/+wAEUR1Y2t5AAEABAAAADwAAP/bAE
MABgQEBAUEBgUFBgkGBQYJCwgGBggLDAoKCwoKDBAMDAwMDAwQDA4PEA8ODBMTFBQTExwbGxs
cHx8fHx8fHx8fH//bAEMBBwcHDQwNGBAQGBoVERUaHx8fHx8fHx8fHx8fHx8fHx8fHx8fHx8fHx8fHx8fH
x8fHx8fHx8fHx8fHx8fH//AABEIAY4CEgMBEQACEQEDEQH/xAAcAAABBQEBAQAAAAAAAAAAAAAAAQID
BAUGBwj/xABREAABAwIEAgUIBwQFCwQCAwEBAAIDEQQhMRIFQQZRYXEEiE4GRobEyUnIHwUJisiMzFN
FTFTWCY3MkNvDhkqLCQ4M0JRYI8dKzRKNUZIRlF//EABcBAQEBAQAAAAAAAAAAAAAAAABAgP/xAAf
EQEBAQEBAAMAAwEAAAAAAAAAARECEiExQVFhA4H/2gAMAwEAAhEDEQA/APqfQz3R5ggPDZ7o8wQH
hs90eZAeGz3R5ggPDb7o8wQGhvuhAeG33R5ggPDZ7o8wQGhvQPMgxuZWtEVvQD2zw6lKsO5ea0smwB7
w9SkRr6Ge6PMtA0N90IDQ33R5kBob7o8yA0N90eZAaG+6PMgD0N90eYIOG5ow3uXqaynmUrUYz
urJZVw2+k/wAaugSTRw9QQim018iNHBA8IBA

Se construye la petición HTTP POST hacia la URL del servidor donde está expuesta la API REST de la aplicación para el procesamiento de las imágenes, tal y como se muestra en las figuras 6.3 y 6.4 respectivamente.

En la figura 6.3 se realiza la creación de la petición HTTP hacia la URL *http://localhost/PFC/api/process* con cabeceras que indican que el agente (cliente) que realiza la petición es Fiddler, que el contenido de la petición está en formato JSON, la longitud en bytes de la misma (que se corresponde con el número de bytes resultado de la serialización a Base64 de la imagen), la credencial que permite el acceso a la API REST del servidor (API-KEY) y el host.

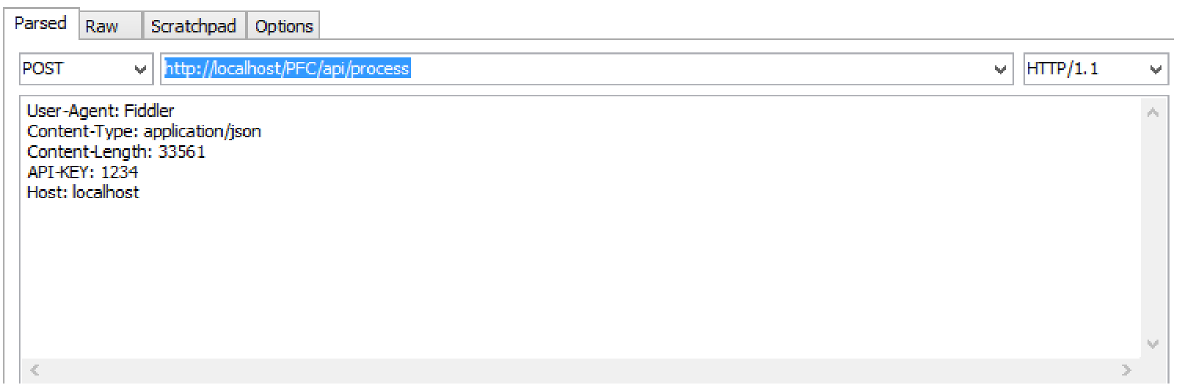

Figura 6.3. Creación de la petición HTTP POST (Cabeceras).

En la figura 6.4 se muestra el cuerpo de la petición HTTP cuyas cabeceras se describen en la figura 6.3. Se envía un objeto JSON con un campo, “imagen” y su valor, la imagen serializada en Base64.

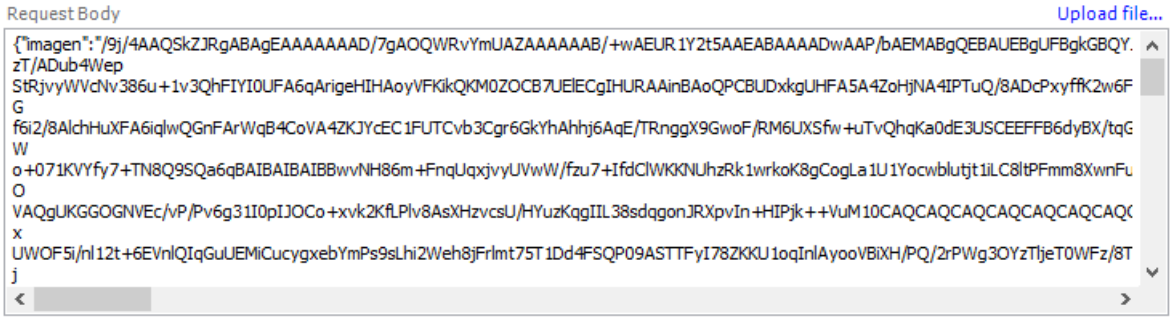

Figura 6.4. Cuerpo de la petición HTTP.

