



Outside

APIs



SOA

Enterprise



Integrando Servicios

Una Visión Práctica de SOA

J. Francisco Ibañez

Copyright © 2026 Juan Francisco Ibañez Salas.

Marcas Comerciales. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. En este libro se ha intentado distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma.

Todos los derechos reservados. No se puede utilizar ni reproducir ninguna parte de este libro en ninguna forma sin permiso escrito, excepto en el caso de citas breves en artículos críticos o reseñas.

Impreso en México.

Para mas información contacto:
pacois20@gmail.com

Diseño del Libro por Juan Francisco Ibañez Salas
Diseño de la cubierta por Juan Francisco Ibañez Salas

ISBN - 978-607-29-7257-5

Primera Edición: febrero 2026

INTRODUCCIÓN	5
FUNDAMENTOS DE SOA Y APIS	7
ORÍGENES DE SOA Y APIS	8
CONCEPTOS Y TECNOLOGÍAS	11
AMBIENTES DISTRIBUIDOS	17
INTERFACES	19
DISEÑO DE SERVICIOS	24
IMPORTANCIA DE LAS APIS	25
CLASIFICACIÓN DE APIS:	26
DISEÑANDO DE UNA API REST SIMPLE	29
INTEGRACIÓN DE APLICACIONES CON TECNOLOGÍAS ACTUALES	49
TECNOLOGÍAS EN INTEGRACIÓN DE SERVICIOS	50
SOAP (SIMPLE OBJECT ACCESS PROTOCOL)	51
EXTENSIBLE MARKUP LANGUAGE (XML)	51
WEB SERVICES DESCRIPTION LANGUAGE	54
DISEÑANDO UNA API SOAP SIMPLE	55
<i>Publicación del servicio</i>	56
<i>Creación del servicio (webService)</i>	66
<i>Consumiendo Servicios</i>	70
TECNOLOGÍAS ACTUALES EN PATRONES DE ARQUITECTURAS DE APIS	84
<i>Tendencias Emergentes en APIs</i>	88
SEGURIDAD EN LA INTEGRACIÓN DE SERVICIOS	91
PROTECCIÓN DE DATOS SENSIBLES	94
CIFRADO DE DATOS	98
FORMA SIMPLE DE CIFRADO EN NODE.JS	103
CASOS DE EXITO	108
NETFLIX COMO REFERENTE DE MIGRACION A LA NUBE	110

<i>Netflix sigue evolucionando.</i>	117
UBER UN MODELO DE NUBE HÍBRIDA	119
<i>La pila tecnológica de Uber (Interacción de apps)</i>	123
CISCO SYSTEMS Y LA IMPLEMENTACIÓN DE SERVICIOS	131
<i>La nueva WAN</i>	135
ACERCA DEL AUTOR	140
AGRADECIMIENTOS	141

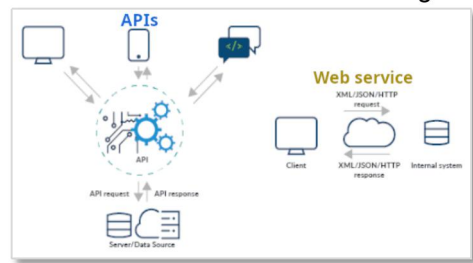
Introducción

En la era digital actual, las organizaciones enfrentan desafíos cada vez más complejos para integrar sistemas, optimizar procesos y ofrecer servicios eficientes y escalables.

En este contexto, la Arquitectura Orientada a Servicios (SOA) y las Interfaces de Programación de Aplicaciones (APIs) han emergido como pilares fundamentales para el desarrollo de soluciones tecnológicas flexibles y robustas.

Este libro ofrece una visión práctica y accesible de SOA y APIs, con el objetivo de proporcionar a profesionales, desarrolladores y estudiantes las herramientas necesarias para comprender y aplicar estos

conceptos en escenarios del mundo real. A diferencia de otras obras que se centran exclusivamente en la teoría, aquí se adoptan un enfoque pragmático y ejemplos concretos que facilitan la asimilación de los principios esenciales de SOA y APIs, así como su análisis para la implementación en entornos empresariales.



La organización de este libro permite una lectura progresiva y estructurada. Se inicia con una introducción a los fundamentos de SOA y APIs, estableciendo sus diferencias y relaciones clave. Posteriormente, se abordan los principios de diseño, las mejores prácticas y las estrategias de implementación, complementadas con estudios de caso y analogías que ilustran su aplicación en distintos contextos.

En un mundo donde la interconectividad es esencial para la competitividad y la innovación, comprender cómo funcionan SOA y las APIs es una habilidad imprescindible. