UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

La Universidad Católica de Loja

ÁREA TÉCNICA

TÍTULO DE INGENIERO EN SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Consideraciones de arquitectura de software a nivel de diseño arquitectónico y desarrollo de software para minimizar vulnerabilidades en aplicaciones web basados en OWASP Top Ten 2013, caso de estudio Arquitecturas: REST.

TRABAJO DE TITULACIÓN

AUTOR: Correa Tenesaca, Roddy Andrés

DIRECTOR: Guamán Coronel, Daniel Alejandro, Mgs.

LOJA – ECUADOR

2015

APROBACIÓN DEL DIRECTOR DEL TRABAJO DE FIN DE TITULACIÓN

Magister**.**

Daniel Alejandro Guamán Coronel

DIRECTOR DEL PROYECTO DE FIN DE TITULACIÓN

De mi consideración:

El presente trabajo de titulación: “Consideraciones de arquitectura de software a nivel de diseño arquitectónico y desarrollo de software para minimizar vulnerabilidades en aplicaciones web basados en OWASP Top Ten 2013, caso de estudio Arquitecturas REST”, realizado por Roddy Andrés Correa Tenesaca, ha sido orientado y revisado durante su ejecución, por cuanto se aprueba la presentación del mismo.

Loja, septiembre de 2015

f). ……

Mgs. Daniel Alejandro Guamán Coronel

CI. 1103777403

DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS

"Yo, Roddy Andrés Correa Tenesaca, declaro ser autor (a) del presente trabajo de titulación: "Consideraciones de Arquitectura de Software a nivel de Diseño Arquitectónico y Desarrollo de Software para minimizar vulnerabilidades en aplicaciones web basados en OWASP Top Ten 2013", de la Titulación de Sistemas Informáticos y Computación, siendo el Ing. Daniel Alejandro Guamán director (a) del presente trabajo; y eximo expresamente a la Universidad Técnica Particular de Loja y a sus representantes legales de posibles reclamos o acciones legales. Además certifico que las ideas, conceptos, procedimientos y resultados vertidos en el presente trabajo investigativo, son de mi exclusiva responsabilidad.

Adicionalmente declaro conocer y aceptar la disposición del Art. 88 del Estatuto Orgánico de la Universidad Técnica Particular de Loja que en su parte pertinente textualmente dice: "Forman parte del patrimonio de la Universidad la propiedad intelectual de investigaciones, trabajos científicos o técnicos y tesis de grado o trabajos de titulación que se realicen con el apoyo financiero, académico o institucional (operativo) de la Universidad"

f)……...……..

Autor: Roddy Andrés Correa Tenesaca

Cédula. 1104542749

DEDICATORIA

Dedico este trabajo a mis padres Rody y Gloria por haberme brindado su apoyo incondicional en todo momento, gracias a ellos he podido culminar mis estudios profesionales.

A mi familia por su preocupación y muestras de cariño.

A mi esposa Viviana por su amor y comprensión durante toda esta etapa.

A mis compañeros de estudio y amigos que han sido parte de este constante aprendizaje.

A mi hija Annie de manera especial por ser motivo de inspiración y motivación para culminar una meta más en mi vida profesional.

AGRADECIMIENTO

A Dios por la vida y sus bendiciones derramadas sobre mí.

A mi esposa Viviana por estar con migo en todo momento y dificultad.

A mis padres por el apoyo y sus consejos que me han permitido crecer espiritual y humanamente.

A la Universidad Técnica Particular de Loja por su acogida y constante capacitación

A mi director de tesis por su paciencia y constancia durante todo este tiempo de desarrollo de tesis.

A todos ellos expreso mi más sincero agradecimiento y estima.

Roddy Andrés Correa T.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

El presente trabajo de fin de titulación presenta un estudio orientado en el aseguramiento y minimización de vulnerabilidades de aplicaciones web. Se realiza este estudio en consecuencia del creciente descubrimiento y ejecución de procedimientos capaces de violar la seguridad de los aplicativos web; es por esto qué, este estudio se basa en OWASP Top Ten 2013 donde se determinan técnicas y recomendaciones de buenas prácticas de aseguramiento de aplicaciones web con la finalidad de mitigar, reducir y enfrentar debilidades de seguridad que comprometan la disponibilidad, integridad y garantía de datos y aplicaciones web.

Para justificar éste estudio se presenta el diseño, implementación y validación de un prototipo de aplicaciones web basadas en el estilo arquitectónico REST, donde se busca exponer información y al mismo tiempo garantizar la seguridad del aplicativo. El aseguramiento como factor primordial de éste análisis se enfoca en la consolidación de la seguridad de software desde la definición arquitectónica, escritura del código fuente y gestión y configuración del servidor de aplicaciones web.

PALABRAS CLAVE: Aplicaciones web, OWASP, vulnerabilidad, estilo arquitectónico, REST, código fuente.

ABSTRACT

This work presents a degree so oriented insurance and minimizing vulnerabilities of web applications studio. This study was done in consequence of the growing discovery and implementation of procedures able to violate the security of web applications; It is why this study is based on OWASP Top Ten 2013 where techniques and best practice recommendations assurance web applications in order to mitigate, reduce and address security vulnerabilities that compromise the availability, integrity and security of data is determined and web applications.

To justify this study design, implementation and validation of a prototype web applications based on the REST architectural style, which seeks to expose information while ensuring the safety of the application is presented. The primary factor assurance as this analysis focuses on the consolidation of security software from architectural definition, writing source code and configuration management and Web application server.

KEYWORDS: Web application, OWASP, vulnerability, architectural style, REST, source code.

INTRODUCCIÓN

En la actualidad, la exigencia de los usuarios de poder acceder desde cualquier lugar a través de internet y realizar tareas convencionales que minimicen el uso de recursos esenciales como el tiempo, han hecho que las organizaciones se vean atentas a satisfacer sus requerimientos mediante aplicaciones web, convirtiéndose estas en un mecanismo esencial en la operatividad de la organización y la gestión de la información. La información es un bien clave a nivel organizacional, representa un alto grado de criticidad, ya que involucra datos de activos organizacionales, personales y funcionales; éste último implica una estructuración de procesos con los cuales se automatiza tareas operativas de la organización.

El desarrollo de software compromete una serie de actividades para obtener una producción de funcionalidades satisfactorias; cuando no se ha realizado un correcto análisis varias son las formas de infringir dichas funcionalidades, por esta razón, las aplicaciones web presentan ventajas y desventajas a nivel organizacional. La facilidad de acceso comúnmente ocasiona resultados contraproducentes si no se ha tomado los correctivos necesarios; por ejemplo: si cualquier usuario hace uso de un aplicativo que es fácilmente vulnerable a ataques, la seguridad de la información se verá comprometida quedando totalmente expuesta. Es por esto qué OWASP presenta un Top Ten de vulnerabilidades que afectan a aplicaciones web, el mismo que sirve de guía para enfrentar y prevenir las debilidades a las que se encuentran expuestas las aplicaciones hoy en día.

Este proyecto determina y considera un compendio de técnicas y recomendaciones que garanticen la seguridad en aplicaciones web; para lo cual, se toma como referencia las especificaciones de OWASP, con la finalidad de utilizar estándares de seguridad especializados en el estilo arquitectónico REST.

Como parte del desarrollo de este proyecto se realiza la construcción de servicios web RESTful, en donde se verán aplicados los conceptos de estudio y las especificaciones de OWASP para garantizar la disponibilidad e integridad del aplicativo REST.

El capítulo uno es un análisis de conceptos básicos de arquitectura de software, REST, seguridad en aplicaciones y recomendaciones de OWASP para abarcar el ámbito de la

seguridad del software, con la finalidad de poseer una visión general de las conceptualizaciones necesarias que este estudio requiere.

El capítulo dos presenta el diseño del prototipo en el que se detalla la arquitectura de la aplicación REST, modelo de datos, consideraciones de seguridad necesarias para minimizar las vulnerabilidades enfrentadas, diseño de servicios web y recursos tecnológicos que son parte de la implementación.

El capítulo tres contempla el desarrollo de software, en donde se observa la implementación de medidas de seguridad determinadas del análisis en el capítulo dos, para comprobar que un correcto desarrollo es capaz de minimizar vulnerabilidades y garantizar la seguridad de la información cuando se realiza aplicaciones basadas REST. El proceso de desarrollo involucra tres niveles de ejecución: datos, código fuente y servidor de aplicaciones. Datos para realizar la implementación física del modelo de datos aplicando técnicas de procedimientos almacenados, código fuente; para la construcción del modelo de la aplicación en donde se realiza el uso de consultas parametrizadas, método de limpieza de caracteres especiales, entre otros, y, a nivel de servidor se realizan configuraciones de seguridad como autenticación, filtros de seguridad ClickJacking y configuración de cabeceras de respuesta HTTP.

El capítulo cuatro contiene las pruebas y resultados obtenidos durante la validación de calidad y seguridad del aplicativo REST implementado. La validación se realizó con la ayuda de herramientas especializadas, en la que se determinó el correcto desarrollo del aplicativo REST frente a la arquitectura diseñada, calidad del código fuente, y la seguridad de los servicios web RESTful.

GLOSARIO

Arquitectura de Software: diseño a alto nivel de la estructura de un sistema de software, usualmente es el paso previo a la codificación de software

REST: (Transferencia de Estado Representacional), es un estilo arquitectónico de software para sistemas basados en la Web.

Software: es la parte intangible de una máquina computacional; es un conjunto de procedimientos y datos que aportan al cumplimiento de funcionalidades lógicas de un sistema computacional.

Caché: representa un tipo de almacenamiento donde se conservan datos temporales con el fin de ahorrar tiempo de cálculo en el despliegue de aplicaciones o actividades basadas en la red.

Atributos de Calidad: es un compuesto de métricas que permiten evaluar la calidad de los sistemas de software.

Seguridad: se refiere a la ausencia de riesgos; desde el punto de vista de software es la confianza que refleja a los usuarios de que la información y datos son confidenciales.

Vulnerabilidad: se comprende como una debilidad de software que permite a usuarios entendidos determinar y violar las seguridades de software.

Framework: (Marco de Trabajo), es una miscelánea de conceptos y normativas para enfrentar algún tipo de problema particular; usualmente se aplica para dar soluciones ágiles de software.

RESTful: corresponde a la denominación de aplicaciones de software que adoptan el estilo arquitectónico REST.

Datos: son una representación simbólica de alguna condición numérica, alfabética, etc., de una variable u atributo.

Servidor de Aplicaciones: trata de un grupo de servicios de aplicación donde se aloja y se despliega una o diversas aplicaciones de software.

Base de datos: es un repositorio de información donde se reúne información categorizada y ordenada pertenecientes a un mismo contexto.

Codificación: es el proceso de escribir, depurar, y corregir código fuente de una aplicación de software.

Iteración: significa repetir un procedimiento con el objetivo de alcanzar una meta específica.

Alertas de seguridad: simboliza una advertencia a la que se le debe prestar atención; puede referirse a una advertencia de rompimiento de seguridades de software.

WWW: World Wide Web o red informática mundial, es un sistema de documentos o hipermedias acoplados entre sí, que son accesibles a través de internet.

HTTP: Protocolo de transferencia de hipertexto, es un protocolo aplicado para realizar transacciones de transporte dentro de la WWW.

SQL: Lenguaje de consulta estructurado es un método de acceso a base de datos que permite ejecutar operaciones de consulta, inserción, actualización y eliminación de datos.

API: Interfaz de programación de aplicaciones, es un conjunto de reglas, métodos, funciones y más que funciona manera de biblioteca para ser utilizado por un software.

JAVA: JAVA es un lenguaje de programación orientado a objetos.

IEEE: Instituto de Ingeniería Eléctrica y Electrónica.

App: Aplicación

CAPÍTULO I

ESTADO DEL ARTE.

1.1. Arquitectura de Software.

Para (Maier, Emery, & Hilliard, 2001) y (L., Clements, & Kazman, 2002), la arquitectura de software es una organización de un sistema de software que contempla relaciones entre componentes, ambientes y principios rectores de su diseño y evolución. Además funciona como medio temprano para documentar soluciones de diseño de alto nivel con el objeto de proporcionar una visión total y estructural de software.

Se exige a partir del crecimiento de requerimientos, funcionalidades y capacidad de procesamiento, lo cual demanda a los arquitectos de software a elaborar diseños arquitectónicos de software capaces de solventar las necesidades de los clientes.

Como se ha explicado hasta aquí, la arquitectura de software representa una abstracción de software que se emplea como metodología temprana en el desarrollo de software. De este modo existen tres razones principales por lo cual la arquitectura de software desempeña un papel esencial durante el ciclo de vida de desarrollo de software, las mismas que se detallan a continuación:

a) Es un medio de comunicación.
b) Es una toma de decisiones tempranas.
c) Es reusable y transferible.

1.1.1. Estilos Arquitectónicos.

Según (Rivera Posso, 2010), los estilos arquitectónicos determinan conjunto de reglas de diseño que identifican clases de componentes y conectores que se pueden utilizar para constituir un sistema o subsistema de software junto con restricciones locales y globales de su composición. (Adriana & Vanina, 2007) y (Ramos & Depetris, 2012) explican que un estilo arquitectónico no es una arquitectura de software, sino una metodología que forma parte de una resolución de problemas concreta.

Existen varios tipos de estilos arquitectónicos, los mismos que se diferencian por su objetivo y características de implementación. Para este estudio se seleccionaron cinco estilos arquitectónicos expuestos en el siguiente listado:

a) Centrado en datos.
b) Flujo de datos.
c) Llamada y retorno.
d) Orientado a objetos.
e) Orientado a servicios.

El estudio de los estilos arquitectónicos mencionados anteriormente se limitará a detalles breves, en los que se estudiará definición, características, ventajas y desventajas. En la tabla 1 se presenta la definición y principales características de cada estilo arquitectónico.

Tabla 1: Estilos arquitectónicos - Definición y características.

Estilo arquitectónico	Definición	Características
Centrado en datos	Funciona como un gran almacén de datos, donde distintas aplicaciones, independientemente de la plataforma pueden conectarse y ejecutar procedimientos a nivel de base de datos.	▪ Facilita la modificabilidad de software. ▪ Los clientes ejecutan los procesos de manera independiente. ▪ Simplifica la transferencia de información entre clientes.
Flujo de datos	Se basa en el patrón arquitectónico tuberías y filtros, comprende una funcionalidad a través de componentes (filtros) conectados mediante tuberías, para transmitir información de un componente a otro.	▪ Permite procesamiento por lotes de datos. ▪ Está diseñado para recibir datos y producir una salida. ▪ La información es canalizada por tuberías.
Llamada y retorno	Este estilo arquitectónico pretende descomponer el sistema de software en objetos. Su funcionamiento comprende peticiones y respuestas entre objetos.	▪ Otorga facilidad de mantenimiento de software. ▪ Los componentes se organizan jerárquicamente. ▪ Tiene dos subestilos: Programa/subprograma y Llamadas de procedimiento remoto.

Orientado a Objetos	Es una técnica en la que los componentes encapsulan datos y operaciones; se comunican por medio de Objetos. (Castro, 2004), detalla que se basa en tres principios básicos: Objetos, Encapsulamiento / Ocultación, Herencia y Polimorfismo.	▪ Los componentes se comunican a través de mensajes. ▪ El desarrollo depende de la creación de objetos. ▪ Las interfaces están separadas de las implementaciones. ▪ Los objetos interactúan mediante invocaciones a las operaciones.
Orientado a Servicios	Es una organización de un sistema de software descrita en forma de servicios. Los componentes de software basados en este estilo, aplican conceptos relacionados con protocolos y estándares que permiten el intercambio de información entre aplicaciones de software que funcionen a través de la red.	▪ Dispone de funcionalidades reusables. ▪ Aporta a la flexibilidad de software. ▪ Integra sistemas de software. ▪ Establece un medio de comunicación entre varias aplicaciones de software.

Elaboración. El Autor.

Seguidamente en la tabla 2 se detallan las ventajas y desventajas de cada estilo arquitectónico para sintetizar soluciones estructurales a problemas específicos de software.

Tabla 2: Estilos Arquitectónicos - Ventajas y Desventajas.

Estilo arquitectónico	Ventajas	Desventajas
Centrado en datos	▪ Posibilita la integración entre clientes y sistemas de software. ▪ Resuelve problemas no deterministas. ▪ Permite entender el estado en cada proceso.	▪ Posee una estructura de datos idéntica para todos los clientes. ▪ Ofrece problemas de carga en caso de concurrencia de clientes. ▪ Falta de integridad de datos, por concurrencia de sistemas de software.
Flujo de datos	▪ Fácil comprensión de combinación entre componentes. ▪ Es reutilizable. ▪ Los filtros son independientes. ▪ Facilidad de mantenimiento.	▪ Presenta problemas de rendimiento. ▪ En caso de existir problemas entre componentes, podría ocurrir una pérdida de información. ▪ Falta de integridad de datos.
Orientado a Objetos	▪ Reutilización de recursos. ▪ Encapsulamiento de datos. ▪ Dinamismo en el manejo de datos.	▪ Incompatibilidad con lenguajes de programación. ▪ Es necesario conocer la identidad del componente para invocar un objeto.
Orientado a Servicios	▪ Respuesta rápida ante las necesidades del cliente. ▪ Permite la interoperabilidad entre sistemas. ▪ Aporta a la escalabilidad de un sistema. ▪ Ofrece una fácil mantenibilidad.	▪ Los desarrolladores necesitan conocer los procesos del negocio. ▪ Al estar expuesto a la red, se encuentra expuesto a ataques si no se toman los correctivos de seguridad necesarios.

Elaboración. El Autor.

1.1.2. Patrones Arquitectónicos.

Para (Almeira & Perez Cavenago, 2007), los patrones arquitectónicos: “son una disciplina de resolución de problemas en ingeniería del software que ha surgido con mayor énfasis en la comunidad de orientación a objetos, aunque pueden ser aplicados en cualquier ámbito de la informática”. De forma concreta (Rivera Posso, 2010) define a los patrones arquitectónicos como plantillas específicas que aportan una forma de resolución de problemas dentro de arquitecturas de software.

En consecuencia, a las teorías expuestas los patrones arquitectónicos detallan una solución puntual a inconsistencias de software; es importante especificar que presentan menor alcance que los estilos arquitectónicos. En similitud a los estilos arquitectónicos se presenta una clasificación en base a características y entorno de ejecución de los tres principales patrones arquitectónicos, estos son:

a) Capas.
b) Pizarra.
c) Modelo Vista y Controlado (MVC).

Los patrones arquitectónicos en cuestión se detallan en la tabla 3, en función de su definición y características de cada uno de ellos.

Tabla 3: Patrones Arquitectónicos - Definición y Características.

Patrón arquitectónico	Definición	Características
Capas	Según (Llorente, Zorrilla Castro, Ramos Barroso, & Calvarro, 2010), “son agrupaciones horizontales lógicas de componentes de software que componen a la aplicación de software o servicio”. Las Capas efectúan una interacción entre sí para cumplir funcionalidades objetivas de software.	▪ Maximiza la reutilización y mantenibilidad. ▪ Las capas son independientes y pueden estar en el mismo nodo o distribuirse sobre varios nodos. ▪ Cada capa es reutilizable. ▪ Su programación es similar a la programación modular.

Pizarra	Este sistema de arquitectura es utilizado frecuentemente en sistemas expertos, multiagentes o basados en conocimiento, la arquitectura en pizarra consta de componentes y elementos que usualmente se aplican en el campo de Inteligencia Artificial.	▪ Permite integrar distintas representaciones del conocimiento. ▪ Tiene un funcionamiento incremental; se basa en el incremento del conocimiento. ▪ Afronta razonamientos complejos.
Modelo Vista Controlador (MVC)	Gestiona una división de componentes en base a su funcionamiento, divide un sistema de software en tres partes: modelo, vista, y controlador. El modelo controla los datos del programa, la vista es la interfaz gráfica que se muestra al usuario, y el controlador gestiona funciones y procedimientos sobre los datos del modelo.	▪ Desacopla las vistas de su modelo y lógica. ▪ Cada componente del patrón está altamente especializado en su función. ▪ Favorece a la modificabilidad de software, debido a la independencia de cada componente de desarrollo.

Elaboración. El Autor.

Como se aprecia, cada patrón arquitectónico resuelve una necesidad específica y se maneja en un ambiente diferente uno de otro. Es importante destacar que éstos se diferencian además de su objetivo de implementación por sus ventajas y desventajas, las mismas que se detallan en la tabla 4.

Tabla 4: Patrones Arquitectónicos - Ventajas y Desventajas.

Patrón arquitectónico	Ventajas	Desventajas
Capas	▪ Cada capa puede ser reutilizada por otro software. ▪ Se puede cambiar una capa, sin afectar a las demás capas. ▪ Es escalable en cuanto a rendimiento y modificabilidad. ▪ Ofrece una fácil comprensión sobre el funcionamiento del software.	▪ Requiere mayor balance de carga. ▪ Dificulta la evaluación del software. ▪ Requiere mayor tiempo de desarrollo.
Pizarra	▪ Es útil cuando para resolver problemas extremadamente complejos. ▪ Facilita la integración entre agentes. ▪ Las fuentes de conocimiento no requieren conocer la identidad de otras fuentes.	▪ Dependencia de los datos. ▪ Es difícil aplicar la reutilización, ya que las fuentes de conocimiento son independientes. ▪ No existe ocultamiento de datos.
Modelo Vista Controlador (MVC)	▪ La vista siempre muestra información actualizada. ▪ Las aplicaciones que implementan este patrón disponen de fácil mantenibilidad y extensibilidad. ▪ Fácil estructuración del código.	▪ Su implementación es costosa. ▪ No todos los lenguajes permiten adoptar este patrón. ▪ Este patrón exige el desarrollo de un mayor número de clases escritas aplicando conceptos de orientación a objetos.

Elaboración. El Autor.

1.1.3. Atributos de Calidad de Software.

Para organizaciones como la IEEE la calidad del software, es el grado con el que un sistema, componente o proceso cumple requerimientos y necesidades específicas del cliente; dichas necesidades exigen el establecimiento de métricas que permitan cuantificar y establecer niveles mínimos de calidad de un sistema de software.

Entre las métricas o atributos de calidad de software más representativos se exponen seis, funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad**;** detallados de forma breve en el siguiente listado**:**

a) **Funcionalidad:** determina la capacidad que tiene el software para cumplir los requerimientos del cliente.
b) **Confiabilidad:** capacidad del software de mantener su nivel de ejecución sobre condiciones regulares en un lapso determinado de tiempo; además precisa que los resultados de la funcionalidad de software sean exactos.
c) **Usabilidad:** estima el esfuerzo que debe realizar el usuario para lograr adaptarse al software.
d) **Eficiencia:** evalúa la capacidad que tiene el software para usar recursos, en relación al funcionamiento y el tiempo de ejecución.
e) **Mantenibilidad:** valora la capacidad y esfuerzo a realizar al momento de realizar modificaciones o corregir errores en el software.
f) **Portabilidad:** define la capacidad del software para ser transferido de un ambiente a otro, sin presentar problemas de despliegue.

1.1.4. Patrones de Diseño.

Según (Blanco, 2011), (Gracia, 2005) y (Tedeschi, n.d.), los patrones de diseño son soluciones simples y elegantes a problemas específicos, comunes del desarrollo de software. Estos proponen una solución establecida y documentada a problemas específicos de software.

(Lea, 1994) En su libro “Christopher Alexander: An Introduction for Object-Oriented Designers” dispone de una descripción de cualidades necesarias de los patrones de diseño:

a) **Encapsulación y abstracción:** cada patrón de diseño debe encapsular un problema y solución como un dominio particular; además deben proporcionar límites necesarios para concretar una solución entorno al problema.
b) **Extensión y variabilidad:** cada patrón debe poseer la capacidad de trabajar e interoperar con otros patrones para dar solución al problema.

c) **Generatividad y composición:** cada patrón posteriormente a ser aplicado genera un resultado, el cual debe coincidir con el contexto inicial de uno o más patrones. Esta subsistencia de patrones podría ser aplicado progresivamente con el propósito de dar una solución integral a un problema.
d) **Equilibrio:** cada patrón de diseño debe ofrecer un balance entre su consecuencia y restricciones para minimizar el problema.

En efecto, los patrones de diseño proponen a los arquitectos, desarrolladores, analistas o expertos en soluciones de software, buena toma de decisiones, y un vocabulario integral que permite a quienes conforman parte del equipo de desarrollo de software comunicarse de manera eficiente, y disponer de simplicidad para mantener los sistemas de software.

Los patrones de diseño se clasifican en tres grupos:

a) Patrones de Creación.
b) Patrones Estructurales.
c) Patrones de Comportamiento.

(Gamma, Helm, Johnson, & Vlissides, 2003) En su libro "*Patrones de Diseño. Elementos de software orientado a objetos reutilizables",* conceptualiza los distintos patrones de diseño junto con su clasificación, la misma que se expone en la tabla 5. Es importante especificar que los nombres de los patrones de diseño se encuentran expuestos junto con su traducción al español.

Tabla 5. Clasificación y especificación de propósito de Patrones de Diseño.