Resultado

La figura 6.5 muestra las cabeceras de la respuesta en formato JSON interpretado por Fiddler. Se trata de un OK, código 200 HTTP, con lo que la petición ha sido procesada por el servidor web.

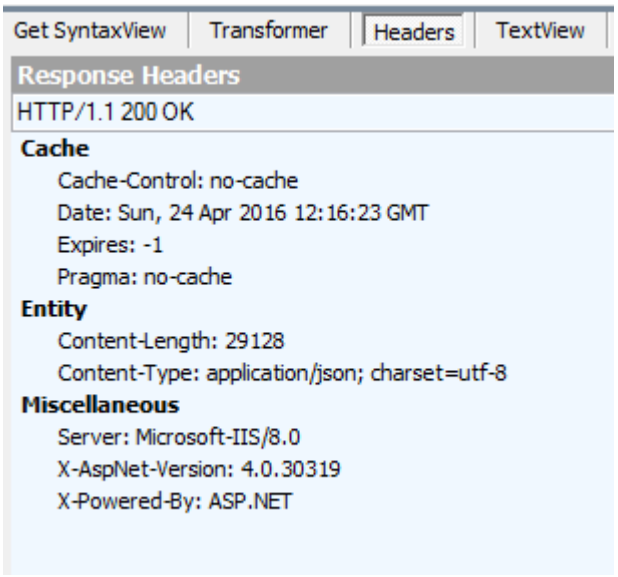

Figura 6.5. Cabeceras de la respuesta HTTP.

La figura 6.6 muestra el cuerpo de la misma respuesta del servidor, que viene en formato JSON y contiene la representación de la imagen en Base64 y un campo adicional, “Riesgo”, que viene a falso.

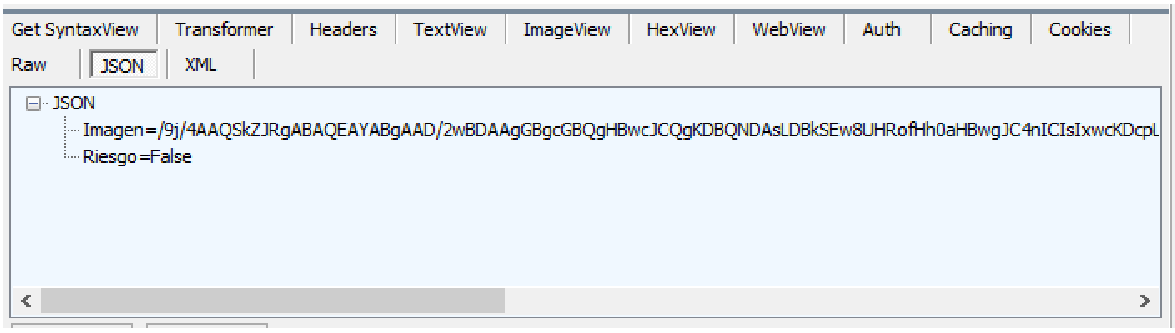

Figura 6.6. Cuerpo de la respuesta HTTP en JSON.

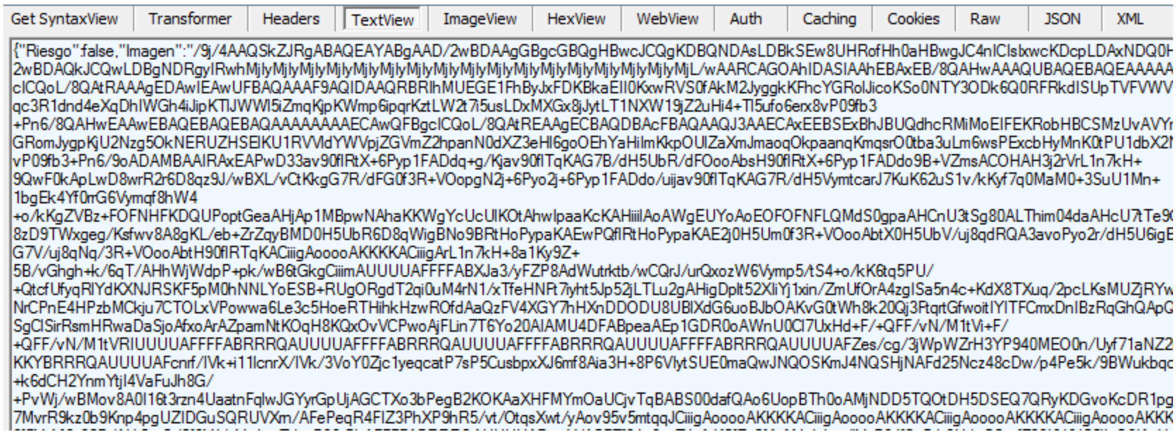

Figura 6.7. Respuesta HTTP en texto plano.