Tipo	Patrón de diseño	Propósito
Creación	**Fábrica Abstracta**	Proporcionar una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
	Constructor	Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
	Método de Fabricación	Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

	Prototipo	Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.
	Único	Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.
Estructurales	**Adaptador**	Convierte la interfaz de una clase en otra interfaz. Permite que las Clases cooperen de forma que no existan Clases con interfaces incompatibles.
	Puente	Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.
	Compuesto	Compone objetos en estructuras de árbol para representar. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
	Decorador	Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
	Fachada	Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
Comportamiento	**Comando**	Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones.
	Intérprete	Define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.
	Iterador	Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

	Mediador	Define un objeto que encapsula cómo interactúan una serie de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
	Observador	Define una independencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualice automáticamente todos los objetos que dependan de él.
	Estrategia	Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Elaboración. El Autor.

1.2. REST (Transferencia de Estado Representacional).

REST según (Fielding, 2000) y (Sun, 2011), es un estilo de arquitectura de software que destaca por un conjunto de principios, políticas o reglas que define una interacción entre componentes. (Patil, 2013) Explica que REST destaca por su funcionamiento e implementación de principios aplicados dentro de un sistema hipermedia distribuido como la WWW, en la figura 1 se representa lo expuesto anteriormente por el autor.

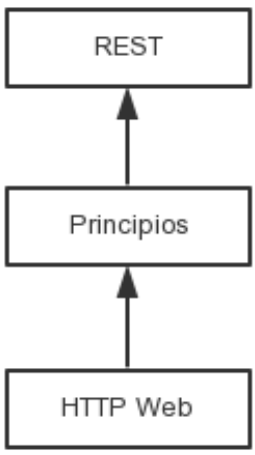


Figura 1. Definición de REST.

Tomado de: (Patil, 2013)

Elaboración. El Autor.

REST se basa en la arquitectura Cliente-Servidor semejante a la figura 2; donde el cliente efectúa una petición a través de un protocolo de transferencia HTTP, con el objetivo de obtener datos o información codificada mediante un formato de datos estructurado que puede ser XML/JSON. Este proceso REST comprende el funcionamiento de los servicios web RESTful, hay que recalcar que RESTful es la denominación que adoptan los aplicativos web basados en REST.

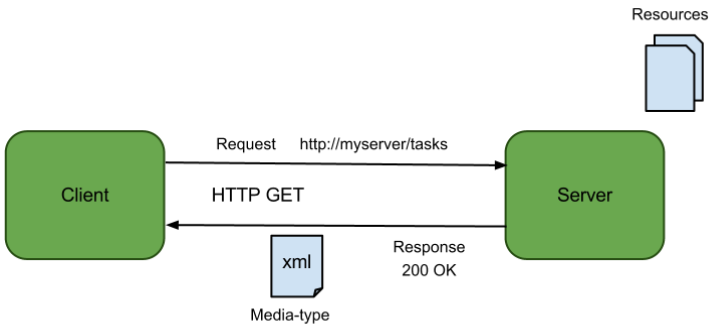


Figura 2. Arquitectura REST.

Fuente. http://prideparrot.com/Source-Codes/Images/REST.png

(Fielding, 2000) En su escrito enfatiza en el cumplimiento de algunos lineamientos a tomar en cuenta para realizar una implementación de software REST, estos son:

- **a)** Usar protocolo de transporte HTTP**.**
- **b)** Los recursos deben estar identificados por una URL
- **c)** Los recursos deben ser representados por formatos establecidos, XML, JSON, HTML, GIF, JPEG, etc.
- **d)** Los recursos tienen que identificarse en base a su tipo MIME, text/xml, text/html, etc**.**

REST adopta conceptos fundamentales, como recursos, verbos, identificadores, cabeceras, entre otros, los mismos que se explican en la tabla 6:

Tabla 6. Conceptos de implementación REST.

Concepto	Teoría
Recursos	Los recursos representan a datos o información.
Identificador	Identifican a los recursos
Verbos	Los verbos simbolizan acciones HTTP, por ejemplo: GET, representa una consulta.

Cabeceras de solicitud	Son los detalles adicionales que se envían cuando se realiza una petición, por ejemplo: tipo de respuesta o detalles de autorización.
Cuerpo de respuesta	Es la respuesta a la petición, usualmente se agrupa en un formato de datos estructurado.
Código de estado de respuesta	Se denominan "*problemas con respuesta*", representan el estado de la solicitud, por ejemplo: 200, significa que la petición se ha resuelto correctamente. Códigos: ▪ **1XX:** Respuestas informativas ▪ **2XX:** Peticiones correctas ▪ **3XX:** Redirecciones ▪ **4XX:** Errores de Cliente ▪ **5XX:** Errores de Servidor

Elaboración. El Autor.

1.2.1. REST – Fundamentos / Principios.

REST exige el cumplimiento de ciertos principios básicos de implementación; su obligatoriedad es indispensable, según (Fielding, 2000) "el software que no cumpla con los principios listados no será reconocido como REST".

El autor en mención define cinco principios básicos:

a) Arquitectura Cliente-Servidor.
b) Sin estado.
c) Caché.
d) Servicios uniformes
e) Arquitectura en capas.

Los mismos que se detallan en los apartados siguientes:

- **Cliente – Servidor:**

REST requiere la implementación de una arquitectura cliente-servidor por qué su funcionamiento precisa el mantenimiento de interfaces aisladas; resolviendo así la necesidad de brindar funcionamiento al sistema de petición y respuesta expuesto por el estilo arquitectónico estudiado.

La figura 3 representa la petición de un cliente al servidor.

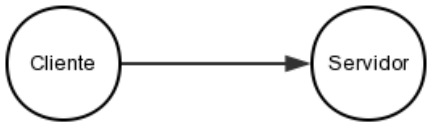


Figura 3. Petición cliente – servidor.
Fuente: (Fielding, 2000)
Elaboración. El Autor.

- **Sin estado.**

(Fielding, 2000), explica que “REST no mantiene un estado directamente con el cliente”; en otras palabras, cada petición que realiza el cliente es complementamente independiente una de otra tal y como se detalla en la figura 4. Para REST es imperativo que el servidor no guarde datos de invocaciones previas con el fin de garantizar la integridad de los datos de respuesta.

Es importante destacar que en REST el uso de cookies o almacenamiento de variables de sesión no refleja una buena práctica, se requiere que en cada petición el cliente envíe los datos necesarios para resolver las peticiones.

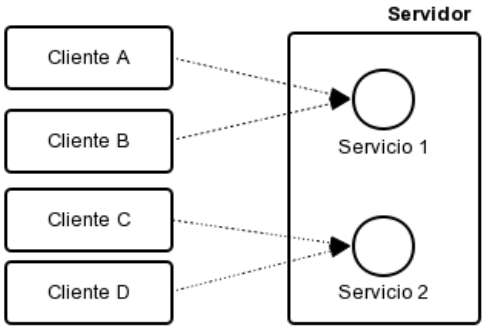


Figura 4. Fundamentos / Principios | Sin estado.
Fuente: (Fielding, 2000)
Elaboración. El Autor.

- **Caché.**

REST ofrece a los usuarios la posibilidad de seleccionar el almacenamiento de los datos en caché; a nivel de servidor (Fielding, 2000) explica que no se permite el almacenamiento de datos, porqué el cliente puede perder integridad en los datos; su teoría concluye que si el cliente realiza una petición y éste mantiene una copia de respuestas previas almacenadas

en caché, los datos de respuesta actuales no se presentan, si no la respuesta se remite a la copia almacenada en caché, en la figura 5 se advierte el almacenamiento de una copia caché a nivel de cliente.

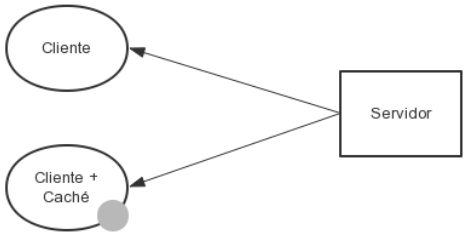


Figura 5. Fundamentos / Principios | Caché.
Fuente: (Fielding, 2000)
Elaboración. El Autor.

- **Servicios Uniformes.**

(Fielding, 2000) Considera que “REST debe mantener el mismo tipo de invocación con los distintos métodos”; es decir los servicios uniformes hacen referencia a la forma en que se ejecuta una petición de recursos, debe ser similar en todas sus acciones, sea estas para consultar, almacenar u otra acción. Hay que subrayar que cada petición ejecuta una acción sobre el servidor. En la figura 6 se observa los cuatro tipos de operación que pueden ejecutarse por parte del servidor, y su similitud entre clases de invocación.

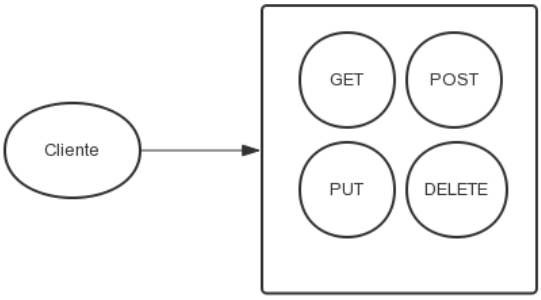


Figura 6. Fundamentos / Principios | Servicios Uniformes.
Fuente: (Fielding, 2000)
Elaboración. El Autor.

- **Arquitectura en Capas.**

Todos los sistemas de software REST según (Caules, 2013) están orientados hacia la escalabilidad, y advierte que “un cliente REST no es capaz de distinguir entre sí está realizando una petición directamente al servidor, o se lo está devolviendo un sistema de caches intermedio o por ejemplo existe un balanceador que se encarga de redirigirlo a otro servidor”. Tal y como se expone en la figura 7. Además REST exige que la construcción de

software implique la adopción de conceptos propios de la arquitectura en capas, permitiendo así entre algunas características que el sistema sea escalable y mantenible.

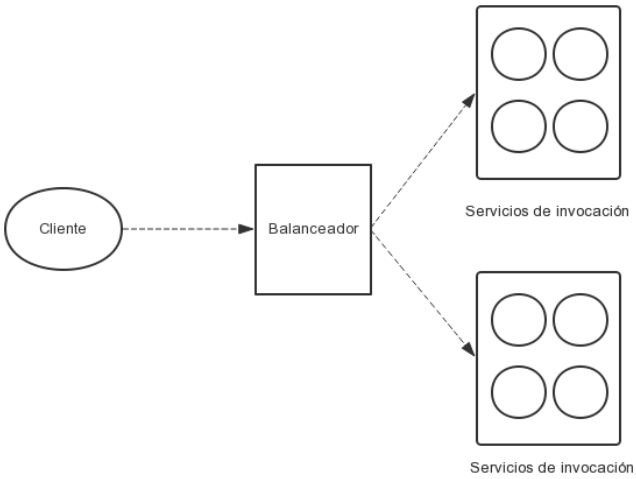


Figura 7. Fundamentos / Principios | Arquitectura en Capas.
Tomado de: (Fielding, 2000)
Elaboración. El Autor.

1.2.2. REST – Propiedades Arquitectónicas.

Según (Fielding, 2000), REST determina el cumplimiento de propiedades arquitectónicas alusivas al estilo arquitectónico estudiado. En la figura 8 se exponen dichas propiedades arquitectónicas, las mismas que enaltecen a REST frente a otros estilos arquitectónicos que pretendan proponer funcionalidades similares.

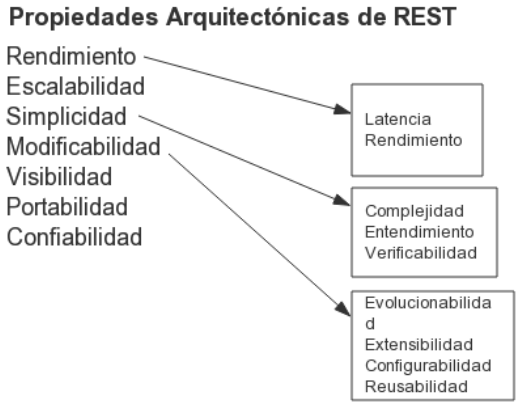


Figura 8. Propiedades Arquitectónicas de REST.
Fuente: (Patil, 2013)

- **Rendimiento.**

En software, esta es quizás la propiedad más visible; para REST el rendimiento es clave en su funcionamiento, para (Fielding, 2000), “el rendimiento es una de las principales razones para centrarse en estilos para las aplicaciones basadas en la red. Se debe a interacciones de los componentes, estos pueden ser el factor dominante en la determinación del rendimiento y la eficiencia de la red percibida por el usuario”.

Visto desde el punto de vista arquitectónico esta propiedad hace que este estilo este siendo ampliamente adoptado. Se busca que quién realice la petición, obtenga resultados inmediatos.

- **Escalabilidad.**

Esta propiedad fomenta la simplicidad que presenta REST hacia el crecimiento. Este atributo puede ser aplicado a distintos puntos de vista, entre los cuales se adapta a: funcionalidades e información. Para (Fielding, 2000) es la aplicación del concepto de separación de componentes dentro de sus funciones permitiendo el crecimiento del aplicativo. En conclusión, es la propiedad que debe tener un software para facilitar el crecimiento en función de las necesidades del cliente.

- **Modificabilidad.**

Determina la facilidad que compromete a un sistema de software para realizar cambios. La adaptación de este estilo arquitectónico exige que se codifique de manera estructurada, tal que permita a los desarrolladores seguir creciendo en funcionalidades y además modificar en caso de ser necesaria la aplicación. (Fielding, 2000) Explica que la modificabilidad de REST puede dividirse en las siguientes áreas:

a) **Evolutividad:** representa el grado en que una implementación del componente puede ser cambiada sin afectar negativamente a otros componentes.
b) **Extensibilidad:** incorpora la capacidad de agregar una nueva funcionalidad a un sistema.
c) **Personalización:** es la capacidad de especializarse temporalmente en el comportamiento de un elemento arquitectónico.
a) **Configurabilidad:** se relaciona directamente con la extensibilidad y la reutilización, se refiere a la modificación posterior a la puesta en producción de un sistema de software.
b) **Reutilización:** se presenta como una propiedad de una arquitectura de software, para que sus componentes, conectores o elementos puedan ser reutilizados, en la misma arquitectura.

- **Visibilidad.**

Según (Fielding, 2000), la visibilidad es la “capacidad de un componente para monitorear o mediar en la interacción entre otros dos componentes”.

Además, explica que la visibilidad puede permitir:

- Mejorar el rendimiento mediante el almacenamiento en caché compartida de las interacciones.
- Escalabilidad a través de servicios en capas.
- Fiabilidad a través del monitoreo reflexivo.
- Seguridad al permitir las interacciones para ser inspeccionado por los mediadores.

Es una forma de aportar al software la capacidad de que esta sea monitoreada y accedida desde cualquier ambiente. Además (Fielding, 2000), precisa que esta propiedad enriquece el cumplimiento de las propiedades arquitectónicas estudiadas.

- **Portabilidad.**

Portabilidad representa la capacidad del sistema de software para ser transportado de un ambiente a otro sin presentar inconvenientes de despliegue a nivel de servidor; asimismo, a nivel de código fuente también exige la capacidad de que la lógica sea reutilizada.

Es importante subrayar que a mayor portabilidad, es menor la dependencia con respecto al ambiente de implementación. De este modo, se busca que los sistemas desarrollados aportando un estilo arquitectónico REST deben ser portátil, y compatible, logrando que sea desplegado desde cualquier ambiente de despliegue de software.

- **Confiabilidad.**

Para (Patil, 2013) y (Fielding, 2000), la confiabilidad es el grado en que sus soluciones y servicios son susceptibles al fracaso; es decir, sus soluciones y resultados ante sus operaciones deben presentar resultados confiables y estar exento de fallos de software.

La confiabilidad en un software REST, no solo implica sus funcionalidades; sino, su cometido viene ligado con la confiabilidad de los datos, ya que al ser un sistema de consulta y respuesta; el usuario espera que sus resultados sean precisos y confiables.

1.2.3. REST – Elementos.

1.2.3.1. Elementos de Datos.

La naturaleza de REST hace de los datos uno de los aspectos más importantes en su funcionamiento. REST posee un funcionamiento donde los componentes de este estilo arquitectónico se comunican enviando mensajes de representaciones de recurso, para obtener una respuesta en un formato de datos estructurado. REST dispone de cuatro elementos recursos, identificador de recursos, representación y representación de metadatos, los mismos que se describen en la tabla 7.

Tabla 7. Elementos de datos REST

Elemento de datos	Descripción
Recurso	El recurso de datos, es equivalente a los datos o información a exponer.
Identificador de recurso	El identificador del recurso comúnmente es la URL. REST dota al cliente de un identificador por cada recurso, habitualmente el identificador debe ser representativo a los recursos.
Representación	Según (Fielding, 2000), una representación es "una secuencia de bytes, más un metadato de representación que describe esos bytes", en resumen es una representación que consiste en datos y metadatos que describen a los datos.
Representación de metadatos	Los metadatos poseen información relevante de los datos, generalmente se tiene información como nombre, tipo de dato, palabras clave, además de controles de cambios que incluyen fecha de creación, fecha de modificación, etc.

Fuente: (Fielding, 2000)

Elaboración. El Autor.

1.2.3.2. Conectores.

Para (Velázquez, n.d.) Los conectores son un mecanismo abstracto que hace posible la comunicación, coordinación y cooperación entre componentes. Basados en REST (Fielding, 2000) explica que se mantiene el uso de conectores para encapsular las actividades de petición y respuesta a los recursos, sin guardar ningún estado.

Se tienen cinco conectores en REST, cliente, servidor, caché, resolutor y túnel, los mismos que se exponen en la tabla 8.

Tabla 8. Conectores REST.

Conector	Descripción	Ejemplos
Cliente	Es quién inicia la comunicación a través de una petición HTTP.	Aplicación Web, Android, iOS.
Servidor	Escucha y responde las peticiones del Cliente.	Apache, Tomcat, Glassfish
Caché	Principalmente se almacena en el navegador, con el fin de reutilizar recursos.	Caché del navegador
Resolutor	Realiza una conexión entre componentes, a menudo se usa DNS como resolutor de nombres de dominio a IP o viceversa.	DNS
Túnel	Transmite la comunicación a través de un límite de conexión, se podría entender su funcionamiento como un puerto por donde se puede enviar y recibir varios tipos de datos.	SOCKS, SSL, Puerto 80

Fuente: (Fielding, 2000)

Elaboración. El Autor

1.2.3.3. Componentes.

(Velázquez, n.d.), explica que “los componentes son una unidad abstracta de instrucciones de software y estados internos que proporciona una transformación de los datos a través de su interfaz”. De manera más detallada se entiende que los componentes de software son imprescindibles y se consideran elementos técnicos necesarios para la puesta en producción de los aplicativos REST, en la tabla 9 se presentan los componentes de REST. Consideremos esta aclaración: Sin infraestructura no tendríamos donde desplegar nuestro aplicativo, por ende estos componentes son esenciales.

Tabla 9. Componentes REST.

Componente	Descripción	Ejemplos
Origen del servidor	Destinatario final de cualquier solicitud o petición.	Apache HTTPD, Microsoft IIS
Gateway / Proxy	Funcionan de forma parecida, ambos proporcionan una interfaz de encapsulación de otros servicios para la conversión de datos, mejora de rendimiento, y seguridad. Cabe mencionar que la diferencia entre Gateway y proxy es que un cliente es el que determina cuando se va a usar Proxy.	Squid, CGI, Reverse Proxy
Agente	Se entiende como un Cliente, inicia una solicitud y se convierte en el destinatario de la respuesta.	Netscape Navigator, Lynx, MOMspider

Fuente: (Fielding, 2000)

Elaboración. El Autor

1.3. Seguridad de software.

La seguridad, contempla un alto grado de criticidad; pues la seguridad desde el punto de vista de software es un bien clave a nivel empresarial; la información debe ser asegurada desde todos los aspectos en relación a datos y software.

(García Rubí & Ríos Clemente, 2011) Define a la seguridad como “un conjunto de recursos destinados a lograr que los activos de una organización sean confidenciales, íntegros, consistentes y disponibles a sus usuarios, autentificados por mecanismos de control de acceso y sujetos a auditoría”.

Existen tres aspectos a tomar en cuenta para el aseguramiento de la información:

1. **Confidencialidad**. Asegura que la información no pueda estar disponible, ni descubierta por terceros.
2. **Disponibilidad**. Garantiza la seguridad de la información, hardware y software, para ser accedido en cualquier momento.
3. **Integridad.** Condición de seguridad que certifica que la información es insertada, actualizada y eliminada por usuarios autorizados.

En referencia a este estudio, es importante considerar el aseguramiento de software tomando en cuenta todas las etapas de desarrollo. Habitualmente el aseguramiento de software viene dado por buenas prácticas de programación, estandarización de mecanismos de seguridad y aplicación de métodos de seguridad a nivel de servidores de aplicaciones.

Fundamentándose en el objeto de estudio de este escrito, se analizará dos temas importantes de seguridad, el aseguramiento lógico y las vulnerabilidades de software.

1.3.1. Seguridad de software: Aseguramiento Lógico.

Consiste en la aplicación de técnicas y procedimientos que limiten el acceso al sistema de software. Existen varios tipos de controles lógicos que permiten el aseguramiento del software entre los cuales se detallan cinco de los principales en la tabla 10.

Tabla 10. Controles de aseguramiento lógico.

Control de aseguramiento lógico	Detalle
Roles	Se crean roles de usuario para limitar el acceso a la información.
Control de acceso	Son controles que se realizan para controlar el acceso de los usuarios a la información.

Autenticación	Constituye la identificación de cada usuario al sistema.
Listas de control de acceso (ACL's)	Filtran el tráfico de red, permitiendo y denegando el acceso basándose en políticas de acceso.
Limitaciones a los servicios	Restricciones que dependen de parámetros propios para restringir el acceso a la información, estas limitaciones las crea el administrador.

Elaboración. El Autor.

Es importante definir el control de aseguramiento lógico en la etapa de diseño arquitectónico de software; usualmente los arquitectos de software adoptan el uso de autenticación con el fin de garantizar el acceso a los distintos niveles de entrada de los sistemas de software.

1.3.2. Seguridad de software: Vulnerabilidades.

(Roldán, 2012), (Somarriba, 2004) y (Prandini & Pallero, 2013) definen a una vulnerabilidad como una debilidad de un sistema informático que es utilizada para causar daño. Las debilidades pueden aparecer en cualquiera de los elementos de un sistema informático, tanto en el hardware, cómo en el software. Para (INTECO, n.d.), "las vulnerabilidades son la piedra angular de la seguridad, puesto que suponen el origen del que derivan numerosos fallos de seguridad".

(Mifsud, 2012), agrupadas a las vulnerabilidades según su función, las mismas que se detallan de acuerdo a su función y causa en la tabla 11.

Tabla 11. Causas de vulnerabilidades.

Función	Causa
Diseño	▪ Debilidad en el diseño de protocolos utilizados en las redes. ▪ Políticas de seguridad deficientes e inexistentes.
Implementación	▪ Errores de programación. ▪ Existencia de "*puertas traseras*" en los sistemas informáticos. ▪ Descuido de los fabricantes.
Uso	▪ Mala configuración de los sistemas informáticos. ▪ Desconocimiento y falta de sensibilización de los usuarios y de los responsables de informática.

	▪ Disponibilidad de herramientas que facilitan los ataques. ▪ Limitación gubernamental de tecnologías de seguridad.
Vulnerabilidad del día cero	▪ Se incluyen en este grupo aquellas vulnerabilidades para las cuales no existe una solución “conocida”, pero se conoce como explotarla.

Elaboración. El Autor.

1.4. OWASP Top 10 2013.

OWASP (Proyecto de Seguridad de Aplicaciones Web, en sus siglas en español) según (Tarlogic, 2013) “es una metodología de seguridad abierta y colaborativa, orientada al análisis de seguridad de aplicaciones Web, comúnmente es usada como referente en auditorías de seguridad”.

El objetivo principal de OWASP, es ayudar a los desarrolladores a escribir código seguro que garantice la seguridad de los sistemas web. OWASP cada tres años socializa una lista actualizada con las diez vulnerabilidades más frecuentes que sufren los sistemas web, su última actualización fue en el 2013. Asimismo (Díaz, 2013) se refiere a esta lista como uno de los principales proyectos de OWASP, y, seguramente, uno de los más conocidos internacionalmente al ser referenciado por numerosos organismos, estándares y libros en la definición de los requerimientos mínimos de seguridad exigidos en los desarrollos de aplicaciones web.

La última actualización de OWASP Top Ten expuesta en la figura 9, considera las vulnerabilidades, ordenadas según la frecuencia de aplicación y robustez de afectación en las aplicaciones web.

OWASP Top 10 – 2013 (Nuevo)
A1 – Inyección
A2 – Pérdida de Autenticación y Gestión de Sesiones
A3 – Secuencia de Comandos en Sitios Cruzados (XSS)
A4 – Referencia Directa Insegura a Objetos
A5 – Configuración de Seguridad Incorrecta
A6 – Exposición de Datos Sensibles
A7 – Ausencia de Control de Acceso a las Funciones
A8 – Falsificación de Peticiones en Sitios Cruzados (CSRF)
A9 – Uso de Componentes con Vulnerabilidades Conocidas
A10 – Redirecciones y reenvíos no validados

Figura 9: OWASP Top 10 - 2013.
Fuente. (OWASP, 2013).

Como objeto de estudio se analizará y mitigará dos de las diez vulnerabilidades del listado OWASP: inyección y secuencia de comandos en sitios cruzados más conocida como XSS, porqué estas dos vulnerabilidades son las que mayormente afectan a las aplicaciones web.

1.4.1. Inyección SQL.

Estos ataques ocurren cuando se envían datos a través de un intérprete como parte de una consulta; generalmente se pretende alterar las cadenas de tipo SQL. Frecuentemente ocurre por una mala práctica de programación aplicada en la codificación de cadenas SQL para ejecutar operaciones sobre los datos.

Específicamente este tipo de vulnerabilidad pretende modificar las consultas inyectando código SQL; OWASP cataloga a los ataques de inyección, como la principal técnica de extracción de información de aplicaciones web. Se resumen los riesgos de estos ataques en la Figura 10.

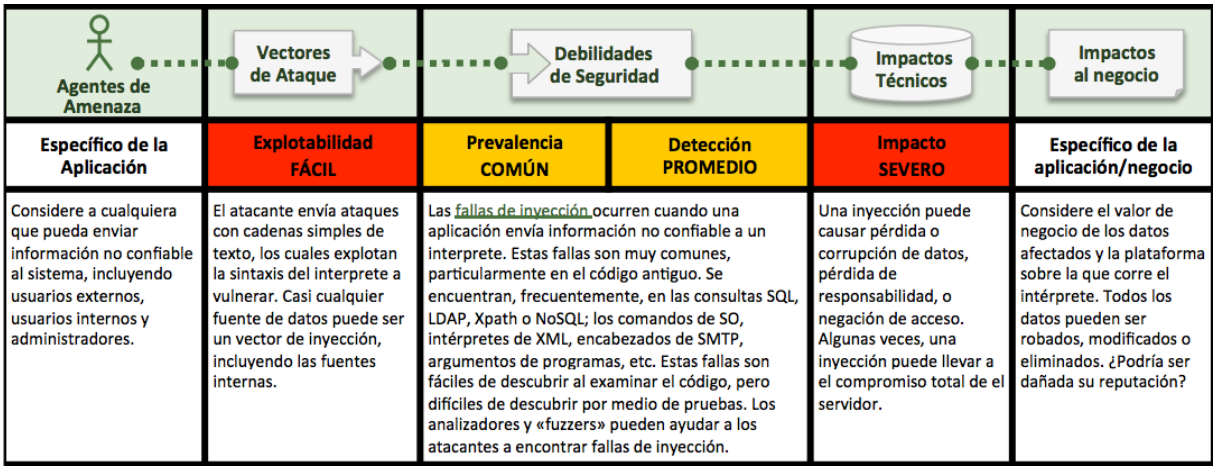


Figura 10: Riesgos de ataques de inyección.

Fuente. (OWASP, 2013).

Para comprender, se analiza la siguiente hipótesis:

Se pretende consultar por medio de un servicio web los datos de un alumno identificado por su número de identificación; la consulta SQL sería:

- SELECT * FROM alumno WHERE cédula = ‘1104542749’.

A nivel de código fuente, los desarrolladores codifican aplicando técnicas de concatenación para completar la consulta SQL quedando de esta manera:

- String SQL = “SELECT * FROM alumno WHERE cédula =’ ” + parámetro + “ ‘ “;

Donde se inicia declarando la variable que contiene la consulta concatenada con el parámetro de búsqueda. Considerando que un atacante inyecta a través del parámetro de búsqueda *‘ or ‘1’=’1* la consulta resultaría:

- String SQL = “SELECT * FROM alumno WHERE cédula =” ***‘ or ‘1’ = ‘1*** *‘* “;

Formando una cadena SQL válida que va a retornar los datos de todos los alumnos, sin tomar en cuenta el parámetro de búsqueda, resultando una exposición de información sensible al atacante.

Existen algunas cadenas de código SQL que se emplean para obtener información de la base de datos, servidor y esquema de datos, seguidamente se detallan algunas de estas porciones de código malicioso:

- **' and 1=convert(int,** @@**version)-- :** obtiene la versión y motor de base de datos
- **' and 1=convert(int,** @@**servername)-- :** obtiene el nombre del servidor

- **null union all select 1,2,3,4,concat(table_name,char(58),column_name),6, 7,8,9,10,11,12,13,14,15 from information_schema.columns—:** obtiene el nombre de las tablas, columnas e información referente al esquema de base de datos.

1.4.2. Secuencia de comandos en sitios cruzados XSS.

Este tipo de vulnerabilidad habitualmente conocida como XSS, es un tipo de debilidad similar a inyección SQL, este tipo de inseguridad se centra en inyectar código JavaScript con el objetivo de sustraer información valiosa, y romper seguridades lógicas.

Las fallas de este tipo de vulnerabilidad ocurren, cuando se envían datos y no se realiza una validación respectiva a los mismos, permitiéndoles a los atacantes secuestrar sesiones de usuario, o destruir sitios web a través de la inyección de código JavaScript.

OWASP identifica los riesgos XSS en la figura 11.

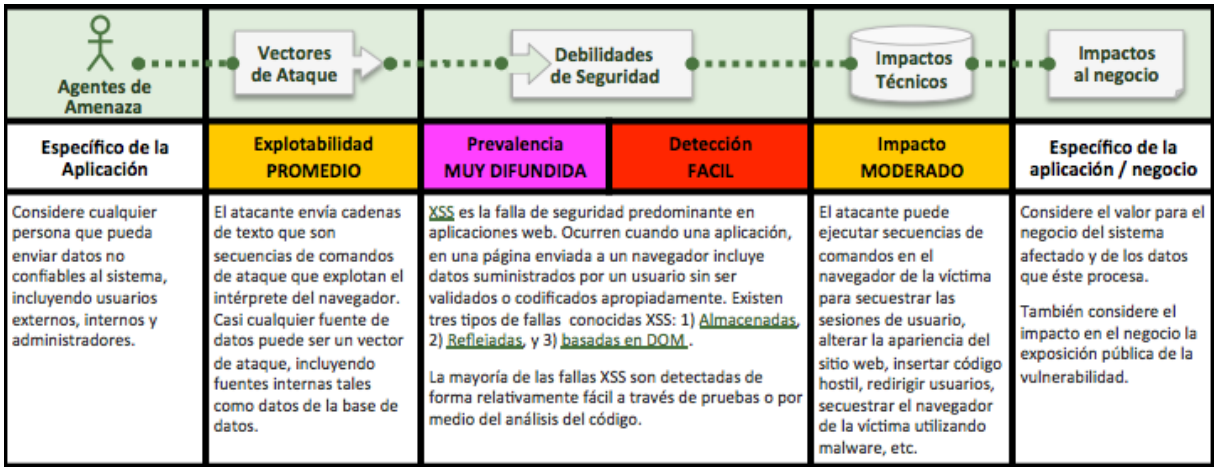


Figura 11: Riesgos XSS.
Fuente. (OWASP, 2013).

Ejemplificando este tipo de debilidad; se considera el siguiente ataque.

Se dispone de un formulario a usuarios finales en el que se solicitan datos personales como nombres y apellidos en el que el usuario final se prepara a realizar un ataque de tipo XSS, para lo cual el atacante ingresa código malicioso para descifrar si la aplicación es vulnerable a XSS; el atacante ingresa un script básico intentando generar una alerta JavaScript:

- <script>alert(“hello world hacking”)<script>

Donde los campos junto con la inclusión del código malicioso se exponen al igual que en la figura 12.

Figura 12: Ejemplo de ataque XSS.
Elaboración. El Autor.

Si posterior al almacenamiento de los supuestos datos personales, se ejecuta el código JavaScript inyectado, generando una alerta similar a la se presenta en la figura 13.

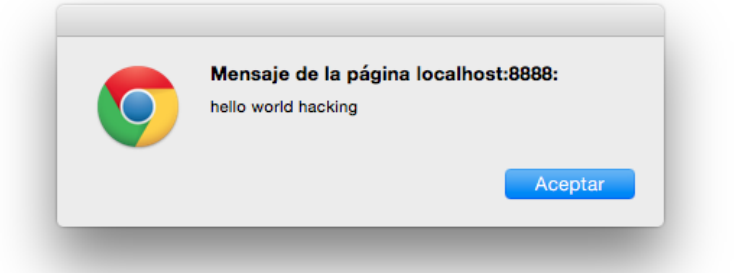


Figura 13: Ejecución de ataque XSS.

Finalmente esta posibilidad facilita al atacante el descubrimiento del tipo de debilidad que sufre el software; y podría proceder con ataques más severos.

1.5. Alcance de estudio.

En base al estudio realizado, se construirá una aplicación web para gestionar servicios web RESTful. Este prototipo de implementación servirá para exponer información sobre el seguimiento de Proyectos de Fin de Titulación de los estudiantes de Ingeniería en Sistemas Informáticos y Computación de la UTPL.

La implementación se limitará a los siguientes aspectos:

- **Verbo HTTP:** GET.
- **Patrón de diseño arquitectónico:** Facade (Fachada, en español).
- **Control de Seguridad lógica:** Autenticación.
- **Recomendaciones de buenas prácticas de codificación:** OWASP.
- **Vulnerabilidades:** Inyección SQL, XSS.

Estos aspectos se justifican en los apartados siguientes, en función de: patrón de diseño, control de aseguramiento lógico y vulnerabilidad:

- **Patrón de diseño arquitectónico: Facade.** provee una lógica sencilla e involucra que la programación sea estructurada. Facade logra abstraer una capa de otra, de

manera que si una capa requiere cambiar no hace falta modificar el resto; lo cual hace que la codificación sea ordenada.

- **Control de seguridad lógica: Autenticación.** como se estudió REST se basa en conceptos HTTP; los servicios web son expuesto en Internet para que sea consumido desde cualquier dispositivo móvil o aplicación web, es por ello que se busca a través de este control garantizar así el acceso a los datos.
- **Vulnerabilidad: Inyección SQL y XSS.** corresponden a la primera y tercera dentro del listado de OWASP Top 10 2013. Además es importante detallar que no solo OWASP hace referencia a los ataques de inyección como los más comunes; si no, organizaciones como WASC (Consorcio de Seguridad en aplicaciones Web, en sus siglas en español) y CVE International (Vulnerabilidades y Exposiciones comunes, en sus siglas en español) hacen mención y clasifican a los ataques de tipo Inyección en los primeros lugares.

Además, el desarrollo de los servicios web RESTful empleará los recursos tecnológicos necesarios para gestión de base de datos, escritura de código fuente y servidor de aplicaciones. Los recursos seleccionados se presentan a continuación:

- **Base de datos**

Se utilizará MySQL, por qué este es un sistema para gestión de base de datos que se encuentra disponible con licenciamiento gratuito. Y que usualmente es usado como repositorio de datos de aplicaciones web, hay que mencionar también que es multiplataforma lo cual contribuye a la portabilidad que exige REST, estas bases de datos pueden ser levantadas en cualquier ambiente de sistema operativo, y otorga al desarrollador compatibilidad con la mayoría de lenguajes de programación.

- **Código fuente.**

Se utilizará JAVA como lenguaje de programación, en su versión empresarial (JAVA EE), ya que es un lenguaje de programación de alto nivel y frecuentemente es la elección principal en el desarrollo de software a nivel empresarial. Este lenguaje posee diversas implementaciones, Frameworks y librerías desarrolladas para dar soporte a la producción de aplicaciones web basadas en REST.

El desarrollo se apoyará en JAX-RS; API propia del lenguaje escogido para la construcción de servicios web RESTful, esta concede el soporte necesario para el desarrollo a realizar mediante el uso de anotaciones que simplifican la codificación.

Esta selección contribuye al cumplimiento de algunos de atributos de calidad que exige una implementación REST, los mismos que se ven expuestos en la tabla 12. En referencia a la similitud entre características JAVA y los atributos de calidad de REST.

Tabla 12. Atributos de calidad de REST y Características java

Atributos de Calidad REST	Características JAVA	Justificación
Rendimiento	**Herramientas de despliegue**	JAVA dispone de sus propios servidores, lo que garantiza la compatibilidad entre componentes (aplicación y servidor), la misma que otorga un mayor rendimiento a la aplicación REST.
Escalabilidad	**Simple**	La codificación en JAVA contribuye a la simplicidad, de modo que la beneficia la adaptación de los desarrolladores a este lenguaje, esta contribución y la forma de programación que expone JAVA asiste y enfoca hacia la escalabilidad del aplicativo. REST requiere sistemas capaces de crecer en cuanto a funcionalidades.
Modificabilidad	**Programación Orientada a Objetos**	La programación Orientada a Objetos que propone JAVA aporta a la modificabilidad de los aplicativos web, porqué expone a que la codificación sea estructurada y ordenada concediendo la posibilidad de que el código sea más sencilla de modificar.
Portabilidad	**Multiplataforma**	Los sistemas de software desarrollados en JAVA se pueden desplegar en cualquier ambiente a nivel de Sistema Operativo, y en algunos casos servidores de aplicaciones que soporten dicho lenguaje.
Confiabilidad	**Confiabilidad y Estabilidad**	JAVA es un lenguaje de programación muy potente, extensa y estable; permite construir desde aplicaciones muy sencillas, hasta aplicaciones empresariales sin perder la confiabilidad que le caracteriza.

Elaboración. El Autor

Es importante destacar el uso de JAVA EE y JAX-RS, frente a un Framework de programación de software REST; JAVA EE facilita la integración entre aplicación y servidor de aplicaciones, no requiere del uso de librerías ni implementos adicionales a la codificación, mientras que los Frameworks de desarrollo, si bien facilitan al desarrollador la programación, no colaboran portabilidad ni compatibilidad frente a cualquier ambiente de despliegue; estos hacen frente únicamente al entorno para el que fueron concebidos; como ejemplo, Spring Framework; los aplicativos desarrollados en este Framework únicamente pueden ser desplegados en Apache Tomcat y requiere de todas las librerías empleadas para su compilación haciendo del software más pesado en cuanto a recursos externos. Frente a ello es conveniente citar que los aplicativos desarrollados usando implementaciones de JAVA EE, permiten ser desplegados en cualquier servidor de aplicaciones que soporte JAVA.

Además es importante especificar que se utilizará JPA (API de persistencia JAVA, en sus siglas en español) en su versión 2.1 para aportar y gestionar la persistencia de la aplicación web REST a desarrollar. La implementación de la capa de persistencia permitirá a los servicios web, conectarse a diferentes motores de base de datos que posean un esquema de datos similar.

Finalmente y no menos importante se destaca la selección del IDE de programación en el que se desarrollará la solución JAVA será Netbeans en su versión 8.0.2.

- **Servidor de Aplicaciones.**

El estudio y selección del servidor de aplicaciones dio como resultado Glassfish en su versión 4.1 como servidor de aplicaciones, resultando entre otras opciones la que más se acercaba a nuestras necesidades de desarrollo. A partir de esto y tomando en cuenta que Glassfish es una implementación propia de JAVA EE, la compatibilidad, implementación y servidor brinda la afinidad necesaria para respaldar el despliegue del aplicativo RESTful. Por otra parte contribuye al empleo de la seguridad lógica para el acceso a los datos RESTful.

Glassfish como una alternativa a Apache Tomcat. Se presenta una comparación entre estos dos servidores en la tabla 13; el contraste entre estos dos servidores se hizo a partir de las necesidades de implementación.

Tabla 13. Contraste entre servidores de aplicación.

FUNCIONALIDAD	Tomcat	Glassfish
Niveles de adopción	Alto	Medio
Escalabilidad	Medio	Alto
Librerías Java EE	No	Si
Ataques de Inyección	No	Si
Entidades de OpenJPA incluidas	No	Si
Autenticación JDBC	No	Si
RENDIMIENTO		
Operaciones/segundo	6615,3	6988,9
Tiempo medio respuesta	0,358	0,242
Tiempo máx. respuesta	3,693	1,519
90% tiempo respuesta	0,75	0,6
Operaciones/segundo	6615,3	6988,9

Fuente: http://www.asjava.com/tomcat/glassfish-vs-tomcat/

Elaboración. El Autor.

Cabe destacar que Apache Tomcat es la solución mayormente adoptada; Glassfish es una solución actualmente en auge por tanto, por las características que posee y su estudio relativamente débil, incentivan a la selección, aplicación y estudio de esta tecnología.

CAPÍTULO II

DISEÑO DE LA SOLUCIÓN.

En busca de minimizar las vulnerabilidades de tipo Inyección SQL y XSS de las aplicaciones web basadas en REST en este capítulo se plantea una solución que garantice el aseguramiento de las mismas en función de diseño arquitectónico, servicios web y modelo de datos. Además se presenta una selección de recursos tecnológicos, y consideraciones fundamentales que respalden la integridad, disponibilidad, funcionalidad y seguridad de datos y software, utilizando técnicas y recomendaciones provistas por OWASP.

La construcción de la aplicación REST servirá para proveer de servicios web a usuarios finales, con la finalidad de que titulaciones; entre estas, Coordinadores de Titulación, Docentes y Estudiantes como actores principales puedan cumplir con el objetivo de dar seguimiento a los Proyectos de fin de Titulación.

2.1. Diseño – Arquitectura de Software.

Partiendo de la necesidad de desarrollar servicios web RESTful para exponer información relevante que aporte a dar seguimiento a los proyectos de fin de titulación, se ha considerado como modelo, la arquitectura que propone JAX-RS (API Java para Servicios Web RESTful, en sus siglas en español), la misma que se representa en la figura 14.

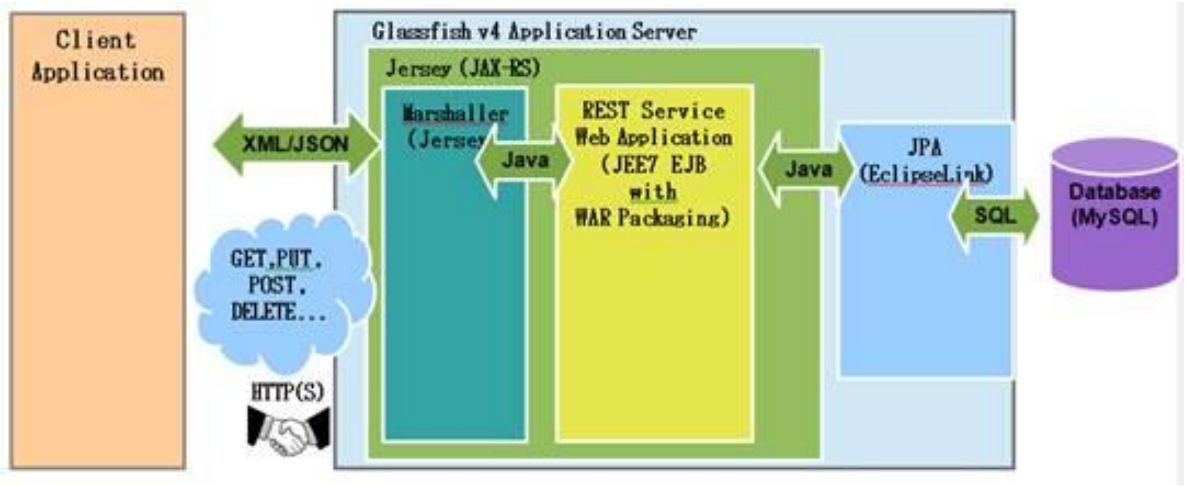


Figura 14. Arquitectura Jersey JAX-RS

Fuente. http://www.developer.com/java/creating-restful-web-services-with-jax-rs.html

Esta arquitectura es una solución de JAVA EE que se desarrolla en base a conceptos de JAX-RS. En este modelo se observa las distintas interacciones entre componentes; además de su protocolo de transporte HTTPS y su servidor de aplicaciones.

Para la solución que se plantea, la arquitectura de software se basa en el modelo anteriormente mencionado más un componente de seguridad como método de autenticación y acceso a los datos del servicio web, los mismos que se detallan en el diseño arquitectónico propuesta en la figura 15.

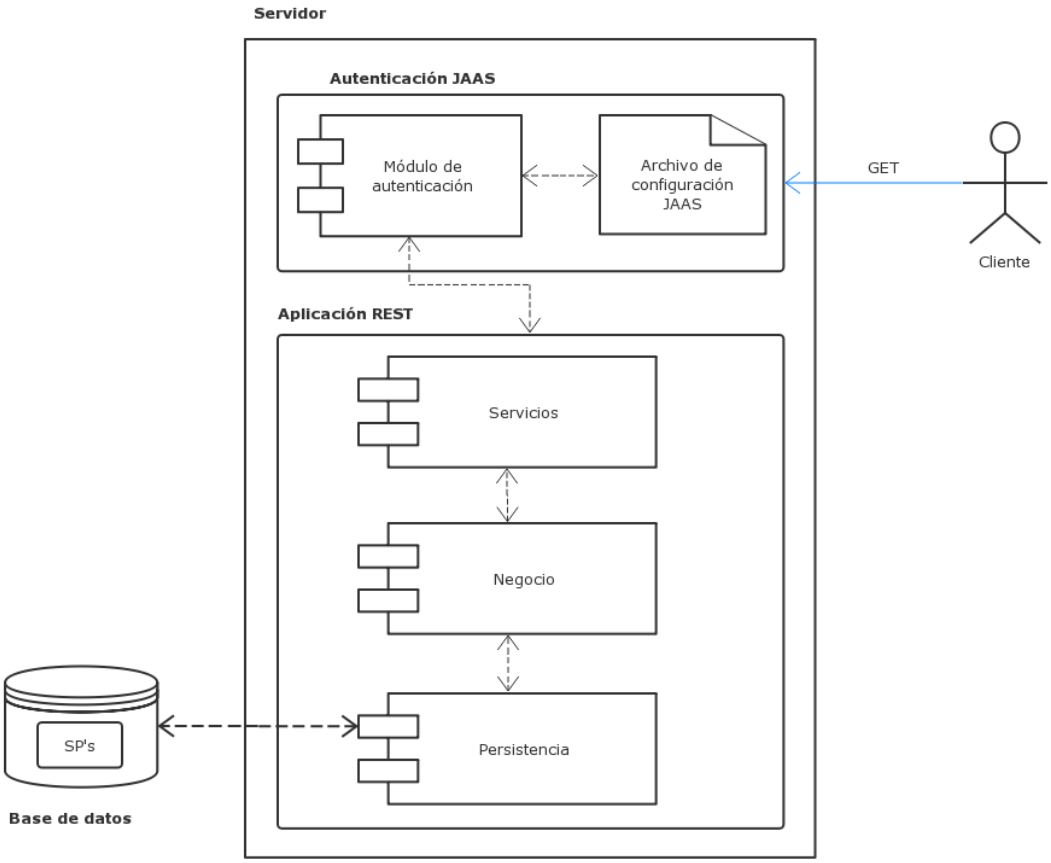


Figura 15. Modelo de Arquitectura REST planteado.

Elaboración. El Autor.

Donde se tiene los siguientes elementos arquitectónicos:

- **Aplicación REST.**

Componente principal de la solución; contiene tres subcomponentes que interactúan entre sí con el objetivo de dar solución a las peticiones que realizará el cliente.

a) **Persistencia:** interactúa directamente con la base de datos, y otorga la capacidad de integrarse a cualquier motor de base de datos, siempre que se considere un esquema similar de datos.
b) **Negocio:** contiene la lógica necesaria para extraer los datos, además de una clase denominada AbstractFacade que representará la fachada del patrón de diseño

seleccionado y ApplicationConfig para configuración, encargada de asociar los archivos necesarios que conlleven al funcionamiento de los servicios RESTful.

c) **Servicios:** serán el componente encargado de consumir la lógica del negocio, específicamente de la clase del componente negocio AbstractFacade, para exponer los datos mediante el uso de los servicios RESTful.

Tanto el componente de negocio y servicios son parte del patrón de diseño seleccionado; el componente de negocio será la fachada y el componente de servicios será quién consumirá el razonamiento de la fachada.

- **Autenticación JAAS:**

La autenticación será el control de seguridad lógica, dotará de seguridad a los servicios web; esta técnica permitirá gestionar el acceso a los servicios web. Este componente permitirá configurar ciertos niveles de privacidad y acceso.

- **Base de datos:**

Este componente es el elemento base de donde partirá la construcción los servicios web, a partir de los datos recopilados se definirá los recursos a exponer. Uno de los propósitos de los servicios web es que el cliente no conozca el esquema de la base de datos por seguridad de la información. También importante citar que una base de datos contribuye a la integridad y seguridad de los datos.

- **Servidor:**

El servidor de aplicaciones gestionará y desplegará la aplicación RESTful que a su vez dará respuesta a las invocaciones de los servicios web. La elección del servidor de aplicaciones conlleva un compromiso por parte de quién participa del diseño, puesto que el servidor debe proporcionar rendimiento y estabilidad necesaria para que los servicios estén disponibles siempre.

- **Cliente:**

Será el destinatario final de los recursos, el cliente puede definirse como una aplicación web, hasta un dispositivo móvil que trabaje sobre cualquier sistema operativo móvil, entre las más populares Android, iOS, entre otros.

2.1.1. Flujo arquitectónico.

REST exige el uso de una arquitectura tipo Cliente – Servidor; por tanto, se detalla la relación o interacción entre actores: cliente y servidor, según el modelo arquitectónico planteado en la figura 16.