Finalmente, la figura 6.8 muestra el resultado de deserializar la imagen y guardarla en formato binario tras haber sido procesada por uno de los algoritmos de procesamiento de imagen enchufados a la aplicación principal como un plug-in. Se observa la marca de agua “Procesado: Green”, lo que garantiza que ha sido procesada en el servidor por uno de los plug-in.

Figura 6.8. Imagen devuelta en la respuesta del servidor.

El algoritmo de procesado fue elegido en base a lo configurado en el fichero XML que se encarga, mediante configuración y no en tiempo de compilación a través de modificaciones en el código, de seleccionar qué algoritmo debe procesar la imagen. La figura 6.9 muestra dicho fichero de configuración XML donde se pueden cambiar valores en tiempo de ejecución, por ejemplo, el valor del algoritmo que debe procesar las imágenes, sin tener que recompilar el código.

Web.config

```
<configSections>
  <!-- For more information on Entity Framework configura
  <section name="entityFramework" type="System.Data.Entit
</configSections>
<connectionStrings>
  <add name="DefaultConnection" providerName="System.Data
</connectionStrings>
<appSettings>
  <add key="webpages:Version" value="2.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="PreserveLoginUrl" value="true" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  <add key="Algoritmo" value="Green" />
  <add key="API-KEY" value="1234" />
  <add key="Role" value="GenericRole" />
  <add key="User" value="GenericUser" />
</appSettings>
<system.web>
```

Figura 6.9. Fichero XML de configuración.

El algoritmo de procesado Green se encuentra compilado y en forma de ensamblado en uno de los directorios de la aplicación de servidor. No pertenece al

proyecto de la aplicación de servidor ni se compiló con él, sino que fue añadido más tarde; exactamente como se espera de una extensión o plug-in.

En la figura 6.10 se muestra a la izquierda la vista del directorio del proyecto de la aplicación del servidor web con un directorio específico para los plug-in: Extensions. A la derecha se muestra el contenido del directorio Extension: la DLL ImageAnalizers, que contiene los algoritmos que se enganchan a la aplicación principal en tiempo de ejecución como plug-in.