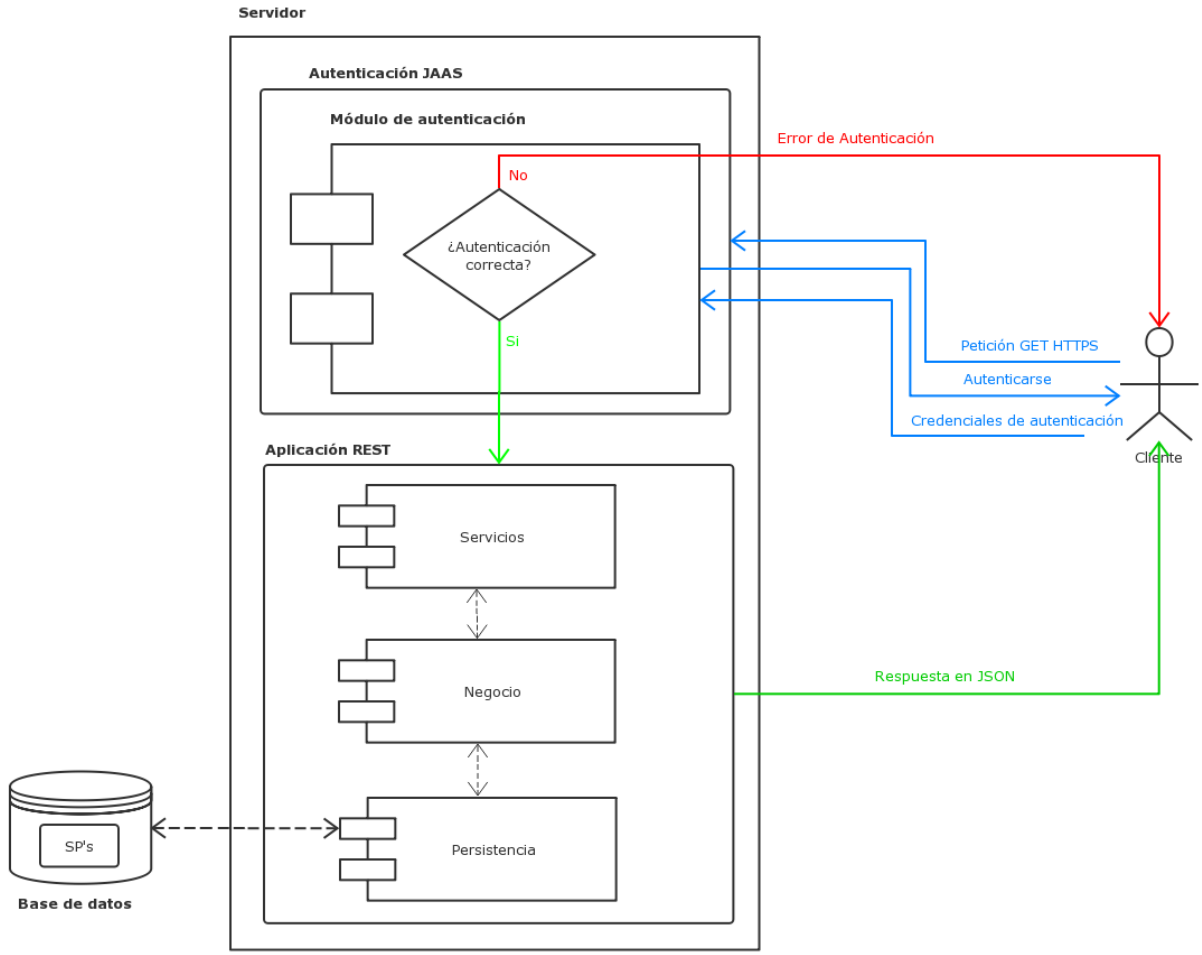


Figura 16. Flujo de diseño arquitectónico.

Elaboración. El Autor.

De acuerdo al flujo se especifica una secuencia lógica de pasos para entender el proceso desde la petición hasta la respuesta del servidor; se detalla la sucesión de pasos necesarios para demostrar el vínculo entre componentes del diseño arquitectónico:

1. El Cliente realiza una petición GET HTTPS al servidor.
2. El servidor responde, solicitando al cliente su autenticación.
3. El Cliente envía sus credenciales de autenticación.
4. El módulo de autenticación comprueba las credenciales de acceso, y decide si el solicitante tiene permisos para acceder a los servicios web. Si tiene permisos invoca a la aplicación REST; sino, retorna al Cliente un mensaje de error de autenticación.
5. Considerando que el cliente realizó una autenticación satisfactoria, el servidor ejecuta la aplicación REST; de ahí, se ejecuta la interacción entre componentes del aplicativo para obtener los datos solicitados. Internamente se ve que:

a. Primeramente, el componente de servicios invoca a la funcionalidad correspondiente desarrollada en el componente de negocio con el objeto de consumar la solicitud del cliente.
b. En segundo lugar, según el método invocado por el componente de servicios web, el componente de negocio; que es quién contiene la lógica necesaria para resolver las peticiones, solicita a la persistencia se ejecute la llamada correspondiente al Procedimiento Almacenado necesario para cumplir con el requerimiento.
c. Luego es el componente de persistencia quién se encarga de negociar e interactuar con el motor de base de datos, para dar respuesta a la solicitud expuesta por el cliente.

6. Finalmente una vez resuelta la petición del cliente por parte del aplicativo; se retorna la información solicitada por el cliente, en un formato de datos estructurado.

De esta manera, mediante la arquitectura propuesta se pretende construir y asegurar los distintos servicios web que cumplan las funcionalidades propuestas.

2.2. Diseño – Servicios Web.

El diseño de servicios web se basa en la adopción de buenas prácticas de desarrollo de aplicaciones RESTful. Cada servicio web diseñado servirá para identificar recursos obtenidos desde la base de datos; además se utilizará JSON para estructurar la información y que esta sea entendible para humanos y máquinas.

La operación soportada por dichos servicios web se referirá a la acción de consultar datos mediante el verbo HTTP: GET. El funcionamiento de esta acción se detalla en la figura 17, donde se aprecia las partes de un servicio web, tomando como ejemplo una petición y respuesta de un servicio RESTful.

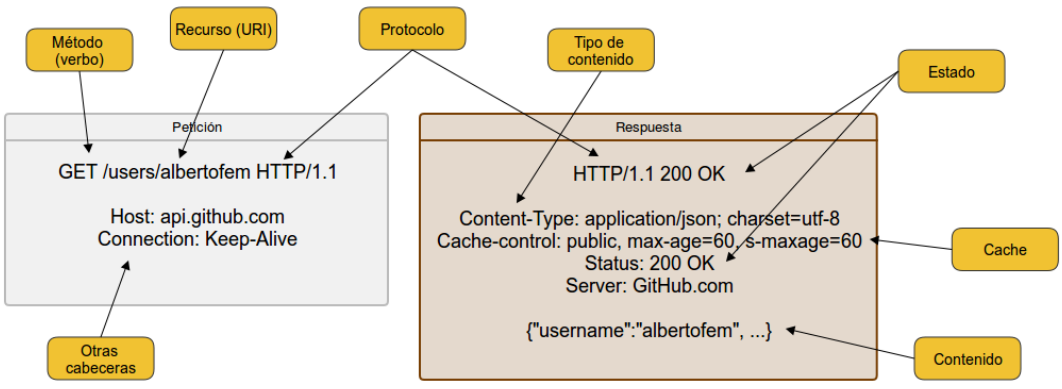


Figura 17. Ejemplo de petición-respuesta de un servicio RESTful.

Fuente. (Fernández, 2013)

2.2.1. Identificadores RESTful.

En esta sección se define los identificadores de recursos (URI) a exponer, los mismos que detallan en la tabla 14. Es primordial destacar que los identificadores se construyen a partir de la información que el cliente desea exponer; de este modo, según las necesidades del prototipo planteado se diseñó los siguientes identificadores:

Tabla 14. Diseño | Identificadores RESTful.

URI	Parámetro	Recurso
https://host/pft/ws/persona	No requiere parámetro	Listará los datos de todas las personas.
https://host/pft/ws/persona/{cedula}	cédula	Listará una persona filtrada por el número de cédula.
https://host/pft/ws/persona/{cedula}/proyecto	cédula	Mostrará el proyecto que tenga asignado una determinada persona.
https://host/pft/ws/persona/{rol}/{cedula}	rol, cédula	Filtrará a las personas por el rol de la persona y número de cédula.
https://host/pft/ws/programa	No requiere parámetro	Listará todos los programas ofertados por la UTPL.
https://host/pft/ws/programa/{modalidad}/{nivelacademico}	modalidad, nivel académico	Programas ofertados por la UTPL, filtrados por la modalidad de estudio y el nivel académico.
https://host/pft/ws/proyecto/{estado}	estado	Se listarán los proyectos filtrados por el estado de aprobación.
https://host/pft/ws/proyecto	No requiere parámetro	Listará todos los proyectos.
https://host/pft/ws/programa/buscar/{nombreprograma}	nombreprograma	URI para búsqueda de programas por el nombre del

		programa, puede ser filtrado por una letra o una parte del nombre del programa.
https://host/pft/ws/proyecto/buscar/{nombreproyecto}	nombreproyecto	URI para búsqueda de proyectos por el nombre del proyecto, puede ser filtrado por una letra o una parte del nombre del proyecto.

Elaboración. El Autor

2.3. Diseño – Modelo de Datos.

El diseño de modelo de datos nace a partir información seleccionada para exponer como recursos y las distintas interacciones determinadas en relación a software-datos. Dicho modelo se presenta de acuerdo al Anexo A, que corresponde al modelo de datos para dar seguimiento a los Proyectos de Fin de Titulación de la titulación de Sistemas Informáticos y Computación.

Es importante destacar que para cumplir con el objetivo de esta investigación únicamente la implementación se limitará a una sección del modelo de datos, el mismo que se expone en la figura 18.

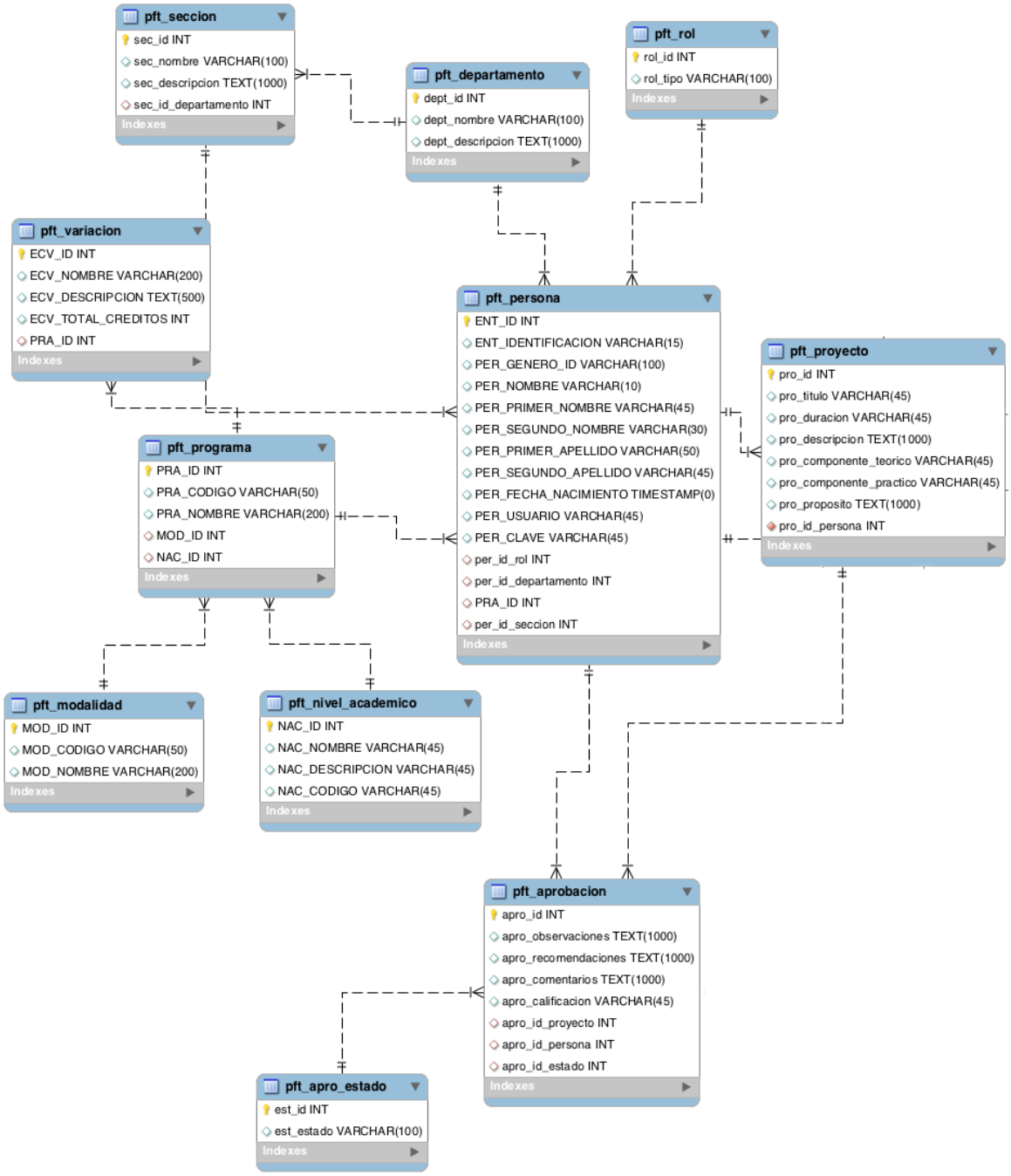


Figura 18. Diseño | Modelo entidad relación de base de datos.

Elaboración. El Autor.

Esta sección corresponde al envío y estado de propuestas, programas, modalidades, personas y tipos de personas, donde se alojarán los recursos necesarios para dar respuesta a las solicitudes del cliente.

2.4. Técnicas de seguridad.

Las técnicas de seguridad seleccionadas, se referencian de acuerdo a las vulnerabilidades estudiadas Inyección SQL y XSS, se aplicarán normativas OWASP para asegurar los componentes diseñados de acuerdo a la arquitectura planteada. En particular deseo puntualizar que el tema de seguridad informática es crítico hoy en día por tanto es necesario adoptar todas las recomendaciones disponibles con el objetivo de minimizar las vulnerabilidades especialmente en aplicaciones web; comúnmente expuestas en internet. Se debe recordar que los atacantes están frecuentemente alertas a encontrar formas de vulnerar el software web, razones por las cuales este estudio se enfocará al aseguramiento en tres niveles: 1) Base de datos, 2) Codificación, 3) Servidor.

2.5.1. Nivel I: Base de datos.

En este nivel se adoptará el uso de procedimientos almacenados, está técnica se basa en el desarrollo de pequeños programas almacenados físicamente en la base de datos; estos ofrecen algunas ventajas adicionales al aseguramiento de los datos en el desarrollo de servicios web, entre estas ventajas se tiene:

- **Acceso y respuesta más rápido:** los procedimientos almacenados trabajan directamente con el motor de base de datos, lo cual permite un acceso directo a los datos y evita la sobrecarga resultante de las consultas SQL usualmente aplicadas para la extracción de datos.
- **Seguridad:** esta técnica es concebida con la idea de exponer únicamente los datos necesarios; por tanto, evita exponer por completo el esquema de base de datos a los desarrolladores y demás participantes en el proceso de desarrollo de software.
- **Independencia:** otorga total independencia a los datos de la codificación, por qué la lógica se encuentra en los procedimientos almacenados y no en la codificación; por ejemplo, si se requiere realizar un cambio a nivel del resultado de los datos, se puede modificar el procedimiento almacenado sin afectar al software como tal.

Los procedimientos almacenados son una recomendación de OWASP, para la prevención de ataques de inyección SQL, aportando no solo a la seguridad, si no al rendimiento del software REST.

2.5.2. Nivel II: Codificación.

OWASP provee de varías guías donde detalla distintos aspectos y técnicas de aseguramiento de software, entre estas se recurrirá a las siguientes.

- **Consultas parametrizadas:** OWASP recomienda el uso de consultas parametrizadas, esta práctica obliga al desarrollador a definir la consulta SQL y luego definir los parámetros a enviar, de este modo se logra distinguir entre la consulta y su parámetro. A continuación se muestra un ejemplo de implementación JAVA EE usando consultas parametrizadas.

  - ```
    String SP = "CALL listarPersona(?)";
    Query q = getEntityManager().createNamedQuery(SP);
    q.setParameter(1, cedula);
    ```

Este método ejemplificado, permite llamar a un procedimiento almacenado y envía un parámetro mediante la técnica seleccionada.

Para mayor comprensión se distinguirá entre una buena y mala práctica de programación en la tabla 15. Para ello se identificará su resultado en base a una cadena de ataque SQL: **' or '1'='1**.

Tabla 15. Buenas y malas prácticas de programación.

Práctica	Codificación	Resultado

Buena	`String SP = "CALL listarPersona(?)";`
	`Query q = getEntityManager().createNamedQuery(SP);`
	`q.setParameter(1, cedula);`
		Error 500

Mala	`String SQL = "SELECT * FROM persona WHERE cédula =' "+ cédula + " ' ";` `Query q = getEntityManager().createNamedQuery(SQL);`	Datos de todas las personas.

Elaboración. El Autor.
    ```

Donde se advierte que la mala práctica da un fragmento SQL resultante válido:

*SELECT * FROM persona WHERE cedula = '' OR '1' = '1'*; el mismo que se convierte en una cadena comprensible para el motor de base de datos y concluye retornando todos los datos almacenados de personas, según la cadena SQL; mientras que la combinación de consultas parametrizadas y procedimientos almacenados arroja un error porque se desconoce la cadena de ataque, esto sucede por qué la consulta parametrizadas distingue al parámetro de la consulta es decir toma al vector de ataque " *' or '1' = '1* " como una sola cadena y produce un error preciso a que dicho parámetro no pertenece a una cadena de consulta válida.

- **Método de limpieza de caracteres especiales:** Este método se construirá con la finalidad de tener un filtro de caracteres especiales. Tanto XSS como Inyección SQL son vulnerabilidades similares que permiten al atacante inyectar código malicioso con la intensión de obtener información o romper seguridades lógicas de los sistemas web. Por tanto, este método se encargará de limpiar el parámetro ingresado por el usuario previo a pasar un parámetro de consulta, este método pretende limpiar los caracteres especiales reemplazándolos por códigos de referencia.

  Como se afirmó previamente se detalla este proceso en la figura 19.

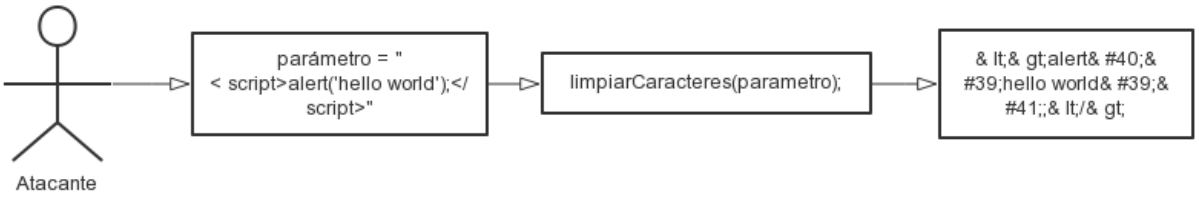


Figura 19: Proceso de limpieza de caracteres especiales.
Elaboración. El Autor.

- **Control de respuestas HTTP:**

El control de respuestas HTTP es muy significativo, ya que estos retornan información valiosa con cada respuesta HTTP. Cuando se producen errores, las respuestas HTTP retornan información delicada detallando la razón de los errores de forma similar a la figura 20; por tal motivo se controlarán las respuestas del servidor, para ello se ejecutará un proceso simple; se crearán páginas HTML que muestre un mensaje de "Error".

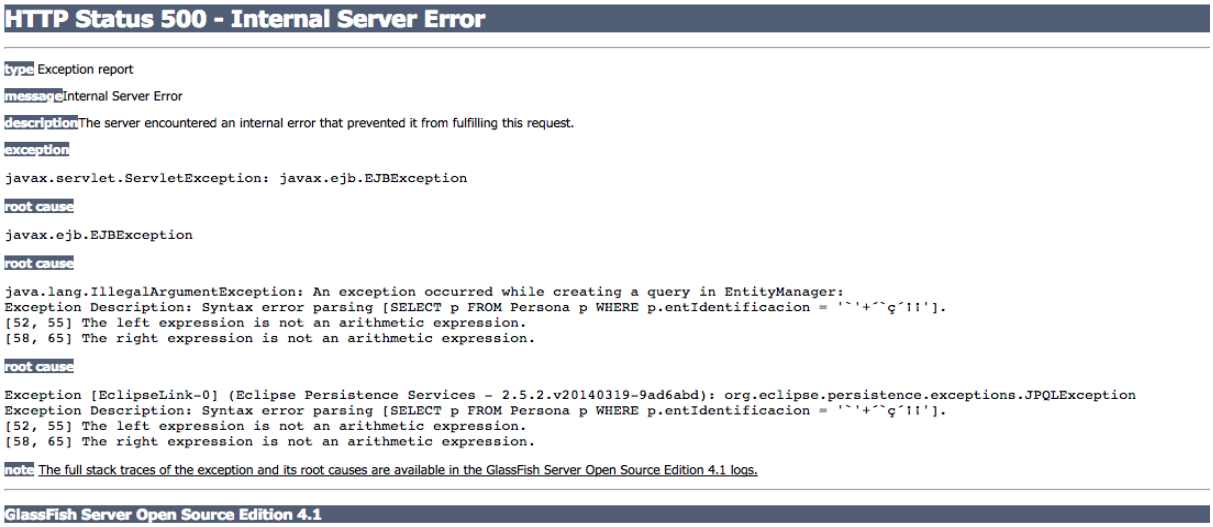


Figura 20: Control de respuestas HTTP

Elaboración. El Autor.

- **Parámetros como parte de la URI:**

Se recurrirá a la especificación de parámetros dentro de la URI que se referirá a un recurso específico, es importante establecer está metodología ya que representa una métrica de calidad en el diseño e implementación de servicios web RESTful.

Es importante especificar que una URI se diferencia de una URL, por qué una URI contempla la identificación de un recurso único independiente del formato; mientras que la URL permite localizar un conjunto de recursos o una ubicación en particular. Según (Marqués, 2013) la URL se compone de:

- *{protocolo}://{dominio/host}/{ruta del recurso}/?{consulta de filtrado}.*

REST reconoce una única forma correcta de construir una URI. Esta viene dada por la forma de enviar parámetros a través de esta, además debe ser un identificador corto y fácil de interpretar; es decir, los parámetros no deben ser enviados de forma tradicional colocando una variable que almacene el parámetros, sino el parámetro debe formar parte de la URI tal y como se especifica en la URI RESTful de la tabla 16.

Tabla 16: URI RESTful vs. URI No-RESTful

URI RESTful	URI No-RESTful
http://utpl.edu.ec/rest/persona/{cedula}	http://utpl.edu.ec/rest/persona?q=cedula

Elaboración. El Autor.

2.5.3. Nivel III: Servidor.

Para este último nivel se aplicarán las siguientes recomendaciones OWASP:

- **Protocolo de transporte:**

OWASP recomienda usar un protocolo de transporte seguro en aplicaciones Web, y de manera especial en aplicaciones que se basan en REST, por lo cual se usará HTTPS como protocolo de transporte. Además es importante especificar que únicamente HTTPS se reconoce como método de seguridad de software REST.

HTTPS es un protocolo seguro basado en HTTP, utiliza un cifrado basado en SSL (Capa de conexión segura) / TLS (Seguridad en la capa de transporte, en sus siglas en español) para crear un canal cifrado por dónde se transmiten datos críticos, de este modo se prevé la intromisión de un atacante a la capa de transporte mostrándole únicamente un flujo de datos cifrado imposible de interpretar.

- **Seguridad Lógica:**

JAAS (Servicio de Autenticación y Autorización Java, en español) será el control lógico de seguridad, Según (ORACLE, s.f.) JAAS proporciona el razonamiento necesario para el acceso de los Clientes a los datos del servicio web.

Tanto la lógica de autenticación como de autorización es un factor que maneja directamente el servidor de aplicaciones por tanto solo se procederá a configurar la seguridad lógica de los servicios web, a nivel de servidor y aplicación.

Finalmente se detallará un cuadro resumen del modelo planteado para el aseguramiento y calidad del software a codificar en la tabla 17.

Tabla 17: Técnicas de seguridad a seleccionadas.

Nivel	Técnica
Base de datos	Procedimientos almacenados
Codificación	Consultas parametrizadas.
	Método de limpieza de caracteres especiales.
	Control de códigos de respuestas HTTP.
	Paso de parámetros como parte de la URI.
Servidor	Autenticación
	HTTPS

Elaboración. El Autor.

CAPÍTULO III

IMPLEMENTACIÓN DE LA SOLUCIÓN.

En este capítulo se presenta el proceso de implementación de la aplicación web REST. El desarrollo del aplicativo se basa en diseños previos expuestos en el capítulo anterior; inicialmente se implementó el repositorio de datos donde se encuentra la información que se expondrá por medio de los servicios web, donde además de realizar la implementación física de la base de datos se construyó los procedimientos almacenados necesarios para extraer los datos necesarios a exponer como recursos web. Seguidamente se procedió a desarrollar el código fuente en base a la arquitectura prevista en la sección 2.1., además se aplicó los mecanismos de seguridad seleccionados para minimizar los riesgos de ataques y afectar la seguridad del aplicativo. Finalmente se especifica las configuraciones a nivel de servidor para que la aplicación REST se encuentre disponible y segura.

3.1. Implementación

La implementación se especifica de acuerdo al modelo arquitectónico planteado y las tecnologías seleccionadas para la construcción del aplicativo REST. Para detallar el desarrollo y despliegue de los servicios web RESTful, se consideraron tres niveles de implementación.

- Datos.
- Codificación.
- Despliegue (Servidor).

3.1.1. Nivel I: Datos.

Inicialmente se construyó una base de datos de acuerdo a la porción del modelo de datos expuesto en la sección 2.3; se consiguieron once tablas en las cuales se almacenó la información extraída del repositorio de datos de la UTPL, buscando dar solución al prototipo planteado para su implementación del proyecto detallado en el capítulo dos.

Posterior a la implementación física de base de datos se definió la información a exponer a través de los servicios web; a partir de entonces se codificaron los procedimientos almacenados necesarios para cumplir con la exposición y aseguramiento de la información que exige las funcionalidades del software REST.

Los procedimientos almacenados se detallan en la tabla 18, se encuentran detallados de acuerdo a los identificadores diseñados, Sección 2.2.

Tabla 18: Procedimientos almacenados según Identificadores RESTful.

Procedimiento Almacenado	Parámetro	Identificador RESTful
listar_personas()	No requiere parámetro	https://host/pft/ws/persona
listar_persona_por_cedula (cedula)	Cédula	https://host/pft/ws/persona/{cedula}
listar_persona_con_proyecto_por_cedula(cedula)	Cédula	https://host/pft/ws/persona/{cedula}/proyecto
listar_persona_por_rol_cedula(rol, cedula)	Rol, Cédula	https://host/pft/ws/persona/{rol}/{cedula}
listar_programas()	No requiere parámetro	https://host/pft/ws/programa
listar_programa_por_modalidad_nivel(modalidad, nivelacademico)	Modalidad, Nivel académico	https://host/pft/ws/programa/{modalidad}/{nivelacademico}
listar_proyecto_por_estado(estado)	Estado	https://host/pft/ws/proyecto/{estado}
listar_proyectos()	No requiere parámetro	https://host/pft/ws/proyecto
listar_programas_por_nombre(nombre)	Nombre de programa	https://host/pft/ws/programa/buscar/{nombre programa}
listar_proyecto_por_nombre(nombre)	Nombre de proyecto	https://host/pft/ws/proyecto/buscar/{nombreproyecto}

Elaboración. El Autor.

Seguidamente se realizó la conexión lógica entre el software y base de datos; dando como resultado un enlace activo para la codificación de los servicios web. En la figura 21 se aprecia dicha conexión y algunos de los procedimientos almacenados alojados en la base de datos.

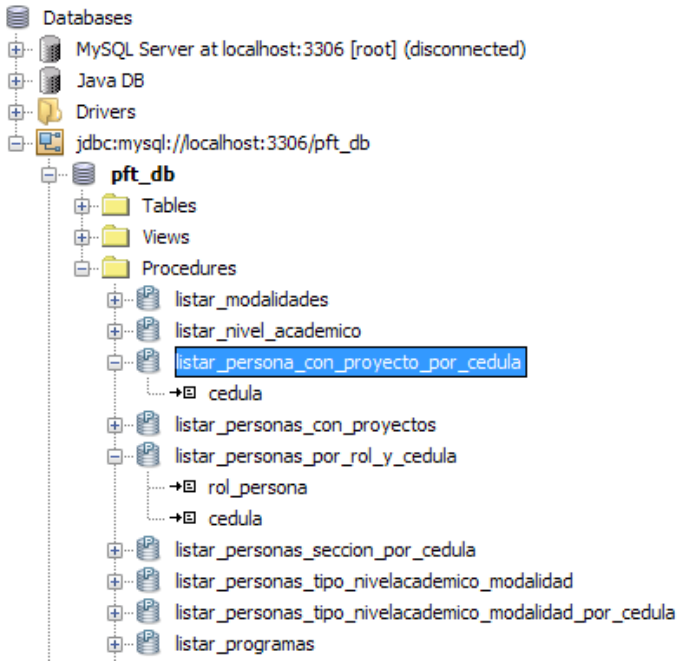


Figura 21: Conexión software y base de datos.

Elaboración. El Autor.

En síntesis, el desarrollo a este nivel se enfocó como base de donde se nacen los recursos REST; en base a ello se siguieron las recomendaciones de seguridad estipuladas en la sección 2.5.1, y recurriendo el motor de base de datos seleccionado en la sección 2.1.4. En particular es vital diferenciar este nivel frente al resto ya que sin datos no se podrían construir los servicios web, para la construcción de servicios web RESTful este es el primer punto a tomar en cuenta; a partir de los datos se define los identificadores y se codifica la solución.

3.1.2. Nivel II: Codificación.

Primeramente se codificó el software usando las herramientas expuestas en la sección 2.4.1. De acuerdo al diseño arquitectónico estipulado en la sección 2.1, internamente se dividió cada componente diseñado en paquetes de programación, los mismos que corresponden a cada componente del software REST; estos consiguen interactúan entre sí para cumplir con las funcionalidades de software pretendidas tal y como se expone en la figura 22.

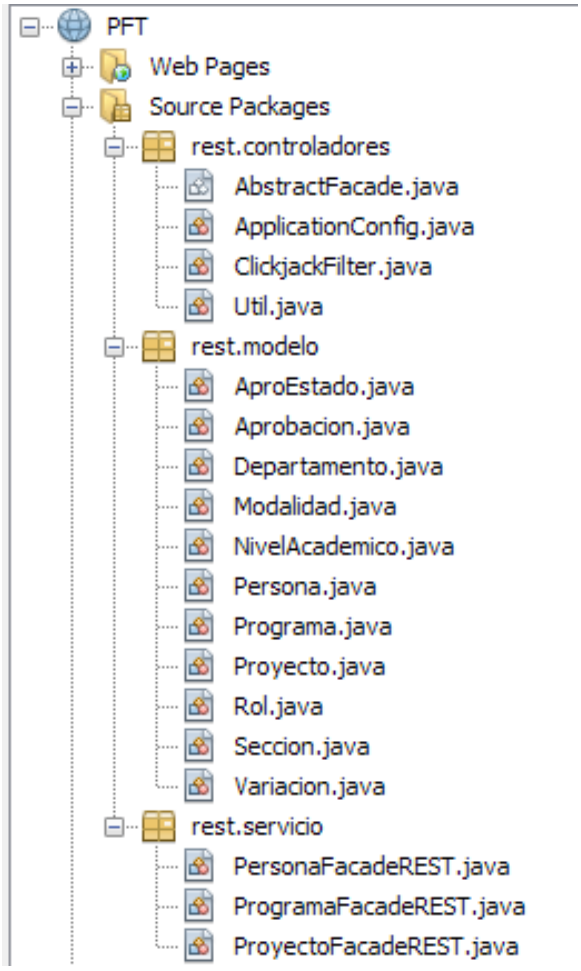


Figura 22: Estructura de codificación de prototipo de software.

Elaboración. El Autor.

Seguidamente se especifica las implicaciones que conllevó la construcción de cada componente diseñado.

3.1.2.1. Componente I: Persistencia.

Definido en la codificación como *rest.modelo,* este paquete de programación contiene mapeada la base de datos; es decir se construyeron clases independientes de programación por cada tabla a nivel de base de datos, las clases codificadas se relacionan de la misma forma que en la base de datos relacional. Este componente se encarga de negociar, solicitar e insertar datos; asimismo de interactuar directamente con las configuraciones de persistencia desplegada mediante JPA reflejadas en la figura 23.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 Resource Definitions//EN" "http://glassfish.org/
<resources>
    <jdbc-connection-pool allow-non-component-callers="false" associate-with-thread="false" connection-creation-retry-attempts="0"
        <property name="serverName" value="localhost"/>
        <property name="portNumber" value="3306"/>
        <property name="databaseName" value="pft_db"/>
        <property name="User" value="root"/>
        <property name="Password" value=""/>
        <property name="URL" value="jdbc:mysql://localhost:3306/pft_db"/>
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="pft_db" object-type="user" pool-name="mysql_pft_db_rootPool"/>
</resources>
```