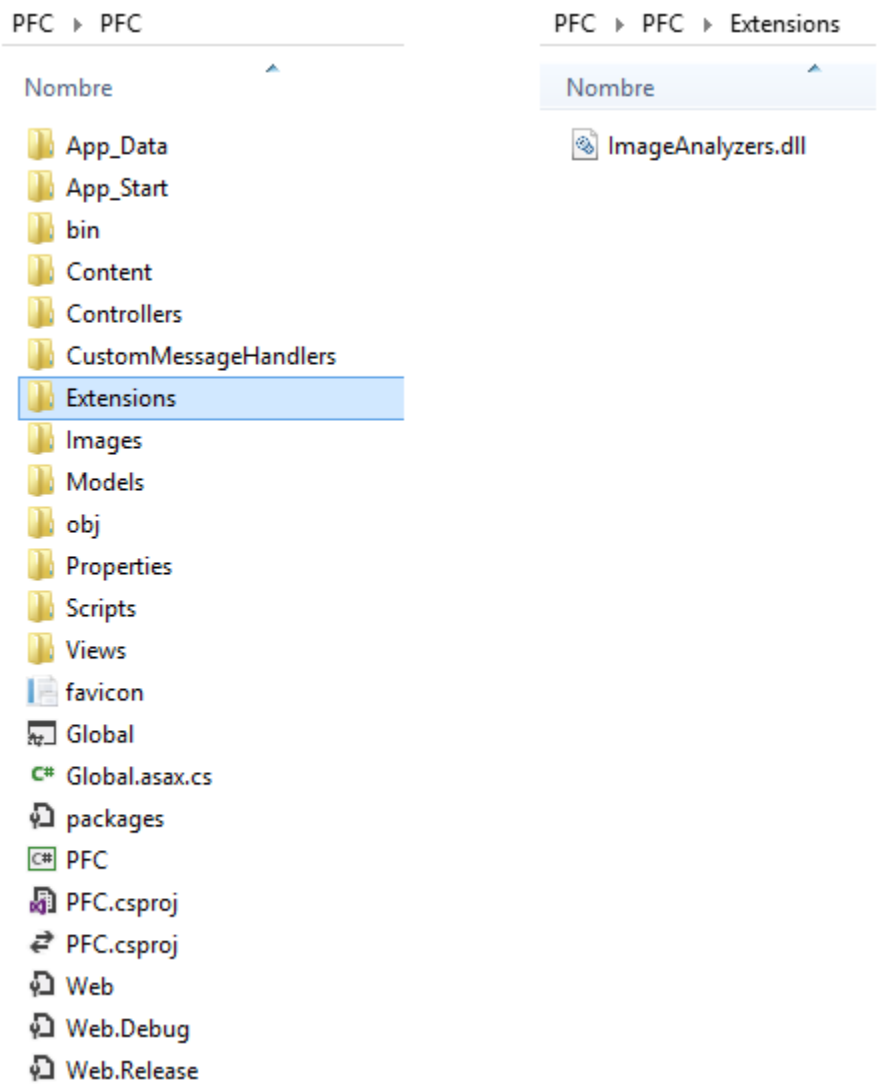

Figura 6.10. Directorios de la API y Extensions

Por tanto, con este experimento se ha comprobado que el comportamiento de la aplicación con la arquitectura y diseño propuesto ha sido el esperado. En particular los dos hitos que se pretendían conseguir se han validado, a saber: recibir una imagen desde un cliente a través de una API REST sobre HTTP estándar (interoperatividad) y procesarla con un algoritmo inyectado en tiempo de ejecución en la aplicación, siguiendo una arquitectura de plug-in para dotar de extensibilidad a la solución.

6.1.2 Experimento extensibilidad dinámica

Experimento

En este experimento se pretendió comprobar la capacidad de extensibilidad dinámica de la aplicación de servidor mediante el cambio en el fichero de configuración del algoritmo de procesado que debía tratar la imagen. Se eligió en este caso el algoritmo Blue, que debía poner una marca de agua azul sobre la imagen. No sería necesario cambiar nada más, ya que el fichero de ensamblado con los algoritmos se encontraba, igual que en el experimento anterior, en el fichero Extensiones de la aplicación.

En la figura 6.11 se muestra el fichero XML de configuración de la aplicación en el que se cambió la clave “Algoritmo” asignándole el valor “Blue” en tiempo de ejecución.

Web.config*

```
</connectionStrings>
<appSettings>
  <add key="webpages:Version" value="2.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="PreserveLoginUrl" value="true" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  <add key="Algoritmo" value="Blue" />
  <add key="API-KEY" value="1234" />
  <add key="Role" value="GenericRole" />
  <add key="User" value="GenericUser" />
</appSettings>
<system.web>
```

Figura 6.11. Fichero XML de configuración cambiado.

Resultado

El proceso de petición y respuesta se condujo de la misma manera que en el experimento anterior. En la figura 6.12 se muestra la imagen que vino en el cuerpo de la respuesta del servidor tras haber sido guardada en formato binario. Se observa que la marca de agua ha cambiado respecto al experimento anterior. Ahora muestra: "Procesado: Blue".

Figura 6.12. Imagen devuelta en la respuesta tras el cambio.

Se comprobó con este experimento que simplemente cambiando un valor en el fichero de configuración XML y sin necesidad de recompilar la aplicación de servidor, la solución ha sido capaz de cargar en tiempo de ejecución un plug-in que contiene un algoritmo de análisis de imagen diferente al de la ejecución anterior. Esta prueba de concepto garantiza la extensibilidad dinámica de aplicación pretendida por su diseño.

7 Líneas Futuras

Este proyecto pretende servir como base para futuras aplicaciones que se puedan beneficiar de una arquitectura cliente – servidor en la que se capturen imágenes mediante dispositivos móviles y se envíen al servidor para ser procesadas por algoritmos especializados, que pueden ser fruto de trabajos exclusivamente dedicados a ese problema. La dedicación exclusiva a la resolución del problema del procesamiento de imágenes sin tener que preocuparse por la aplicación que albergará estos algoritmos ni de su arquitectura para que se puedan poner en uso y sean funcionales dentro del contexto de una aplicación puede ser muy interesante.

Por otro lado, dado que el mayor mérito de este proyecto reside en la resolución de un problema mediante la utilización de abstracciones de diseño de software que permiten la puesta en marcha de una determinada arquitectura con las ventajas ya mencionadas, otra línea futura de trabajo podría consistir en ahondar en dichas abstracciones, patrones y conceptos de diseño de software que dotan a las aplicaciones de extensibilidad e interoperatividad, proponiéndose alternativas a esta implementación o transportándolas a otras plataformas (algo sencillo, dado que el valor está en el concepto y no en el propio código). Esta línea puede ser muy interesante ya que ambas temáticas, extensibilidad e interoperatividad, son dos aspectos absolutamente clave de las aplicaciones en el momento actual, y a buen seguro lo seguirán siendo en el futuro próximo.

8 Conclusiones

En este proyecto se ha realizado el diseño e implementación de una solución que conforma el Back End de un sistema cliente – servidor de mayor alcance que se completa con la aplicación móvil que actúa como cliente y que ya fue desarrollada en el proyecto referenciado en [3]. Los objetivos que se perseguían al inicio del proyecto, a saber: interoperatividad, extensibilidad dinámica y optimización de las comunicaciones, se han abordado mediante las siguientes decisiones de diseño de software y tecnologías utilizadas:

- **Interoperatividad:** se ha diseñado e implementado una interfaz para el Back End basada en una API de servicios web REST sobre HTTP y JSON. Esto ha dado como resultado una interoperatividad completa, ya que se utilizan sólo tecnologías y protocolos estándar que cualquier plataforma de desarrollo puede fácilmente manejar mediante librerías estándar. Para garantizar la interoperatividad se ha utilizado Fiddler en los experimentos y pruebas de concepto como cliente independiente y no ligado a una tecnología concreta. Fiddler hace uso de tecnologías, protocolos y formatos estándar en el desarrollo web actual. En este caso se han utilizado sus capacidades de trabajo con el protocolo estándar HTTP y de JSON como formato de intercambio también estándar.
- **Extensibilidad dinámica:** para dar respuesta a este requisito se ha realizado un diseño de software que arranca con el concepto de Inversion Of Control concretado en Dependency Injection e implementado finalmente bajo la forma de un patrón Plug-in. Para poder llevar a cabo esta implementación se ha hecho uso de Programación Orientada a Atributos y de las capacidades de composición aportadas por .NET a través de la tecnología que se generó en el marco del Managed Extensibility Framework y que ahora es nativa del framework de .NET. Por tanto, el uso de conceptos de ingeniería del software y de su aplicación mediante el uso de varias tecnologías concretas han proporcionado la extensibilidad dinámica deseada, ya que como se ha detallado en los experimentos descritos en la sección 6.1, el Back End es capaz de incorporar nuevos algoritmos de análisis de imagen sin necesidad de ser recompilado ni de ser parada su ejecución.

- **Optimización de las comunicaciones:** una vez decidido que se iba a implementar una API de servicios web para garantizar la interoperatividad del Back End con cualquier cliente, la decisión sobre la optimización de las comunicaciones en este contexto basculaba sobre qué tipo de servicios web utilizar y el formato de los datos que se intercambiaban. Tras el análisis de la opción SOAP para crear los servicios web y de XML como su formato de intercambio de datos, quedó claro que no eran las tecnologías que se buscaban, debido al gran overhead que introducen en las comunicaciones añadiendo un valor que no interesa en este proyecto en concreto. La alternativa de diseño de servicios web REST, con su peso absolutamente ajustado a las necesidades de comunicación entre aplicaciones a través de Internet, emparejada con el formato JSON, que nace específicamente para minimizar el tránsito de datos entre cliente y servidor, utilizando la mínima metainformación posible para trasladar los datos, han permitido crear una solución ajustada al requisito que se planteaba.

La ejecución de los experimentos han resultado en una ejecución satisfactoria de la aplicación de Back End desarrollada, en tanto en cuanto ha sido capaz de comunicarse con un cliente tecnológicamente agnóstico como Fiddler, garantizando así la interoperatividad mediante tecnologías estándar; ha podido extenderse también su funcionalidad sin recompilaciones y en tiempo de ejecución simplemente añadiendo nuevos algoritmos compilados en un directorio específico y cambiando un campo en el XML externo de configuración y también se ha comprobado cómo los datos intercambiados entre cliente y servidor han sido los mínimos necesarios, no existiendo prácticamente overhead producido por metadatos excesivos introducidos para el control o formato de las comunicaciones.

9 Glosario

Agnóstico: que desconoce lo divino y lo que trasciende a la experiencia. Agnóstico tecnológicamente se refiere a que no se preocupa ni conoce tecnologías concretas, sino principios genéricos que funcionan en cualquiera de ellas.

API: acrónimo que de Application Programming Interface. Interfaz que un sistema expone parar que otros interactúen e intercambien datos con él.

Atom: Atom Syndicated Format es un lenguaje basado en XML utilizado para los resúmenes que publican ciertos sitios web sobre su contenido.

Base64: es un formato de codificación de datos binarios en texto plano utilizando los caracteres ASCII. Se utilizan 64 caracteres ASCII para codificar cualquier mensaje.

Binding: en jerga informática es la acción de asociar o conectar dos elementos entre sí.

BSON: acrónimo que significa Binary JSON. Se trata de un formato binario para almacenar datos típicamente en una base de datos MongoDB.

Código objeto: es el lenguaje producido tras la compilación de otro lenguaje. Típicamente suele ser código máquina de algún procesador concreto, pero puede ser algún lenguaje intermedio.