Figura 23. Configuración de persistencia.

Elaboración. El Autor.

Las mencionadas configuraciones permiten establecer una conexión directamente a la base de datos empleada.

3.1.2.2. Componente II: Negocio.

El componente de negocio se ve reflejado en el paquete de denominado *rest.controladores*, donde se escribieron cuatro clases de programación que contienen la lógica necesaria para cumplir con las funcionalidades de los servicios web REST.

- **AbstractFacade:** Esta clase contiene la lógica para obtener los recursos del repositorio de datos; es la fachada del patrón de diseño Facade. Esta clase posee las funcionalidades que resolverán las invocaciones de cada identificador previamente definido en el punto 2.1.2. En la figura 24, se presenta una muestra de la codificación desarrollada.

```java
    public List<T> listarModalidades() {
        Query q = getEntityManager().createNamedQuery("Modalidad.ListarModalidades");
        return q.getResultList();
    }

    public List<T> listarNivelAcademico() {
        Query q = getEntityManager().createNamedQuery("NivelAcademico.ListarNivelAcademico");
        return q.getResultList();
    }

    public List<T> listarPersonaProyectoPorCedula(Object cedula) {
        String parametro = u.limpiarXSS(cedula.toString());
        Query q = getEntityManager().createNamedQuery("Persona.ListarPersonaProyectoPorCedula");
        q.setParameter(PARAMUNO, parametro);
        return q.getResultList();
    }
```

Figura 24: Codificación | Clase AbstractFacade

Elaboración. El Autor.

- **ApplicationConfig:** en esta se declaran las clases que responderán a la invocación de los servicios web, además se especifica las clases que harán uso de la clase AbstractFacade la misma que se expone en la figura 25.

```
20      private void addRestResourceClasses(Set<Class<?>> resources) {
21          resources.add(rest.servicio.PersonaFacadeREST.class);
22          resources.add(rest.servicio.ProgramaFacadeREST.class);
23          resources.add(rest.servicio.ProyectoFacadeREST.class);
24
25      }
```

Figura 25: Codificación | Clase ApplicationConfig

Elaboración. El Autor.

- **ClickjackFilter:** esta clase contiene la codificación correspondiente al método de evasión de vulnerabilidades de tipo ClickJacking, además se configuraron las cabeceras de respuesta seguras recomendadas por OWASP, dicha codificación se presenta en la figura 26.

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
    HttpServletResponse res = (HttpServletResponse) response;
    res.addHeader("Pragma", "no-cache");
    res.addHeader("Cache-Control", "no-cache, no store");
    res.addHeader("X-FRAME-OPTIONS", MODE);
    res.addHeader("Strict-Transport-Security", "max-age=16070400; includeSubDomains");
    res.addHeader("X-XSS-Protection", XSSPROTECTION);
    res.addHeader("X-Content-Type-Options", "nosniff");
    res.addHeader("Content-Security-Policy", "default-src 'self'");
    chain.doFilter(request, response);
}
```

Figura 26: Codificación | Clase ClickjackFilter

Elaboración. El Autor.

- **Útil:** En esta clase contiene el método para limpieza de caracteres especiales de los parámetros de entrada ingresados por el cliente. A continuación en la figura 27 se expone el código fuente desarrollado para cumplir con el objetivo de dicho método.

```
8  public class Util {
9      public String limpiarXSS(String value) {
10         String param = "";
11         param = value.replaceAll("<", "& lt;").replaceAll(">", "& gt;");
12         param = value.replaceAll("\\(", "& #40;").replaceAll("\\)", "& #41;");
13         param = value.replaceAll("'", "& #39;");
14         param = value.replaceAll("eval\\((.*)\\)", "");
15         param = value.replaceAll("[\\\"\\\']+[\\s]*javascript:(.*)[\\\"\\\']+", "\"\"");
16         param = value.replaceAll("script", "");
17         param = value.replaceAll("BODY", "");
18         param = value.replaceAll("onload", "");
19         param = value.replaceAll("alert", "");
20         param = value.replaceAll("%", "");
21         param = value.replaceAll("-", "");
22         return param;
23     }
```

Figura 27: Codificación | Clase Útil

Elaboración. El Autor.

3.1.2.3. Componente III: Servicios.

Este paquete corresponde al componente nombrado de forma similar, contiene las clases para implementar los servicios web. Estas clases son parte del patrón de diseño implementado, invocan funcionalidades de la fachada AbstractFacade, las clases escritas son:

- PersonaFacadeREST
- ProgramaFacadeREST
- ProyectoFacadeREST

En la figura 28 se observa la invocación de métodos necesarios para resolver las peticiones de los clientes, la porción de código expuesta muestra dos métodos propiedad de la clase PersonaFacadeREST.

```
28     @GET
29     @Path("{cedula}/proyecto2")
30     @Produces({"application/json"+ ";charset=utf-8"})
31     public List<Persona> buscarPersonaPorCedula(@PathParam("cedula") String cedula) {
32         return super.buscar(cedula);
33     }
34
35     @GET
36     @Path("{cedula}/proyecto")
37     @Produces({"application/json"+ ";charset=utf-8"})
38     public List<Persona> listarPersonaConProyectoPorCedula(@PathParam("cedula") String cedula) {
39         return super.listarPersonaProyectoPorCedula(cedula);
40     }
```

Figura 28: Codificación | Clase PersonaFacadeREST.

Elaboración. El Autor.

Para finalizar este nivel hay que mencionar además, que se configuraron algunos archivos que responden al aseguramiento del aplicativo a nivel de servidor:

- Web.xml
- Glassfish-web.xml

Los mismos que se profundizarán en el siguiente nivel de implementación.

3.1.3. Nivel III: Servidor.

El servidor es un factor importante para el despliegue de la aplicación, es el encargado de contener, y, gestionar las distintas funcionalidades y acceso de un sistema de software. Por tanto se ha dado tanta importancia al servidor de aplicaciones en el presente desarrollo.

Previo al despliegue y conformación de seguridad, se inició configurando el control de seguridad lógico, ello implicó la especificación de roles y usuarios con acceso a los servicios web. Seguidamente se codificaron los archivos de configuración correspondientes al aseguramiento de la aplicación a nivel de servidor, para obtener un enlace entre software y servidor.

En el archivo web.xml, se configuró lo siguiente:

- **HTTP Only,** para proteger la aplicación de la lectura y escritura de scripts por parte de los atacantes; es decir, que no sea accesible por código script. Para ello se escribió en el archivo:
  - ```
    <session-config>
            <cookie-config>
                <http-only>true</http-only>
            </cookie-config>
     </session-config>
    ```
- **Clickjack**, se aplicó como medida de fortalecimiento para evitar ataques de tipo ClickJacking, este ataque comúnmente pretende robar la identidad de un usuario con el fin de obtener el control de su máquina al hacer clic en una página web, su cadena de configuración corresponde a un filtro de seguridad implementado en la Clase ClickjackFilter, dentro del paquete de controladores, donde su lógica deniega este tipo de ataque:
  - ```
    <filter>
           <filter-name>ClickjackFilterDeny</filter-name>
            <filter-class>
                rest.controladores.ClickjackFilter
            </filter-class>
        <init-param>
            <param-name>mode</param-name>
            <param-value>DENY</param-value>
    ```

```
        </init-param>
    </filter>
    <filter>
        <filter-name>ClickjackFilterSameOrigin</filter-name>
        <filter-class>
             rest.controladores.ClickjackFilter
         </filter-class>
        <init-param>
            <param-name>mode</param-name>
            <param-value>SAMEORIGIN</param-value>
        </init-param>
    </filter>
```

- **HTTPS y Autenticación**, se configuró el protocolo de transporte seguro de HTTP con el fin de cifrar las conexiones y que no sean interceptadas por terceros; además de la lógica de autenticación según la siguiente codificación:
    -
```
<security-constraint>
        <display-name>Constraint1</display-name>
        <web-resource-collection>
            <web-resource-name>ssl</web-resource-name>
            <description/>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <description/>
            <role-name>admin</role-name>
        </auth-constraint>
        <user-data-constraint>
            <description/>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>loginPFTRest</realm-name>
    </login-config>
    <security-role>
```

```
        <description/>
        <role-name>admin</role-name>
    </security-role>
```

Dichas configuraciones dieron como resultado la implementación de un protocolo de transporte seguro HTTPS para cifrar conexiones, además de su control de seguridad lógica, limitando así el acceso a usuarios no permitidos, así como se expone en la figura 29, donde se aprecia el resultado final de la aplicación REST más sus mecanismos de seguridad.

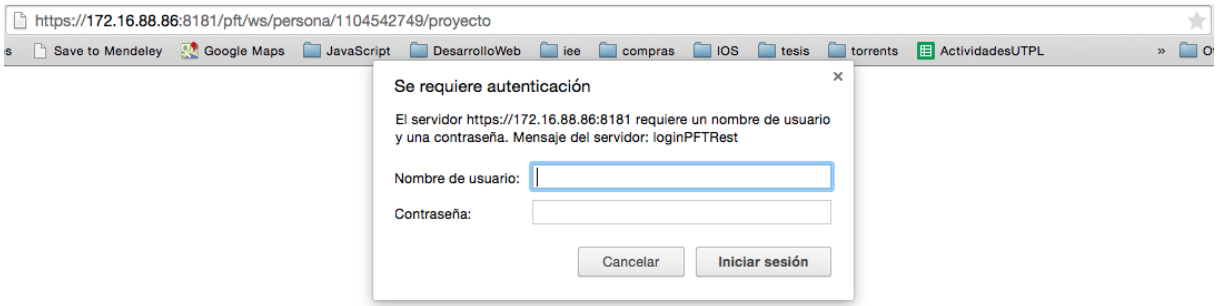


Figura 29: Implementación | Autenticación de JAAS.
Elaboración. El Autor.

CAPÍTULO IV

PRUEBAS Y RESULTADOS.

Este capítulo contiene los resultados de las validaciones efectuadas con el fin de garantizar la seguridad del aplicativo REST. Para esto se presenta los resultados obtenidos de las pruebas realizadas a través de herramientas especializadas en tres etapas de evaluación: arquitectura de software, calidad de código fuente, y seguridad del aplicativo REST.

En la primera etapa se evidencia los resultados de implementación entre código fuente y la arquitectura de software planteada, en la segunda etapa se evalúa la calidad del código fuente, y finalmente en la tercera etapa correspondiente a la seguridad se presentan los resultados objetivos del proceso de estudio, en el que se logra minimizar las vulnerabilidades seleccionadas y se obtiene resultados capaces de garantizar la seguridad en las aplicaciones web basadas en REST.

4.1. Etapa I: Diseño Arquitectónico.

En esta etapa temprana de desarrollo de software se describirán las pruebas que se realizaron para validar el modelo y su interacción entre componentes, a través de las diferentes herramientas investigadas.

Las herramientas aplicadas durante esta etapa de validación son:

1. Structural Analisys for Java.
2. Sonargraph Architect.

Las herramientas destacadas disponen de licenciamiento gratuito; los dos medios de validación seleccionados presentan grandes diferencias de funcionalidad, por tal motivo las pruebas realizadas se detallarán a partir de sus funcionalidades.

4.1.1. Structural Analisys for Java.

Este artefacto de software es una solución desarrollada por IBM, permite analizar las dependencias estructurales de soluciones escritas en lenguaje JAVA con la finalidad de analizar su diseño en todos los componentes escritos. Esta herramienta tiene la característica especial para medir la estabilidad del software desarrollado; expone que los resultados superiores al 90% de estabilidad disponen de altos estándares de desarrollo de software proporcionando alta estabilidad en su etapa de despliegue.

El análisis que ejecuta esta herramienta permite validar la correcta aplicación del modelo planteado frente al software codificado. El análisis empleado detalla una diagramación pormenorizada del código escrito permitiendo evaluar de manera profunda la aplicación de software.

Se obtuvieron los siguientes resultados a partir de las funcionalidades prestadas por Structural Analisys for JAVA.

- **Arquitectura de software.**

Los resultados obtenidos en esta validación se reflejan en la figura 30.

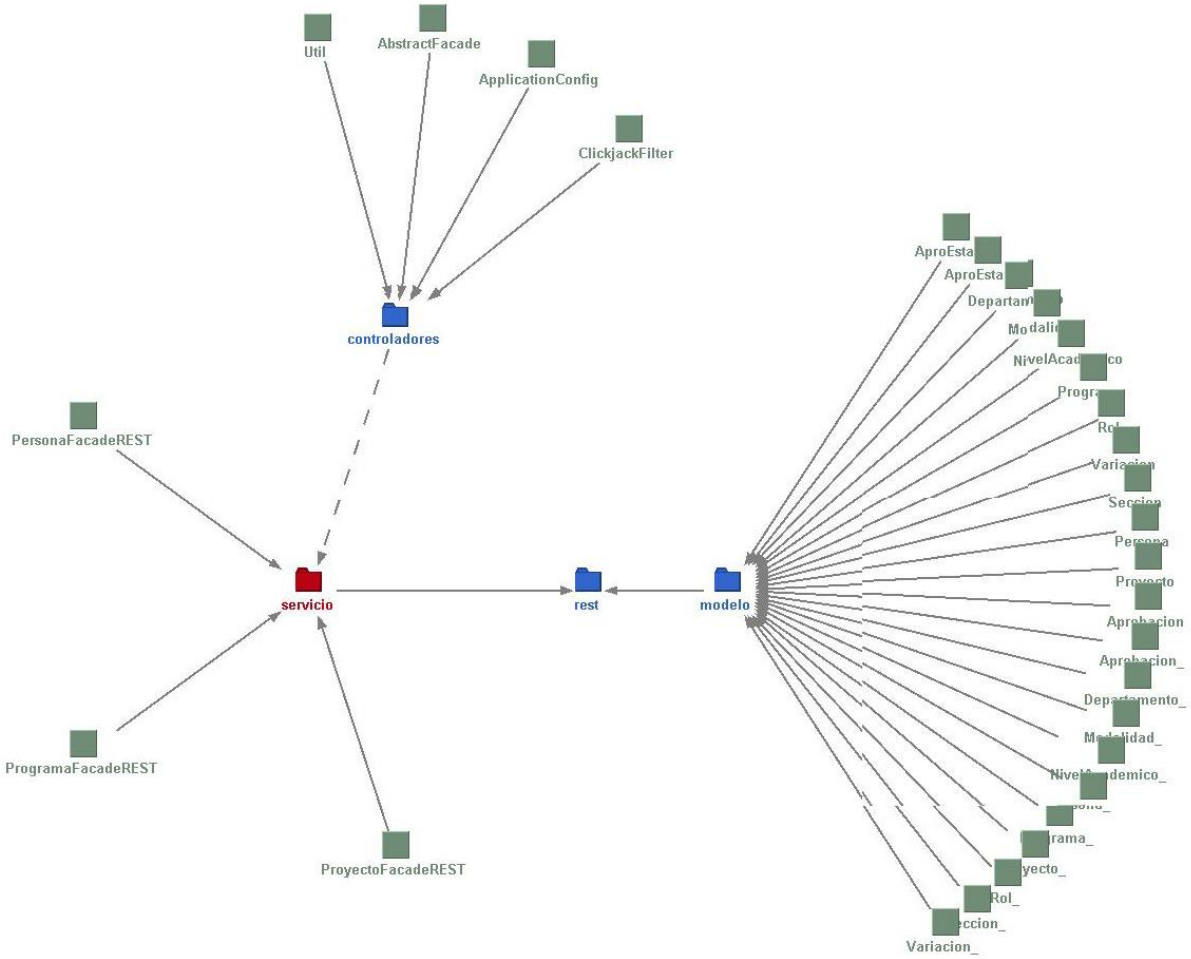


Figura 30: Pruebas | Diseño arquitectónico.

Elaboración. El Autor

Donde la validación resultante evidencia un 100% de similitud entre la arquitectura planteada y el software desarrollado.

- **Componentes.**

Los resultados alcanzados se exponen en la figura 31.

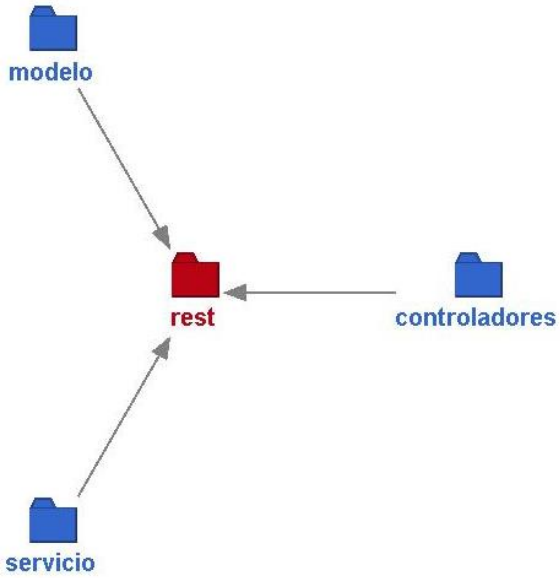


Figura 31: Pruebas: componentes de software.
Elaboración. El Autor.

Este resultado proporciona un enfoque directo, donde se puede validar la creación de los componentes diseñados de acuerdo al modelo planteado.

- **Patrón de diseño.**

En base al análisis realizado, la implementación del patrón Facade se cumplió de forma exitosa, según los resultados obtenidos en la figura 32. La herramienta diagramó la codificación efectuada, dando como resultado una perspectiva similar a la que propone el patrón de diseño en cuestión

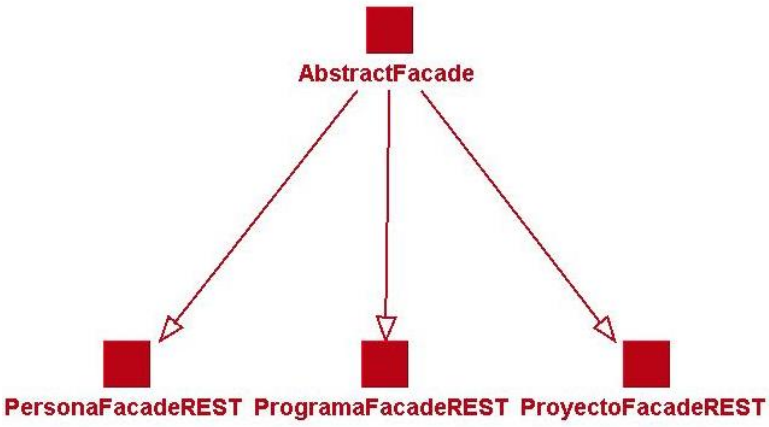
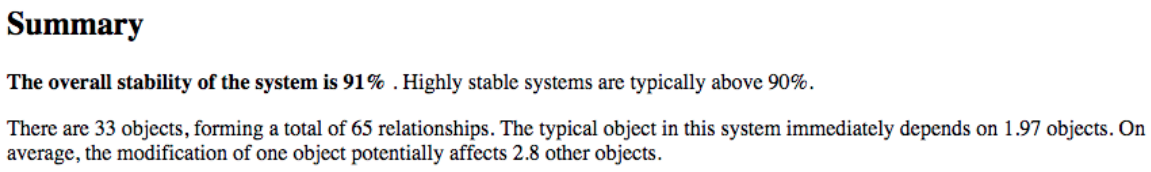


Figura 32: Pruebas | Patrón de diseño.
Elaboración. El Autor.

- **Estabilidad.**

Esta es la característica más importante que provee esta herramienta de pruebas. A partir de los resultados proporcionados por la herramienta, se obtuvo el resultado detallado en la figura 33.



Summary

The overall stability of the system is 91% . Highly stable systems are typically above 90%.

There are 33 objects, forming a total of 65 relationships. The typical object in this system immediately depends on 1.97 objects. On average, the modification of one object potentially affects 2.8 other objects.

Figura 33: Pruebas | Estabilidad de software.
Elaboración. El Autor.

Donde se aprecia que el resultado alcanzado, en base al modelo planteado determinó el 91% de estabilidad, calificando así a nuestro aplicativo como un software altamente estable.

Para finalizar, las pruebas realizadas con este elemento de software; dieron como resultado que la aplicación tiene un alto grado de estabilidad, y cumple con el modelo de arquitectura planteado en su etapa de diseño. Por tanto en esta fase de pruebas, se puede concluir que los resultados son satisfactorios tomando como referencia el diseño propuesto.

4.1.2. Sonargraph Architect.

Sonargraph Architect es una herramienta comercial escrita en Java; ofrece varios tipos de licenciamiento, entre esos existe una versión gratuita para estudios.

(hello2morrow, 2015), explica que “es una herramienta de análisis estático que permite definir el modelo de arquitectura de software que puede ser verificada y ejecutada en modo autónomo o con plugins IDE de Eclipse o IntelliJ”.

Esta herramienta aporta al usuario la capacidad de entender la estructura del sistema desarrollado, muestra la interacción que existe entre componentes, clases, y otros artefactos de programación que son parte del desarrollo de software.

En cuanto al análisis de la aplicación REST desarrollada es importante especificar que se requirió alojar el código desarrollado, en el IDE Eclipse ya que Sonargraph Architect únicamente ejecuta análisis a sistemas desarrollados en el IDE de programación antes mencionado.

Es considerable explicar que los resultados obtenidos en este análisis no podrían ser exactos, en su apartado de resumen.

- **Estructura.**

En la figura 34 se expone el resultado de esta validación.

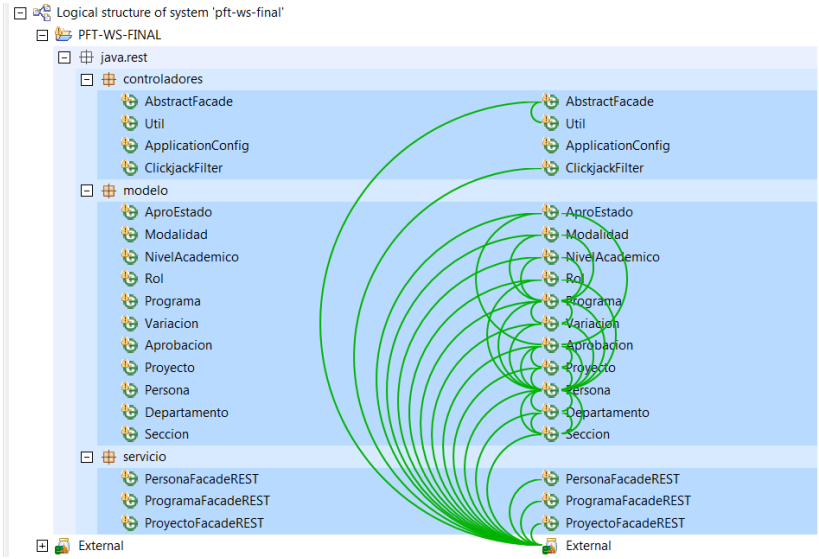


Figura 34: Pruebas | Estructura de la aplicación REST.

Elaboración. El Autor.

Este resultado demuestra la estructura de paquetes/componentes y sus clases. Al igual que la herramienta anterior se observa que el desarrollo de la aplicación produce un estado de igualdad del software escrito con el modelo planteado.

- **Paquetes/Componentes.**

En la figura 35 se exponen los resultados de este apartado.

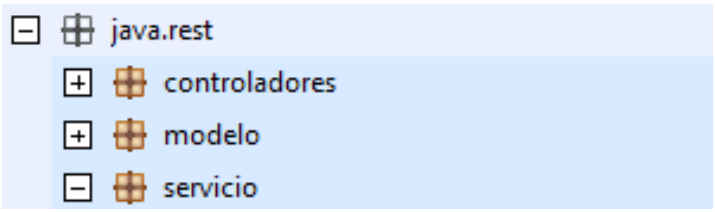


Figura 35: Pruebas | Componentes de software REST.

Elaboración. El Autor.

Se observa que los tres componentes implementados actúan como componentes individuales, cada uno posee su cometido conforme al diseño ideado. La implementación desarrollada demuestra una similitud con la arquitectura de software en capas.

- **Clases**

Sonargraph, permite evaluar la interacción que existe entre componentes desarrollados; por ello, una vez determinados que los resultados dieron como consecuencia la correcta

aplicación del modelo planteado; se mostrarán los resultados de las interacciones que existen entre componentes.

- **Interacción clase de modelo:**

Por objeto de evaluación del diseño de software se inició seleccionando la clase Persona del componente modelo. En la figura 36, se observa las distintas vías de interacción que posee la clase antes destacada. El resultado de esta validación expone que la clase en cuestión interfiere con clases del mismo componente de software; sucede porque el modelo mapea la base de datos en clases de programación, y sus interacciones son resultantes de las relaciones de datos que existen.

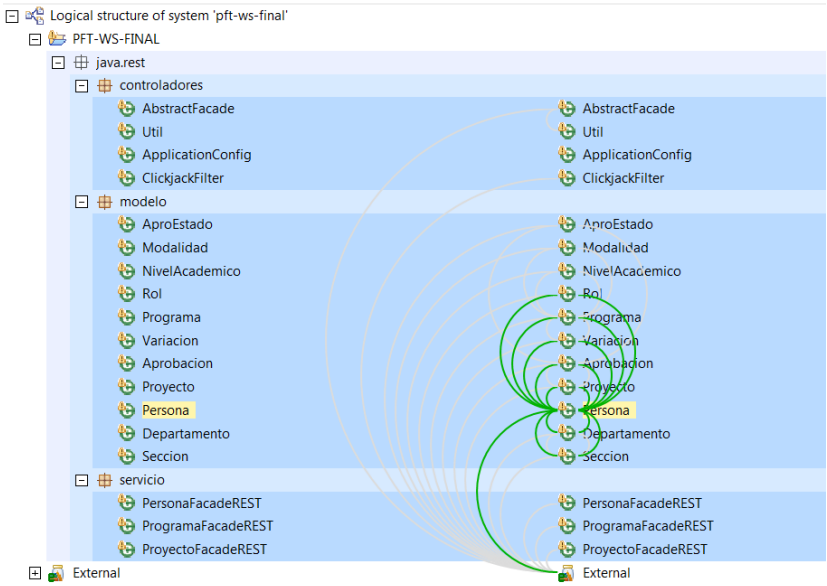


Figura 36: Pruebas | Interacción entre clases del componente Modelo.

Elaboración. El Autor.

- **Interacción de clases de servicios**

Las clases correspondientes a los servicios interactúan directamente con métodos externos y con la clase correspondiente al servicio invocado tal y como se observa en la figura 37. Como consecuencia de los resultados; analizamos a la clase PersonaFacadeREST; donde se puede observar que esta clase interactúa directamente con las clases AbstractFacade y Persona; AbstractFacade es la fachada el patrón de diseño seleccionado, y, Persona es la clase que obtiene los datos de la persistencia, pues aquí se encuentra mapeada la tabla Persona de la base de datos junto con sus relaciones. Además responde a los métodos EntityManager correspondiente a la persistencia de datos; y, con el método GET que es quién expresa la acción de consulta de los servicios web.

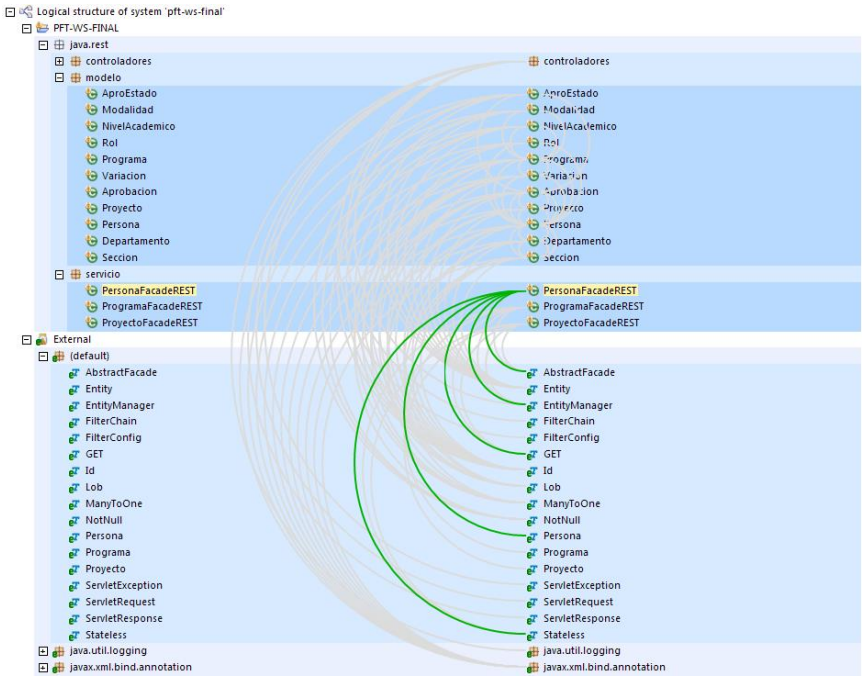


Figura 37: Pruebas | interacción entre clases del componente de Servicios.

Elaboración. El Autor.

- **Interacción de GET con clases de programación**

El método GET expresa la acción para invocar a los servicios web. Según los resultados obtenidos en la figura 38, se observa que este método es externo al software desarrollado; accede directamente al componente de servicios; es decir, el servicio es invocado por un usuario y este llama a la clase que corresponda del componente de servicios, para que este realice la ejecuciones pertinentes con el fin de dar respuesta a las peticiones del cliente.

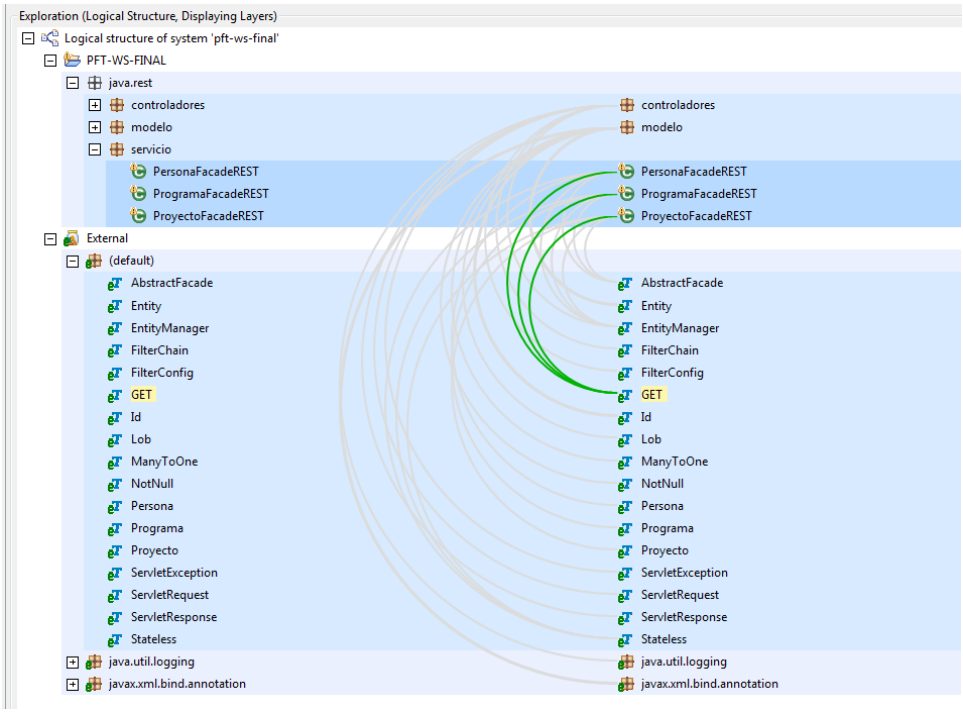


Figura 38: Pruebas | Interacción de método HTTP GET con el software REST.

Elaboración. El Autor.

- **Resumen**

Finalmente se presenta la figura 39, donde se exponen los resultados generales del análisis del diseño planteado. Como se mencionó al inicio de esta sección los resultados obtenidos en esta herramienta de pruebas podrían no ser íntegros por la incompatibilidad que existe entre IDE's de programación.

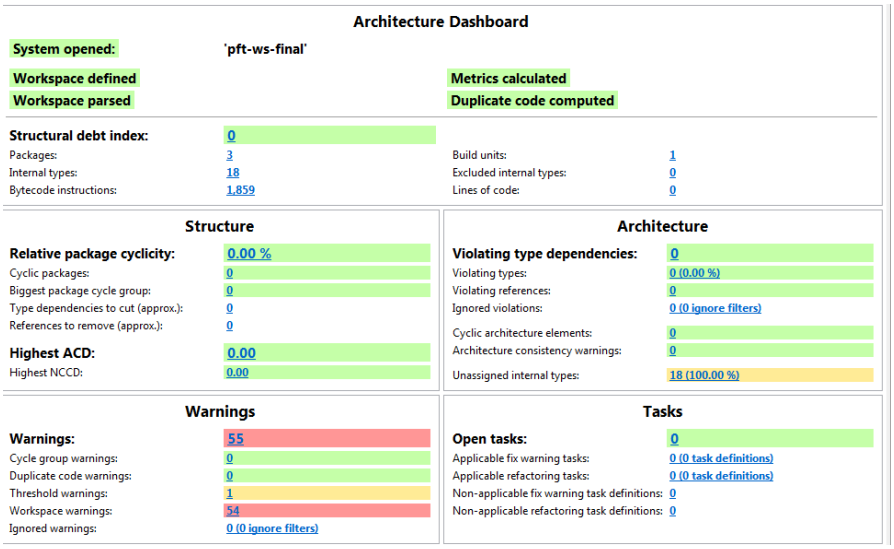


Figura 39: Pruebas | Resumen de análisis Sonargraph Architect.

Elaboración. El Autor.

4.2. Etapa II: Codificación.

Esta etapa de desarrollo de los servicios web, presenta mayor relevancia en las pruebas realizadas; esta etapa demuestra ser fundamental debido a que lógicamente inicia el aseguramiento de software.

Las validaciones efectuadas en esta fase de desarrollo corresponden al aseguramiento y evaluación de la calidad del software REST a través del código fuente seleccionada; estas pruebas se hicieron desde algunos ámbitos; entre ellos, buenas prácticas de programación ensalzadas por OWASP.

La herramienta seleccionada para realizar esta validación fue:

- SonarQube.

SonarQube, es una herramienta muy completa para el análisis de código fuente. Es software libre, en esta herramienta se puede evaluar el código en todos sus aspectos de codificación; presenta un análisis muy completo del código desarrollado haciendo uso de métricas de calidad específicas configuradas en la herramienta de software.

Además SonarQube brinda una característica importante para medir la calidad del código escrito, esta se basa en SQALE (Evaluación de la Calidad de Software basado en las expectativas de ciclo de vida), esta característica evalúa el código fuente realizando una calificación secuencial a través de cinco valores A, B, C, D y E; donde, A es la mejor calificación y E la menor calificación de esta evaluación de calidad, finalmente esta aptitud permite definir una Deuda Técnica que presenta resultados referente al tiempo que tomaría solucionar los problemas técnicos resultantes.

De este modo, en lo que sigue se presentaran las distintas iteraciones de pruebas realizadas al código fuente de la aplicación REST.

4.2.1. Resultados | Problemas de codificación.

Durante las pruebas realizadas al código fuente de la aplicación se encontraron algunos errores que conciernen, en general a malas prácticas de programación. Estos problemas se corrigieron durante cada iteración.

De forma general, los problemas presentados durante las pruebas de código fuente se listan a continuación:

a) Nombres de los métodos deben cumplir con una convención de nombres.
b) Los nombres de campo deben cumplir con una convención de nombres.
c) Nombres de parámetros variables y métodos locales deben cumplir con una convención de nombres.

d) Bloques Duplicados.

e) Parámetros del método, excepciones capturadas y variables foreach no deben ser reasignados.

f) Nombres constantes deben cumplir con una convención de nombres.

g) System.out y System.err no deben utilizarse como loggers.

h) Evite las líneas comentadas de salida de código.

i) Los loggers deben ser "private static final" y deben compartir una convención de nombres.

j) Métodos no deben estar vacíos.

k) Tipos de comodines genéricos no se deben utilizar en los parámetros de retorno.

Estadísticamente se reflejan en la figura 40; se observa la cantidad de problemas encontrados en relación a las iteraciones de pruebas; se ejecutaron cuatro iteraciones. Los problemas se ven representados de acuerdo a la numeración de los problemas listados previamente.

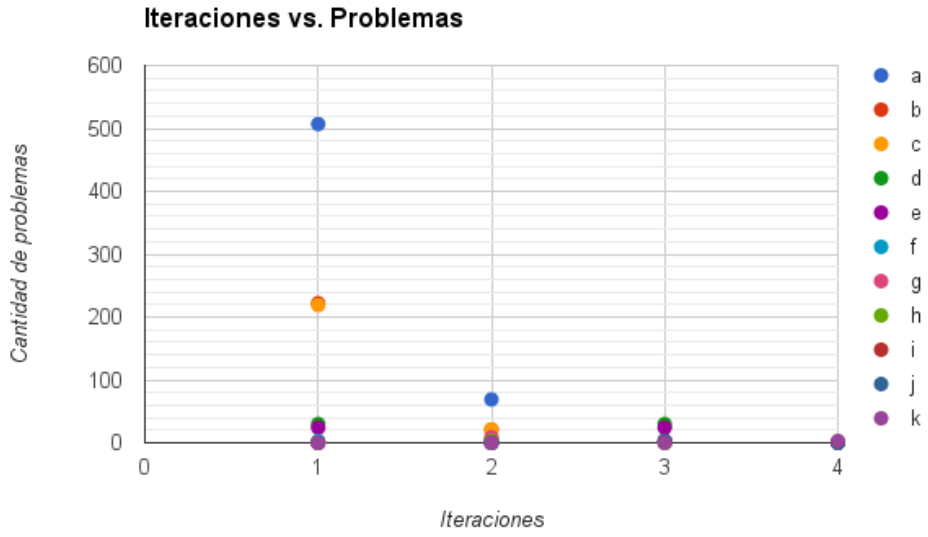


Figura 40: Pruebas de código | Iteraciones vs Problemas.

Elaboración. El Autor.

Cuantificando los problemas encontrados durante cada iteración de pruebas resultante fue la siguiente:

- **Iteración 1:** 1038 problemas.
- **Iteración 2:** 240 problemas.
- **Iteración 3:** 102 problemas.
- **Iteración 4:** 3 problemas.

De las cuales se obtuvo una curva de resolución de problemas positiva, la misma que se representa en la figura 41.

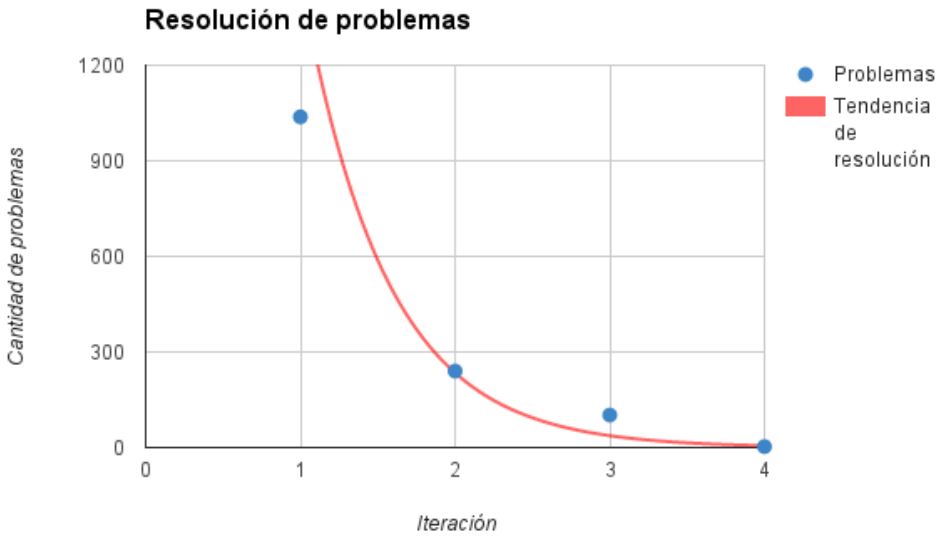


Figura 41: Pruebas de código | Curva de resolución de problemas.
Elaboración. El Autor.

Donde se observa una curva decreciente, el pico máximo se presentó en la iteración número uno, y secuencialmente decrecen en las iteraciones subsiguientes logrando llegar a la última iteración con un porcentaje de problemas de codificación minúsculo.

4.2.2. Iteraciones de codificación.

4.2.2.1. Iteración 1.

En esta primera iteración de pruebas se tuvieron los problemas detallados en la tabla 19. Además de una calificación “B” según el modelo de evaluación de calidad de software SQALE.

Los resultados obtenidos de esta primera iteración son:

- **Líneas de código:** 4857
- **Funciones:** 303
- **Problemas:** 1038
- **Deuda técnica:** 36 días.

Tabla 19: Pruebas de código | Problemas Iteración 1.

Problema	Cantidad
1. Nombres de los métodos deben cumplir con una convención de nombres.	507
2. Los nombres de campo deben cumplir con una convención de nombres.	222
3. Nombres de parámetros variables y métodos locales deben cumplir con una convención de nombres.	219
4. Bloques duplicados	30
5. Parámetros del método, excepciones capturadas y variables foreach no deben ser reasignados.	24