Commodity: cualquier bien producido por el hombre o presente en la naturaleza del que existe abundancia, genera utilidad y tiene muy baja diferenciación respecto a otros, con lo que su precio tiende a ser relativamente bajo.

Compilación Just-In-Time: técnica de compilación de un programa en código máquina que efectúa la compilación en tiempo de ejecución del programa, no previamente.

Ensamblado: fichero con código ejecutable. En .NET representa la mínima unidad

Framework: Marco de desarrollo de software que proporciona típicamente unas librerías y utilidades que facilitan la construcción de aplicaciones.

Garbage Collection: conocido en español como recolección de basura, es el proceso mediante el cual las máquinas virtuales de algunas tecnologías, JAVA o .NET se

encargan de liberar la memoria ocupada por objetos que ya no son referenciados por un programa.

Git: es un software de control de versiones de código fuente muy utilizado y creado en 2005 para la gestión de versiones del kernel de Linux.

GitHub: es un servicio de hosting de repositorios de Git que permite el acceso distribuido a través de Internet de un equipo de desarrollo de software que trabaje sobre el mismo código fuente.

Hyperlink: enlace dentro de un Hypertexto.

Hypertext: texto con enlaces a otros textos. Típicamente utilizado en páginas web y generado con lenguaje HTML.

IDE: acrónimo de Integrated Development Environment. Se refiere a un entorno de desarrollo de software en el que se integran las herramientas y librerías necesarias para editar compilar y depurar un programa de software.

Interoperatividad: capacidad de un sistema para intercambiar datos con otros sistemas de diseño y/o tecnologías diferentes.

JSON: es un acrónimo que significa JavaScript Object Notation. Se trata de un formato estándar abierto que utiliza texto plano para transmitir objetos en forma de pares atributo-valor. Es un formato muy simple, fácil de leer por un humano y fácil de procesar por lenguajes muy asociados a Internet como Javascript.

Lazy Loading: técnica de desarrollo de software mediante la cual, para ahorrar recursos y optimizar la ejecución de un programa, no se carga en memoria una estructura de datos en el momento de la creación del objeto que la referencia, sino que se espera hasta su primera utilización real en tiempo de ejecución para hacer la carga e incurrir en ese coste.

Lenguaje fuertemente tipado: característica de algunos lenguajes de programación que no permite que dado el valor de una variable concreta no se permite que se utilice como si fuera de otro tipo sin antes convertirse a ese otro tipo (typecast).
ejecutable. Su equivalente en Java es el fichero JAR.

Manejador: en la jerga de desarrollo de software se refiere a un fragmento de código que entra en juego cuando se produce un determinado evento, encargándose de la gestión del mismo.

Máquina Virtual: simulación de una máquina hardware mediante software.

Memory Leak: las fugas de memoria son errores que se producen en un programa cuando la memoria no se gestiona correctamente. Puede ocasionar que se agote la memoria de la máquina debido a que no se eliminen objetos que ya no se utilizan, etc.

Metadato: dato que provee información sobre otro dato al que típicamente está asociado de alguna manera.

Metainformación: información sobre otra información a la que está asociada de alguna manera.

mHealth: abreviación de mobile health, un término que se utiliza para designar a la medicina que se apoya en dispositivos móviles.

Mock: Objeto simple que se utiliza para simular una dependencia real de otro objeto que se quiere probar con tests unitarios. Elimina la complejidad de tener que programar la dependencia real para realizar el test.

Overhaed: sobrecarga para el rendimiento óptimo que produce alguna técnica de software o formato de datos.

Patrón de Diseño: solución de software de demostrada efectividad y reutilizable para un tipo de problema común que se repite en el desarrollo de programas.

Parser: un parser es el analizador sintáctico de un compilador y transforma su entrada otra estructura de datos (típicamente un árbol) en la que se representa la jerarquía de los elementos de la entrada y que sirve para el análisis de la misma.

Payload: carga o cuerpo de una petición HTTP.

Pipeline: conjunto de etapas sucesivas que componen un sistema.

Programación Orientada a Objetos: Paradigma de desarrollo de software en el que se descompone un problema de un determinado dominio en objetos del mismo.

Protocol Buffers: es un formato para serializar datos estructurados diseñado por Google y puesto a disposición del público en general mediante generadores de este formato para diferentes lenguajes y plataformas.
Renderizar: término utilizado típicamente en computación gráfica para referirse al hecho de hacer los cálculos necesarios para mostrar gráficamente una imagen o vídeo.

REST: Acrónico de REpresentational State Transfer. Estilo arquitectural de desarrollo de software que sigue una API de servicios web que funcione sobre HTTP y los verbs estándar de este para operar con los datos del lado del servidor. El intercambio de datos se produce típicamente en formato JSON o XML.

RPC: acrónimo de Remote Procedure Call. En desarrollo de software distribuido es la acción de ejecutar un procedimiento, método o función en una máquina remota y no en la que realiza la llamada al procedimiento.

SOLID: Acrónimo de Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation y Dependency Inversion. Estos cinco principios son aceptados de facto como distintivos del buen diseño de un programa orientado a objetos.

Stub: método o funcionalidad simple que se utiliza para simular la respuesta de una más compleja y real con el objetivo de probar el fragmento de código en el que se incluye la misma mediante tests unitarios. Elimina la complejidad de tener que programar la funcionalidad real para realizar el test.

Suite de Protocolos de Internet: Familia de protocolos en los que se basa Internet y que permiten la transmisión de datos entre ordenadores.

Test Unitario: prueba que se realiza para comprobar el correcto funcionamiento de un elemento concreto de un programa de software y que se caracteriza por ser lo más simple posible, en el sentido en el que es indivisible en pruebas más pequeñas, lo que garantiza que el objeto de la prueba queda validado si se supera el test. Típicamente se generan baterías completas de tests unitarios para probar todo un programa.

Typecast: conversión de una variable de un tipo a otro.

URI: acrónimo de Uniform Resource Identifier. Se trata de una cadena de caracteres que identifican de forma unívoca a un recurso en una red.

Verb: término utilizado en la jerga del desarrollo web para designar a los métodos definidos por HTTP para realizar una determinada acción sobre un recurso identificado por una URI.

VMWare: Plataforma de virtualización de servidores y máquinas hardware en general. Simula por software la ejecución de una máquina física completo, incluyendo su hardware y software.

WSDL: acrónimo de Web Service Description Language. Es un lenguaje basado en XML que se utiliza para definir la funcionalidad de un servicio web SOAP.

WS-*: acrónimo que se utiliza para designar las especificaciones de los servicios web SOAP (Web Service Specifications: WS-*). Estas especificaciones son muy variadas y cada una se encuentra en un recorrido de madurez diferente, no existiendo un conjunto oficial que defina al cuerpo completo de dichas especificaciones ni un organismo o cuerpo que aglutine su definición y autoría.

10 Referencias

[1] Consumo móvil en España
http://www2.deloitte.com/es/es/pages/technology-media-and-telecommunications/articles/Consumo-movil-2015.html

[2] Front End y Back End
https://en.wikipedia.org/wiki/Front_and_back_ends

[3] Sistema de Seguimiento de la Evolución de Lunares para Prevención de Melanomas Basado en Smartphones
A. Pacheco y M. Jarysz, Proyecto Fin de Carrera, Escuela Técnica Superior de Ingeniería Informática, Universidad de la Laguna, 2011

[4] Hollywood Principle
http://martinfowler.com/bliki/InversionOfControl.html

[5] Inversion of Control
http://martinfowler.com/bliki/InversionOfControl.html

[6] Open/Closed principle
https://blog.8thlight.com/uncle-bob/2014/05/12/TheOpenClosedPrinciple.html

[7] SOLID
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[8] Service Locator
http://www.martinfowler.com/articles/injection.html

[9] Java Preprocessor
http://www.cs.ait.ac.th/~on/O/oreilly/java/javanut/ch02_04.htm

[10] C# Preprocessor
https://msdn.microsoft.com/en-us/library/ed8yd1ha.aspx

[11] HTTP/1.1 Specification, IETF RFC 2616
https://www.ietf.org/rfc/rfc2616.txt

[12] SOAP vs REST Performance
http://www.ateam-oracle.com/performance-study-rest-vs-soap-for-mobile-applications/

[13] C y C++ NUL Terminated Strings
http://queue.acm.org/detail.cfm?id=2010365

[14] BSON
http://bsonspec.org/

[15] Protocol Buffers
https://developers.google.com/protocol-buffers/

[16] Base64
https://blogs.oracle.com/rammenon/entry/base64_explained

[17] Base85
https://en.wikipedia.org/wiki/Ascii85

[18] Base91
http://base91.sourceforge.net/

[19] Base94
https://gist.github.com/iso2022jp/4054241

[20] MobiHealth
A. Van Halteren, R. Bults, K. WAC, N. Dokovsky, G. Koprinkov, I. Widya, D. Konstantas, V. Jones, *"Wireless Body Area Networks for Healthcare : the MobiHealth Project"*, Wearable eHealth Systems for Personalised Health Management, Studies in Health Technology and Informatics, Vol. 108, 2004, pp. 121-126.

[21] Wireless Remote Healthcare Monitoring with Motes
E. Lubrin, E. Lawrence, K. Navarro, "Wireless Remote Healthcare Monitoring with Motes", Proceedings of the International Conference of Mobile Business (ICMB'05), IEEE.

[22] Distributed Image Application Processing Considering CORBA and XML Technology
S. Moldovan, M. Vaida, G. Toderean, "Distributed Image processing Application Considering CORBA and XML Technology", Acta Tecnica Napocencis – Electronics and telecommunications, Vol. 45, Nº 1, 2004.