6. Nombres constantes deben cumplir con una convención de nombres.	3

Elaboración. El Autor.

Estadísticamente se exponen dichos resultados en la figura 42. Los errores más frecuentes demuestran la falta de uso de estándares de programación.

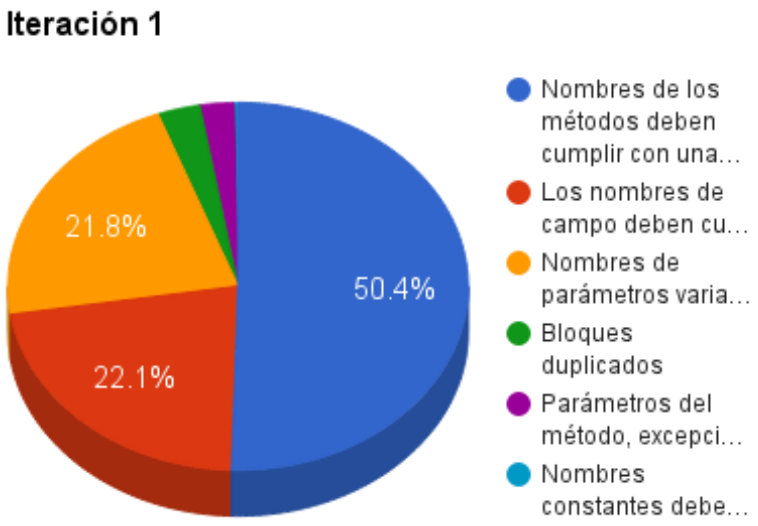


Figura 42: Pruebas de Código | Gráfica de Problemas Iteración 1.

Elaboración. El Autor.

4.2.2.1.1. Resolución de problemas.

En la tabla 20 se expone los diferentes métodos de resolución de problemas encontrados en el código fuente desarrollado.

Tabla 20: Iteración 1 | Resolución de problemas.

Problema	**Error de codificación**	**Resolución**

Nombres de los métodos deben cumplir con una convención de nombres.	public Integer GetInt_id() {
	return int_id;
	}
		public Integer getIntId() {
		return intId;
		}

Los nombres de campo deben cumplir con una convención de nombres.	private Integer int_id;	private Integer intId;

Nombres de parámetros variables y métodos locales deben cumplir con una convención de nombres.	public void setInt_id(Integer int_id) {
	this.int_id = int_id;
	}
		public void setIntId(Integer intId) {
		this.intId = intId;
		}

Bloques duplicados	Existen métodos con nombres similares entre clases de programación	No deben existir métodos con nombres repetidos en todo el código.
Parámetros del método, excepciones capturadas y variables foreach no deben ser reasignados.	value = value.replaceAll("<", "& lt;").replaceAll(">", "& gt;");	paramCorrecto = value.replaceAll("<", "& lt;").replaceAll(">", "& gt;");
Nombres constantes deben cumplir con una convención de nombres.	static final Logger logger = Logger.getLogger("Logs");	private static final Logger LOGGER = Logger.getLogger(NOMBRELOG);

Elaboración. El Autor.

4.2.2.2. Iteración 2.

En esta segunda instancia de pruebas de código fuente se obtuvieron los resultados detallados en la tabla 21.

Durante esta iteración se obtuvo una calificación de “A”, según SQALE. Asimismo se encontraron los datos:

- **Líneas de código:** 4899
- **Funciones:** 306
- **Problemas:** 240
- **Deuda técnica:** 7 días 4 horas.

Tabla 21: Pruebas de código | Problemas Iteración 2.

Problema	Cantidad
1. Nombres de los métodos deben cumplir con una convención de nombres.	69
2. Los nombres de campo deben cumplir con una convención de nombres.	21
3. Nombres de parámetros variables y métodos locales deben cumplir con una convención de nombres.	21
4. Bloques duplicados	6
5. System.out y System.err no deben utilizarse como loggers.	9
6. Evite las líneas comentadas de salida de código.	3

Elaboración. El Autor.

En esta iteración de pruebas, los problemas se vieron reducidos significativamente; la mayoría de problemas se repiten en relación a la primera iteración. La resolución de problemas es similar a la tabla 20, de la iteración número uno. En la figura 43, se presenta una gráfica referente a los resultados de esta segunda iteración de pruebas de código fuente.

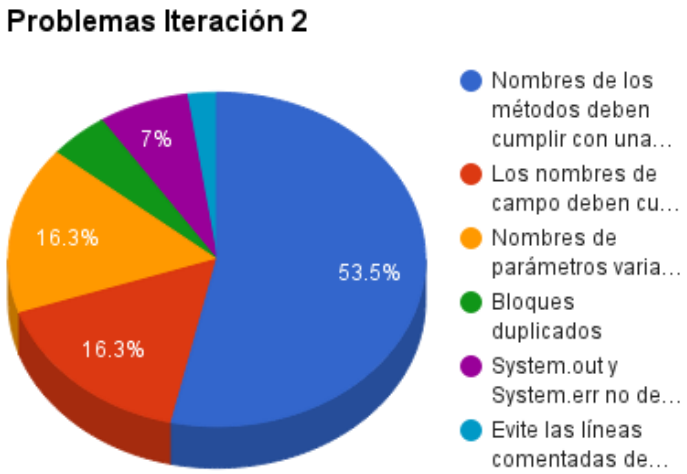


Figura 43: Pruebas de Código | Gráfica de Problemas Iteración 2.

Elaboración. El Autor.

4.2.2.3. Iteración 3.

En una tercera fase de pruebas el resultado propinó los problemas detallados en la tabla 22; similar al análisis anterior, se mantuvo la calificación de calidad de código con una nota de "A", correspondiente a la mayor nota de otorgada a partir de las métricas expuestas por SQALE. Por otra parte el estudio en esta etapa de pruebas dio los siguientes datos:

- **Líneas de código:** 4860
- **Funciones:** 303
- **Problemas:** 102
- **Deuda técnica:** 7 días 4 horas.

Tabla 22: Pruebas de código | Problemas Iteración 3.

Problema	Cantidad
1. Bloques duplicados	30
2. Parámetros del método, excepciones capturadas y variables foreach no deben ser reasignados.	24
3. Los nombres de campo deben cumplir con una convención de nombres.	3
4. Nombres de parámetros variables y métodos locales deben cumplir con una convención de nombres.	3
5. Nombres de constantes deben cumplir una convención de nombres	9
6. Los loggers deber ser "private static final" y deben compartir una convención de nombres.	3
7. Métodos no deben estar vacíos	3

Elaboración. El Autor.

Gráficamente se reflejan los resultados de este análisis en la figura 44.

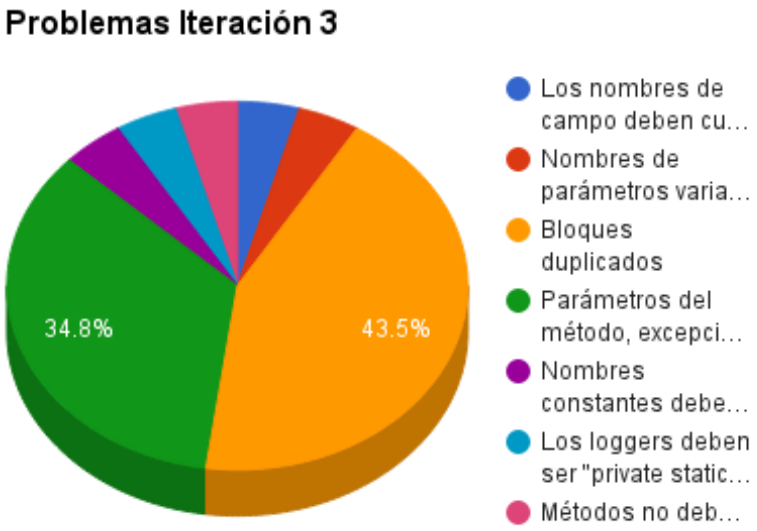


Figura 44: Pruebas de Código | Gráfica de Problemas Iteración 3.

Elaboración. El Autor.

4.2.2.3.1. Resolución de problemas.

Las soluciones ejecutadas se exponen en la tabla 23.

Tabla 23: Resolución de problemas, iteración 3

Problema	**Error de codificación**	**Resolución**
Bloques duplicados	Existen métodos con nombres similares entre clases de programación	No deben existir métodos con nombres repetidos en todo el código.
Parámetros del método, excepciones capturadas y variables foreach no deben ser reasignados.	value = value.replaceAll("<", "& lt;").replaceAll(">", "& gt;");	paramCorrecto = value.replaceAll("<", "& lt;").replaceAll(">", "& gt;");
Los nombres de campo deben cumplir con una convención de nombres.	private Integer int_id;	private Integer intId;

Nombres de parámetros variables y métodos locales deben cumplir con una convención de nombres.	public void setInt_id(Integer int_id) {
	this.int_id = int_id;
	}
		public void setIntId(Integer intId) {
		this.intId = intId;
		}

Los loggers deber ser “private static final” y deben compartir una convención de nombres.	static final Logger logger = Logger.getLogger("Logs");	private static final Logger LOGGER = Logger.getLogger(NOMBRELOG);

Nombres de los métodos deben cumplir con una convención de nombres.	public Integer GetInt_id() {
	return int_id;
	}
		public Integer getIntId() {
		return intId;
		}

Métodos no deben estar vacíos	public void destroy () {
	}
		public void destroy(){
		Logger.info(“sesión destruida”);
		}

Elaboración. El Autor.

4.2.2.4. Iteración 4.

Finalmente en la iteración final, se ven reflejados los resultados de todas las resoluciones, se obtuvieron únicamente tres problemas, los mismos que no se atendieron por qué hacen referencia a una implementación propia de JAX-RS.

En esta iteración final se obtuvo una calificación de calidad “A”; es la máxima calificación de calidad al código fuente desarrollada. Resumidamente se obtuvo:

- **Líneas de código:** 4716
- **Funciones:** 303
- **Problemas:** 3
- **Deuda técnica:** 1 hora.

En la figura 45 se aprecia la evaluación de calidad y la cantidad de problemas excluyentes. En esta segunda etapa de validaciones los resultados fueron satisfactorios de tal manera que se consiguió que el código fuente del aplicativo posea la mayor calidad posible.

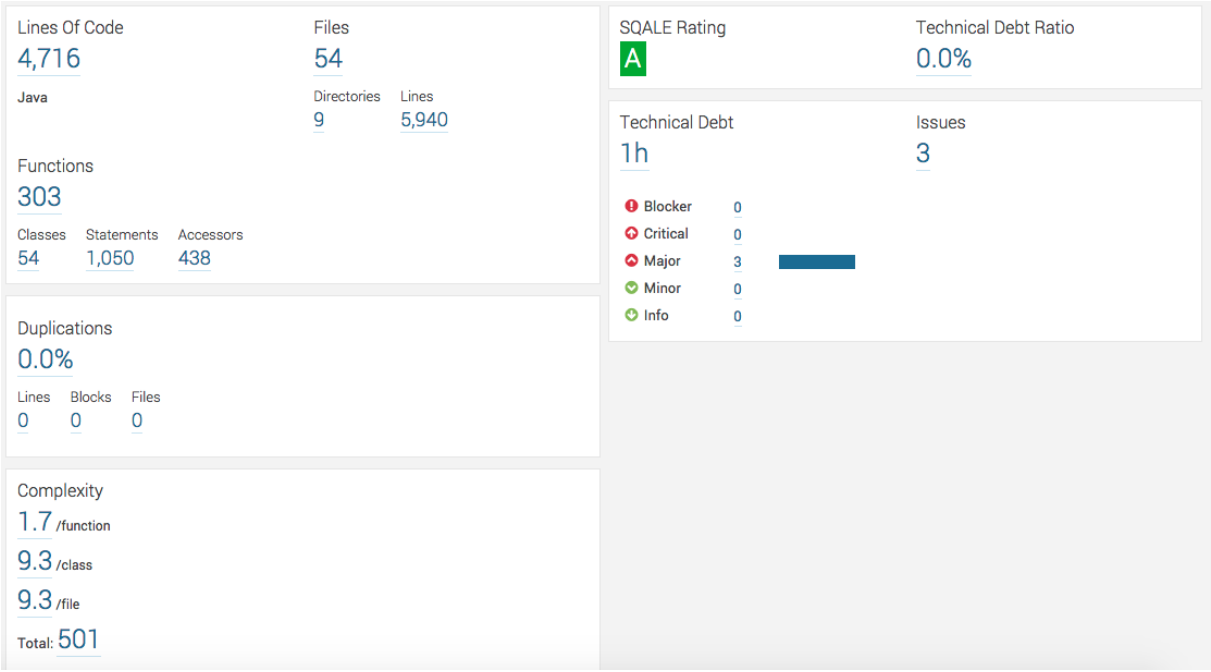


Figura 45: Pruebas de Código | Resumen de resultados de Iteración 4.
Elaboración. El Autor.

4.3. Etapa III: Pruebas de seguridad de servicios web RESTful.

En lo que a la seguridad del software REST respecta, se realizaron las validaciones necesarias con la finalidad de encontrar fallos de seguridad. OWASP explica que la seguridad en los aplicativos web, es un tema muy importante; aquí se expone información con alto grado de criticidad, pues de forma general la información en cualquier ámbito, desde lo empresarial hasta lo educativo es confidencial.

Para evaluar el tema de seguridad en la aplicación desarrollada usando el estilo arquitectónico REST se escogieron tres herramientas para validar este tema esencial, en este estudio, las herramientas seleccionadas son:

- SoapUI.
- Vega Subgraph.
- OWASP ZAP.

Estas herramientas poseen características imprescindibles para el análisis de la seguridad del software web.

Es importante destacar a la herramienta de software SoapUI, esta es especializada en el análisis de arquitecturas orientadas a servicios SOA y REST, efectúa invocaciones y simulación de ataques a los servicios web.

Por otro lado las herramientas Vega Subgraph y OWASP ZAP son herramientas de análisis de seguridad software como tal y no a los servicios REST, de este modo evalúa rutas, uso de cabeceras HTTP seguras; en síntesis, su análisis no es especializado hacía los servicios web.

Todas estas observaciones se reflejan en los resultados de las validaciones realizadas, puntos siguientes:

4.3.1. Resultados | Alertas de seguridad.

Se iniciará este apartado demostrando la totalidad de alertas de seguridad encontradas en la figura 46. Corresponde al extracto de todas las alertas encontradas en las distintas herramientas de software seleccionadas.

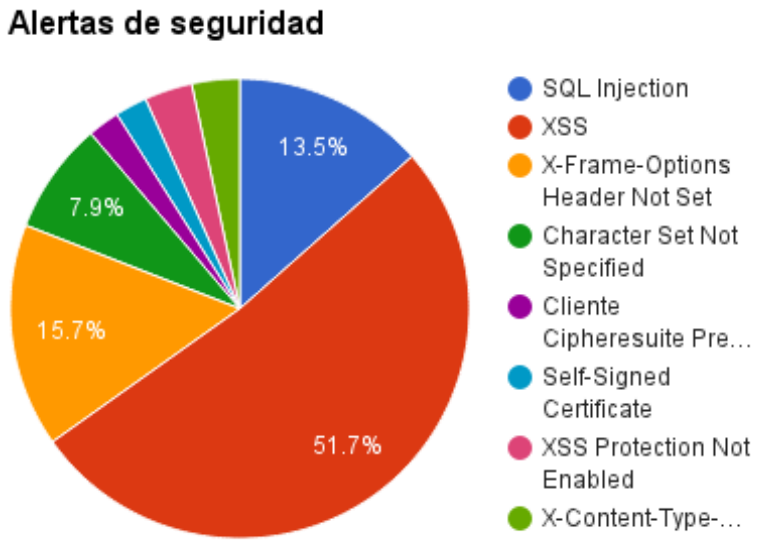


Figura 46: Pruebas de seguridad | Alertas de seguridad.

Elaboración. El Autor.

La mayoría de alertas encontradas hacen referencia a los ataques XSS, X-Frame-Options e Inyección SQL. De forma integral resultan las vulnerabilidades elegidas para este estudio; y, las que mayormente se presentan en resultado de todas las etapas de pruebas ejecutadas.

4.3.2. Iteraciones de seguridad.

Las distintas iteraciones de seguridad se realizaron en base vulnerabilidad, capa de transporte y herramienta de pruebas, de modo tal que se seccionarán las pruebas por herramienta de software.

4.3.2.1. SoapUI

Como se mencionó anteriormente SoapUI es una herramienta especializada en el análisis de software web que se basa en arquitecturas orientadas a servicios, REST y SOA. En síntesis es la principal herramienta que se usó en el estudio de los servicios web RESTful; específicamente para ejecutar las validaciones correspondientes se usó el servicio para filtrar a una persona por cédula:

- http://host/pft/ws/persona/{cédula}/proyecto.

Los resultados de este estudio se reflejan en la figura 47, contiene de forma general los distintos problemas de seguridad encontrados.

Es importante destacar que la figura en cuestión representa las alertas encontradas, según el método de extracción de datos, vulnerabilidad y capa de transporte.

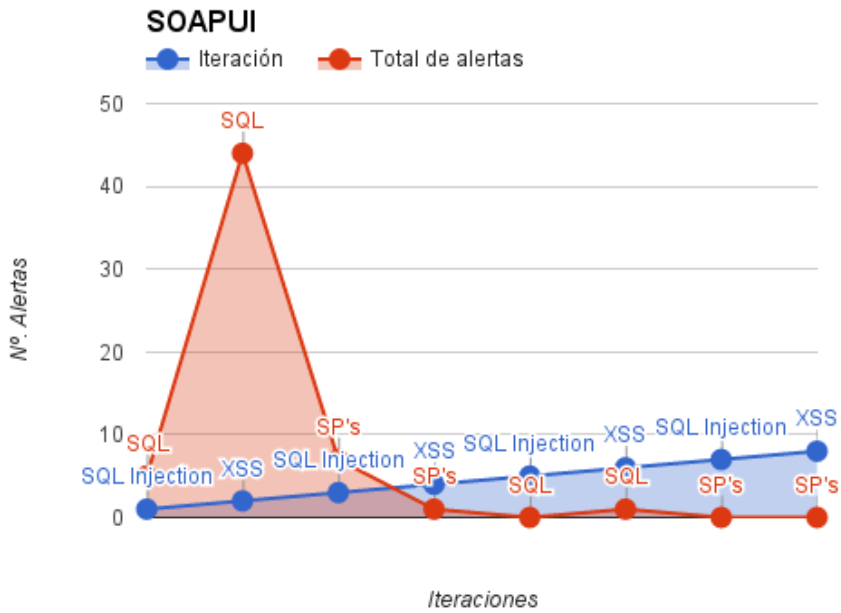


Figura 47: Pruebas de Seguridad | Alertas de seguridad SoapUI.
Elaboración. El Autor.

En la figura 47 se observa un pico de alertas fuerte, en la iteración que corresponde a la vulnerabilidad XSS y SQL como metodología de extracción de datos. Estos resultados se

presentan de forma general; el estudio de estos puntos se lo ve reflejado en las secciones posteriores divididas por vulnerabilidad de estudio.

4.3.2.1.1. Inyección SQL.

A partir de los resultados expuestos en la figura 48, se subdividió las evaluaciones de seguridad, en base a la forma de extracción de datos (Sentencias SQL, Llamadas a Procedimientos Almacenados) escrito en la codificación de la aplicación REST, y su protocolo de transporte configurado a nivel de servidor; a partir de aquí, las pruebas de inyección SQL reflejan los siguientes resultados:

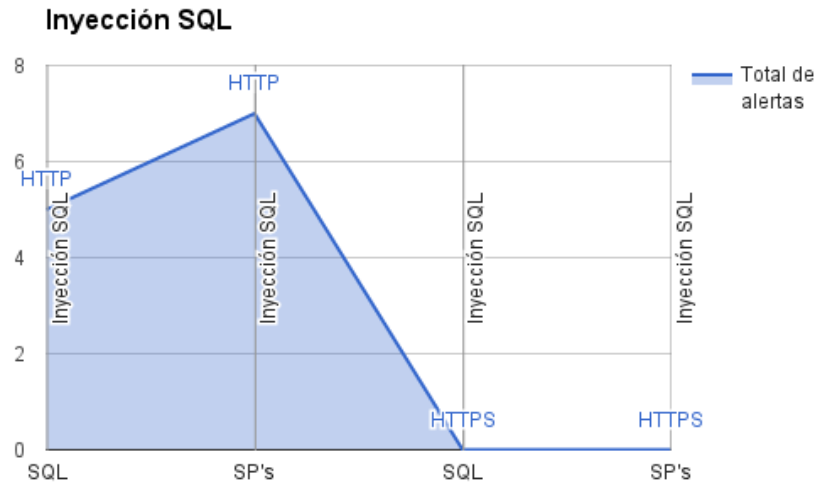


Figura 48: Pruebas de Seguridad | Resultados de alertas de Inyección SQL.
Elaboración. El Autor.

Los resultados reflejan mayor cantidad de alertas de seguridad cuando se usa un protocolo de transporte no seguro. También se aprecia que los mayores problemas de seguridad corresponden a las dos formas de extracción de datos aplicadas; es decir, el aseguramiento de este tipo de vulnerabilidad viene dado por la escritura de código y se ve complementada con la aplicación del protocolo de transporte seguro HTTPS.

4.3.2.1.2. Cross Site Scripting (XSS).

En lo referente a XSS, se aplicó una validación similar a la vulnerabilidad anterior; se evaluó a partir de capa de transporte, vulnerabilidad, y forma de extracción de datos, dichos resultados se ven reflejados en la figura 49.

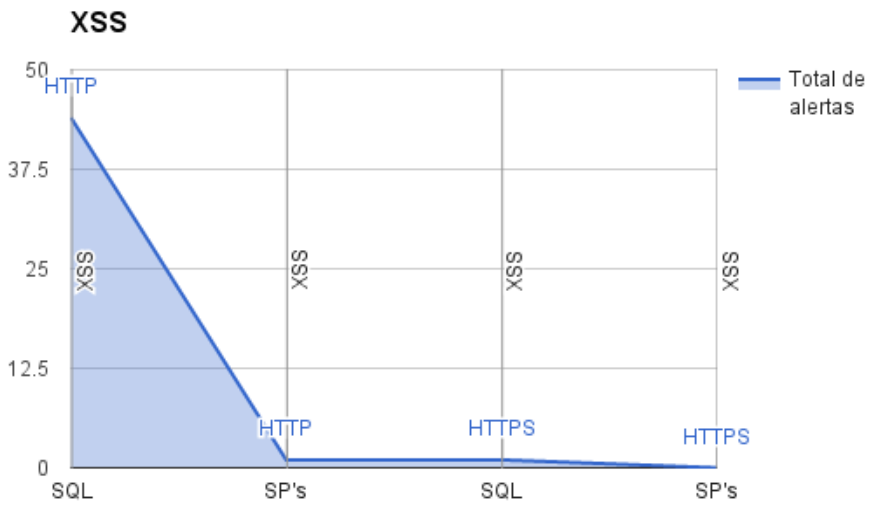


Figura 49: Pruebas de Seguridad | Resultado de alertas de XSS.
Elaboración. El Autor.

En la figura 49 expuesta, se observa un pico de alertas cuando se usa el protocolo de transporte no seguro y sentencias SQL para la extracción de datos.

Hay que recalcar que la técnica de llamadas a procedimientos almacenados brinda un eficiente grado de seguridad, incluso cuando se usa un protocolo de transporte no seguro.

4.3.2.2. Vega Subgraph.

Esta herramienta de software es un escáner de código y pruebas con licenciamiento libre desarrollado bajo JAVA para el análisis de aplicaciones web. Vega es un software especializado en la búsqueda de problemas de seguridad referente a las vulnerabilidades estudiadas.

El uso de esta herramienta para las pruebas de seguridad correspondientes se hizo en dos iteraciones que se dividieron a partir de los protocolos de transporte aplicados.

- **HTTP.**

En este primer análisis se obtuvieron un total de doce alertas de seguridad referentes a dos tipos de vulnerabilidades, las mismas que se presentan en la figura 50.

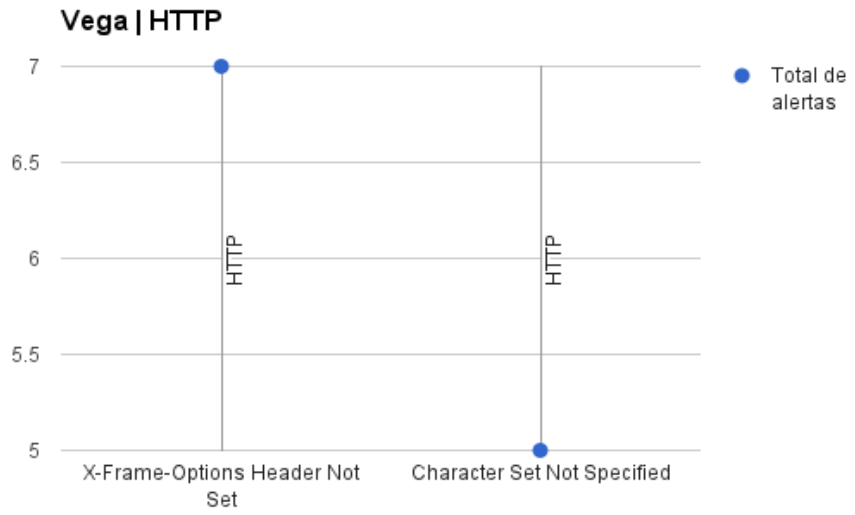


Figura 50: Pruebas de Seguridad | Resultado de alertas de seguridad - HTTP.
Elaboración. El Autor.

Las alertas de seguridad resultantes representan la respuesta del servidor de aplicaciones. Para lo cual se adicionó a la respuesta común del servidor las cabeceras listadas:

- X-Frame-Options
- @Produces({"application/json"+ ";charset=utf-8"})

Donde la primera bloquea al navegador o atacante, renderiza la aplicación mediante el uso de etiquetas de embebido HTML, para que el contenido de la aplicación no sea objeto de contenido para otros sitios web; la segunda es la anotación necesaria para dar conocimiento al navegador el formato de salida de información, en el caso de nuestro estudio s específica el uso de JSON con una codificación UTF-8.

- **HTTPS.**

En segunda instancia se analizó las vulnerabilidades resultantes, aplicando la técnica de HTTPS o protocolo de transporte seguro. En ese estudio se encontró los siguientes problemas de seguridad, se exponen en la figura 51.

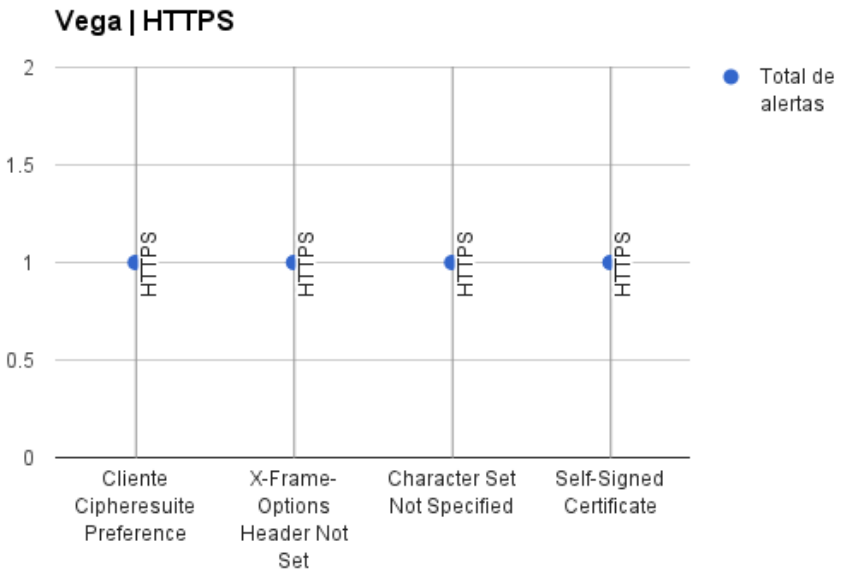


Figura 51: Pruebas de Seguridad | Resultado de alertas de seguridad - HTTPS.
Elaboración. El Autor.

Las alertas resultantes, hacen referencia; la primera y cuarta a la configuración de HTTPS, a nivel de servidor y la segunda y tercer resultan debido a las cabeceras de respuesta del servidor de aplicaciones; motivo por el cual ya no se buscó una resolución a dichas alertas.

4.3.2.3. OWASP ZAP.

OWASP ZAP, es un escáner de seguridad de aplicaciones web que posee licenciamiento libre; este es un producto desarrollado como parte de uno de los proyectos de seguridad auspiciados por OWASP. Esta aplicación funciona como un proxy de tal manera que permite al usuario manipular todo el tráfico que pasa por dicha herramienta.

Las validaciones realizadas mediante el uso de dicho artefacto de software, se dividieron por capa de transporte; similar a la técnica usada en el punto anterior.

- **HTTP.**

En primera instancia se validó la aplicación sobre un protocolo de transporte no seguro, dando como resultado alertas referentes al uso de cabeceras de respuesta segura. Seguidamente en la figura 52 se demuestra el resultado de este análisis.

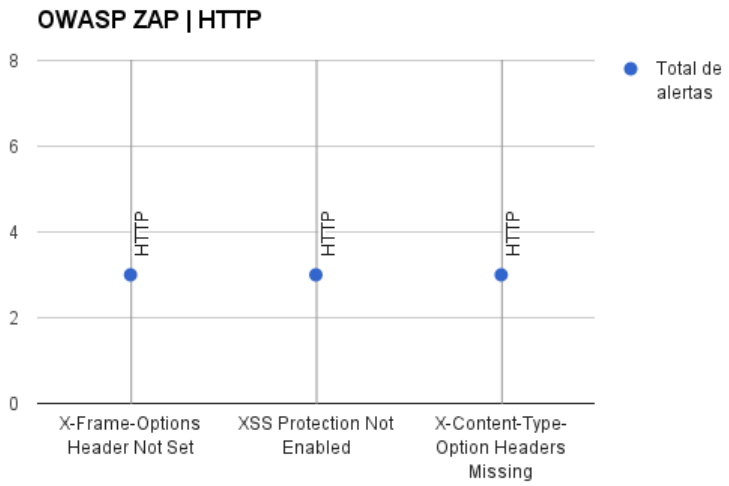