[23] iHealth
http://www.ihealthlabs.com

[24] Mobile Assay
http://www.mobileassay.com

[25] Mobile Assay: test de tira reactiva
https://www.youtube.com/watch?v=dWy6whttgNE

[26] Fiddler
http://www.telerik.com/fiddler

[27] .NET
https://www.microsoft.com/net/default.aspx

[28] Compilación Just-In-Time
http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

[29] Managed Code
https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664(v=vs.85).aspx

[30] CIL
https://en.wikipedia.org/wiki/Common_Intermediate_Language

[31] Lenguajes de programación en la plataforma .NET
https://en.wikipedia.org/wiki/List_of_CLI_languages

[32] .NET Framework Class Library
https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx

[33] .NET Base Class Library
https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx

[34] GitHub
https://github.com/

[35] .NET Core
https://dotnet.github.io/

[36] Especificación de C#
https://www.microsoft.com/en-us/download/details.aspx?id=7029

[37] ASP.NET Web API
http://www.asp.net/web-api

[38] Xamarin
https://www.xamarin.com/

[39] HTTP Message Handlers in ASP.NET Web API
http://www.asp.net/web-api/overview/advanced/http-message-handlers

11 Bibliografía

[Davis 2008]
I. Davis, "*What Are The Benefits of MVC?*". Documento en formato HTML accesible por Internet en la dirección: http://blog.iandavis.com/2008/12/what-are-the-benefits-of-mvc/.

[Fowler, Beck, Brant, Opdyke y Roberts 1999]
M. Fowler, K. Beck, J. Brant, W. Opdyke y D. Roberts *Refactoring: Improving the Design of Existing Code*. O'Reilly, 1999.

[Fowler 2004]
M. Fowler, "*Inversion of Control Container and the Dependency Injection Pattern*". Documento en formato HTML accesible por Internet en la dirección: http://www.martinfowler.com/articles/injection.html.

[Freeman, Robson, Bates y Sierra 2004]
E. Freeman, E. Robson, B. Bates y K. Sierra, *Head First Design Patterns*. O'Reilly, 2004.

[Galloway, Wilson, Allen y Matson 2014]
j. Galloway, B. Wilson, S. Allen y D. Matson, *Professional ASP.NET MVC 5*. Wrox, 2014.

[Gamma, Helm, Johnson y Vlissides 1994]
E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Johnson y Foote 1988]
R. E. Johnson y B. Foote, *"Designing Reusable Classes"*, Journal of Object Oriented Programming, Vol. 1, Nº 2, 1988, pp. 22-35.

[Kamp 2011]
P. Kamp, "*The Most Expensive One-byte Mistake. Did Ken, Dennis, and Brian choose wrong with NUL-terminated text strings?*". Acmqueue, Association for Computing Machinery, Vol. 9, Nº 7, 2011. Documento en formato HTML accesible también por Internet en la dirección: http://queue.acm.org/detail.cfm?id=2010365.

[Larman 2004]
C. Larman, *Applying UML and Patterns*. Prentice Hall, 2004.

[Lhotka 2006]
R. Lhotka, *Expert C# Business Objects*. Apress, 2006.

[Martin 2000]
R. C. Martin, *More C++ Gems*. Cambridge University Press, 2000.

[Martin 2002]
R. C. Martin, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[Martin 2012]
R. Martin, *The principles of OOD*. Documento en formato HTML accesible por Internet en la dirección:
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[McLaughlin, Pollice y West 2007]
B. D. McLaughlin, G. Pollice y D. West, *Head First Object Oriented Analysis and Design*. O'Reilly, 2007.

[Meyer, Hill, Homer, Taylor, Bansode, Wall, Boucher y Bogawat 2009]
J. Meyer, D. Hill, A. Homer, J. Taylor, P. Bansode, L. Wall, R. Boucher y A. Bogawat, "*Microsoft Applicatio Architecture Guide, 2nd Edition*". Documento en formato HTML accesible por Internet en la dirección:
https://msdn.microsoft.com/en-us/library/ff650706.aspx.

[Robinson, Nagel, Glynn, Skinner, Watson y Evjen 2004]
S. Robinson, C. Nagel, J. Glynn, M. Skinner, K. Watson y B. Evjen, *Pro C#*. Wrox, 2004.

[Sahni 2013]
V. Sahni, "*Best Practices for Designing a Pragmatic RESTFul API*". Documento en formato HTML accesible por Internet en la dirección:
http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api.

[Troelsen 2012]
A. Troelsen, *Pro C# 5.0 and the .NET 4.5 Framework*. Apress, 2012.

[Ugurlu, Zeitler y Kheyrollahi 2013]
T. Ugurlu, A. Zeitler y A. Kheyrollahi, *Pro ASP.NET Web API. HTTP Web Services in ASP.NET*. Apress, 2013.