Figura 52: Pruebas de Seguridad | Resultado de alertas de seguridad – HTTP – OWASP ZAP.

Elaboración. El Autor.

Las mismas que se solucionaron de forma similar a las alertas expuestas por la herramienta anterior; se agregaron cabeceras seguras de respuesta, recomendadas por OWASP.

- X-Frame-Options.
- X-XSS-Protection.
- X-Content-Type-Options.

- **HTTPS.**

El análisis resultante, usando este protocolo de transporte seguro reflejó las siguientes alertas detalladas en la figura 53.

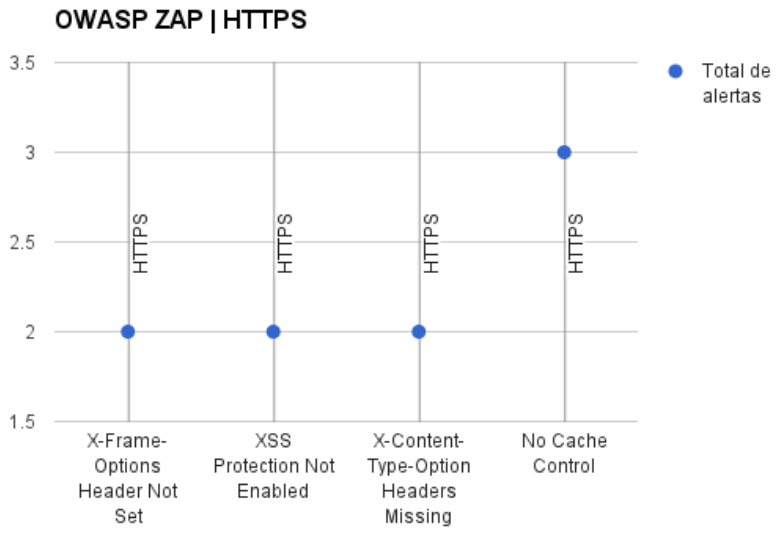


Figura 53: Pruebas de Seguridad | Resultado de alertas de seguridad – HTTPS – OWASP ZAP.

Elaboración. El Autor.

En el que, adicional a las soluciones expuestas en el punto anterior se adicionó la cabecera:

- Cache-Control.

Conviene subrayar que al igual que la Vega Subgraph, OWASP ZAP realiza validaciones a la raíz del servidor por tal motivo las alertas sobrantes no se tomarán en cuenta, debido a que no son alertas que se presentan en los niveles previstos en este estudio.

CONCLUSIONES

Al finalizar el presente trabajo de fin de titulación se concluye:

- El patrón de diseño Facade contribuye a la estructuración del código fuente y la seguridad del software, ya que este mantiene interfaces de comunicación aisladas, permitiendo así que la lógica de programación no se vea expuesta a posibles atacantes.

- El proceso de validación por iteraciones permitió ajustar el código fuente a los estándares de calidad recomendados por la herramienta SonarQube la cual utiliza métricas y estándares de calidad basados en SQALE.

- El uso de normativas OWASP como medio de aseguramiento de aplicaciones web, mitiga en gran medida la ausencia de estándares de seguridad establecidos para REST.

- La implementación de procedimientos almacenados como recomienda OWASP, garantiza el aseguramiento de información y principalmente previene ataques de tipo Inyección SQL, además esta técnica aporta al rendimiento en relación a petición y respuesta de los servicios web RESTful.

- La principal forma de prevención de ataques de tipo XSS consiste en realizar una limpieza preventiva a los parámetros ingresados por el usuario final en los identificadores de recursos (URI), previo a la ejecución y resolución de la petición HTTP.

- El uso de parámetros como parte de la URI representa una buena práctica de diseño de identificadores RESTful ya que beneficia al ocultamiento de variables de especificación de recursos a posibles atacantes.

- El desarrollo de servicios web RESTful certifica total independencia de tecnologías y lenguajes de programación, su lenguaje de respuesta JSON es universal y permite la intercomunicación entre distintos tecnologías de software; además gestiona contenidos más livianos que benefician el rendimiento de aplicaciones web.

- El uso de canales seguros de comunicación como HTTPS es imprescindible dado que cifra las conexiones, ayudando así a prevenir el robo de información y violación de accesos lógicos.

- Es importante en cualquier tipo de desarrollo de software utilizar persistencia de datos, ya que esta permite gestionar la comunicación con cualquier motor de base de datos lo cual garantiza el cumplimiento de atributos de calidad de software como: portabilidad y compatibilidad.

RECOMENDACIONES

Al finalizar el presente trabajo de fin de titulación se recomienda:

- Considerar el establecimiento de estándares de seguridad que establezcan formas de programación, configuración y aseguramiento que garanticen la integridad y disponibilidad de los servicios web RESTful.

- Estudiar, analizar y considerar el ciclo de vida de desarrollo de software seguro establecido por OWASP.

- Examinar el uso de herramientas empleadas para las validaciones efectuadas durante éste estudio; se ha visto que disponen de características con más detalles técnicos que podrían permitir la obtención de resultados más específicos.

- Se debe considerar la extensión de este estudio hacia los demás verbos HTTP, donde se permita validar si el modelo planteado soporta diferentes acciones a los hacia los servicios web RESTful.

- Estudiar normas ISO que garanticen internacionalmente la seguridad y calidad de los servicios web RESTful como alternativa a los resultados generados por SonarQube y SoapUI.

- Se debe ser precavido con el uso de herramientas capaces de generar servicios web RESTful automáticamente ya que estas no garantizan la seguridad, ni tampoco realizan una elección de los recursos a exponer, y siempre se requiere del apoyo de un experto.

- Usar herramientas de control de versiones tomando en consideración que las aplicaciones web REST apuntan hacia la escalabilidad, de modo que se pueda tener control sobre las diferentes versiones y funcionalidades.

- Analizar la opción de configurar un firewall especializado en seguridad de aplicaciones web a nivel de servidor, esta implementación puede reducir los riesgos de vulnerabilidad; actualmente OWASP provee de un firewall capaz de mitigar las diez vulnerabilidades expuestas en el Top Ten 2013.

BIBLIOGRAFÍA

- Barraza, F. (s.f.). *Ciencias e ingeniería de la Computación.* Recuperado el 05 de 12 de 2014, de Modelado y Diseño de Arquitectura de Software: http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:s2_conceptosdemodelado.pdf
- L., B., Clements, P., & Kazman, R. (2002). *Software Architecture in Practice.* Addison-Wesley.
- Rivera Posso, A. N. (2010). *Estudio de la Aarquitectra de Software.* Tesis, Universidad Técnica del Norte, Facultad de Ingeniería en Ciencias Aplicadas.
- Adriana, S. A., & Vanina, P. C. (03 de 2007). *Universidad Nacional De La Patagonia San Juan Bosco .* Recuperado el 05 de 01 de 2015, de Arquitectura de Software: Estilos y Patrones: http://www.dit.ing.unp.edu.ar/graduate/bitstream/123456789/203/1/Tesina%20Arquitectura%20de%20Soft.pdf
- Ramos, J. C., & Depetris, N. (2012). *Universidad Tecnológica Nacional - Facultad Regional Santa Fe.* Recuperado el 08 de 12 de 2014, de Diseño de Software basado en Arquitecturas Estilos Arquitectónicos: http://www.frsf.utn.edu.ar/
- Montilla, J. (15 de 10 de 2013). *Sistema Centrado en Datos*. Recuperado el 06 de 12 de 2014, de Sistema Centrado en Datos (repositorios): http://sistemascentradosendatos.blogspot.com/2013/10/sistemas-centrados-en-datos-repositorios.html
- Reynoso, C., & Kicillof , N. (2004). *Estilos y Patrones en la Estrategia de Arquitectura de Microsoft.* Universidad de Buenos Aires.
- Castro, P. (2004). *Ingeniería del Software Orientada a Objetos.* Universidad Politécnica de Valencia, Instituto Tecnológico de Informática. Valencia: GRUPO QUATREMEDIA.
- Almeira, A. S., & Perez Cavenago, V. (03 de 2007). *Repositorio Graduate.* Recuperado el 06 de 12 de 2014, de Arquitectura de Software: Estilos y Patrones : http://www.dit.ing.unp.edu.ar/graduate/bitstream/123456789/203/1/Tesina%20Arquitectura%20de%20Soft.pdf
- Llorente, C. d., Zorrilla Castro, U., Ramos Barroso, M. A., & Calvarro, J. (2010). *Guía de Arquitectura N-Capas orientada al Dominio con .NET 4.0.* Guía, Microsoft.
- Blanco, C. (2011). *Universidad de Cantabria.* Recuperado el 09 de 01 de 2015, de Ingeniería del Software I: http://ocw.unican.es/ensenanzas-tecnicas/ingenieria-del-software-i/materiales-de-clase-1/is1_t06_Patrones.pdf

- Gracia, J. (27 de 05 de 2005). *IngenieroSoftware*. Recuperado el 09 de 01 de 2015, de Patrones de diseño: http://www.ingenierosoftware.com/analisisydiseno/patrones-diseno.php
- Tedeschi, N. (s.f). *Microsoft Developer Network*. Recuperado el 09 de 01 de 2015, de ¿Qué es un Patrón de Diseño?: http://msdn.microsoft.com/es-es/library/bb972240.aspx
- Lea, D. (1994). *Christopher Alexander: An Introduction for Object-Oriented Designers.* Paper, ACM SIGSOFT.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2003). *Patrones de diseño. Elementos de sofware orientado a objetos reutilizable.* Madrid, España: PEARSON EDUCACIÓN S.A.
- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* . Recuperado el 12 de 11 de 2014, de CHAPTER 5: Representational State Transfer (REST): http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Sun, B. (03 de 08 de 2011). *IBM*. Recuperado el 10 de 11 de 2014, de IBM DeveloperWorks: http://www.ibm.com/developerworks/ssa/library/wa-aj-multitier/
- Cano Parra, R. (11 de 12 de 2012). Entorno de simulación de redes TCP/IP usando servicios REST basados en la nube computacional. Valladolid.
- Patil, M. (2013). *MRBOOL*. Recuperado el 03 de 01 de 2015, de Introduction to REST: http://mrbool.com/introduction-to-rest/29246
- Caules, C. Á. (14 de 06 de 2013). *arquitecturajava*. Recuperado el 12 de 11 de 2014, de Introducción a Servicios REST: http://www.arquitecturajava.com/servicios-rest/
- Velázquez, A. C. (s.f). *Representational State Transfer (REST). Un estilo de arquitectura para Servicios Web. Panorámica y estado del arte.* Recuperado el 12 de 11 de 2014, de Biblioteca de Ingeniería Universidad de Sevilla: http://bibing.us.es/proyectos/abreproy/11247/fichero/Memoria%252F8-Representational+State+Transfer+(REST).pdf
- García Rubí, D. I., & Ríos Clemente, E. (2011). *Canal cifrado para comunicación cliente-servidor.* Tesis, Universidad Nacional Autónoma de Mexico, Mexico D.F.
- Roldán, C. S. (12 de 09 de 2012). *CODEJOBS. Aprende a programar*. Recuperado el 12 de 01 de 2015, de Seguridad Informática: ¿Qué es una vulnerabilidad, una amenaza y un riesgo?: http://www.codejobs.biz/es/blog/2012/09/07/seguridad-informatica-que-es-una-vulnerabilidad-una-amenaza-y-un-riesgo#sthash.qAzjWUkN.dpbs
- Somarriba, H. (2004). *El ABC de la Gestión de Riesgos.* HUMBOLDT.

- Prandini, P., & Pallero, M. (25 de 05 de 2013). *Magazcitum* . Recuperado el 12 de 01 de 2015, de Vulnerabilidades, amenazas y riesgo en “texto claro”: http://www.magazcitum.com.mx/?p=2193
- INTECO. (s.f.). *¿Qué son las vulnerabilidades del software?* Observatorio de la Seguridad de la Información.
- Mifsud, E. (26 de 03 de 2012). *Observatorio Tecnológico*. Recuperado el 12 de 01 de 2015, de MONOGRÁFICO: Introducción a la seguridad informática - Vulnerabilidades de un sistema informático: http://recursostic.educacion.es/observatorio/web/gl/software/software-general/1040-introduccion-a-la-seguridad-informatica
- Livshits, V. B., & Lam, M. S. (2005). *Finding Security Vulnerabilities in Java Applications with Static Analysis.* Recuperado el 28 de 04 de 2015, de usenix The Advanced Computing Systems Association: https://www.usenix.org/legacy/event/sec05/tech/full_papers/livshits/livshits_html/
- Tarlogic. (2013). *Tarlogic*. Recuperado el 08 de 12 de 2014, de Auditoría OWASP: https://www.tarlogic.com/servicios/auditorias-de-seguridad-it/auditoria-seguridad-owasp
- Díaz, V. A. (09 de 2013). OWASP Top 10 2013: actualización de los riesgos más extendidos asociados a las aplicaciones web . *SIC* .
- Fernández, A. (12 de 11 de 2013). *Asociación de Desarrolladores Web de España*. Recuperado el 24 de 03 de 2015, de Servicios web RESTful con HTTP. Parte I: Introducción y bases téoricas: http://www.adwe.es/general/colaboraciones/servicios-web-restful-con-http-parte-i-introduccion-y-bases-teoricas
- Paredes, A. M. (15 de 07 de 2012). *Arquitectura de Software*. Recuperado el 12 de 11 de 2014, de Spring V.S. Java EE: http://elblogdelfrasco.blogspot.com/2012_07_01_archive.html
- Marqués, A. (11 de 04 de 2013). *Apuntes personales, desarrollo de software y negocios en Internet*. Obtenido de Conceptos sobre APIs REST: http://asiermarques.com/2013/conceptos-sobre-apis-rest/
- ORACLE. (s.f.). *ORACLE Help Center*. Recuperado el 13 de 04 de 2015, de 5 Securing RESTful Web Services: http://docs.oracle.com/cd/E24329_01/web.1211/e24983/secure.htm#RESTF254
- hello2morrow. (2015). *HELLO2MORROW Empowering Software Craftsmanship*. Recuperado el 02 de 06 de 2015, de Sonargraph Product Family: https://www.hello2morrow.com/products/sonargraph/architect
- Universidad Carlos III. (2013). *Software*. Obtenido de Conceptos Generales de Informática: http://www.giaa.inf.uc3m.es/docencia/trabajo/tema1.pdf

- JavaServer Faces Technology. (2012). *JSF Architecture.* Obtenido de http://aragorn.pb.bialystok.pl/~dmalyszko/PSS_Project/JavaServer%20Faces.htm
- Martínez, A. N. (2008). *Marco de trabajo para el desarrollo de arquitecturas de software orientado a aspectos.* Tesis, Universidad de Extremadura, Departamento de Ingeniería de Sistemas Informáticos y Telemáticos .
- Universidad de Buenos Aires. (11 de 09 de 2008). *Departamento de Computación.* Recuperado el 29 de 12 de 2014, de Ingeniería de Software II: http://www-2.dc.uba.ar/materias/isoft2/2008_02/clases/Clase7-DocumentacionArquitecturas-Viewtypes.pdf
- Ruiz, F. (2010). *CTR Computadores y Tiempo Real*. Recuperado el 29 de 12 de 2014, de Ingeniería de Software I: http://www.ctr.unican.es/asignaturas/is1/is1-t13-trans.pdf
- Cordero, R. N. (2010). *Las vistas arquitectónicas de software y sus correspondencias mediante la gestión de modelos.* Tesis, Universidad Politécnica de Valencia, Departamento de Sistemas Informáticos y Computación.
- GISC. (2002). *Universidad de las Palmas de Gran Canaria.* Recuperado el 29 de 12 de 2014, de Grupo de Ingeniería de software y del conocimiento: http://serdis.dis.ulpgc.es/~a034403/carpeta/is1/Apuntes/UT04.%20Fundamentos%20del%20dise%F1o.pdf
- Universidad de Santander. (2013). *Universidad de Santander.* Recuperado el 29 de 12 de 2014, de Aula Virtual UDES: http://aulavirtual.eaie.cvudes.edu.co/publico/lems/L.000.008.MG/Documentos/cap3/3_1.pdf
- Dalling, T. (31 de 05 de 2009). *Tom Dalling*. Recuperado el 05 de 01 de 2015, de Model View Controller Explained: http://www.tomdalling.com/blog/software-design/model-view-controller-explained/
- Tabares, M. S. (05 de 09 de 2011). *Slideshare.* Recuperado el 06 de 12 de 2014, de Arquitecturas de software - Parte 2: http://es.slideshare.net/mstabare/arquitecturas-de-software-parte-2
- Almeira, A. S., & Cavenago Perez, V. (2007). *Arquitectura de Software: Estilos y Patrones.* Tesis, Universidad Nacional de la Patagonia San Juan Bosco, Facultad de Ingeniería .
- Ros, J. N. (2011). *Construcción de Software.* Presentación, Universidad de Murcia, Facultad de Informática, Murcia.
- EPICOR Business Inspired. (2014). *EPICOR Business Inspired*. Recuperado el 05 de 01 de 2015, de Arquitectura Orientada a Servicios: http://www.epicor.com/lac/Solutions/Pages/Serviceoriented.aspx

- Centro de Alto Rendimiento de Accenture (CAR). (2008). Arquitectura Orientada a Servicios (SOA). *accenture* .
- Santos, M. (19 de 12 de 2013). *Las tendencias de seguridad informática para 2014*. Recuperado el 23 de 04 de 2015, de ENTER.CO: www.enter.co/chips-bits/enterprise/las-tendencias-de-seguridad-informatica-para-el-2014/

ANEXOS

ANEXO A. Modelo de datos para el proyecto de seguimiento a trabajos de fin de titulación.

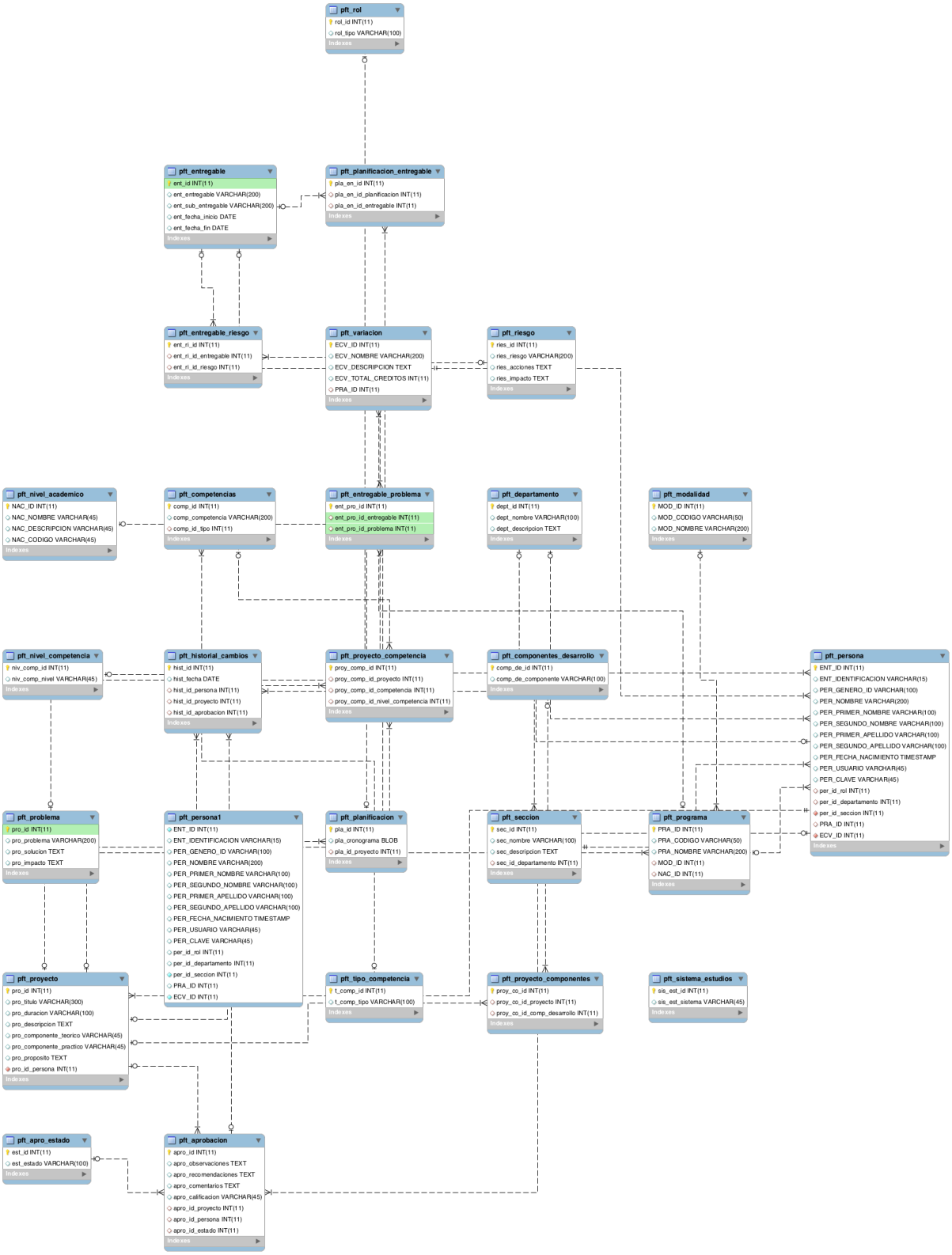


Figura 54: Anexo A | Modelo de datos para proyectos de fin de titulación.

Elaboración. El Autor.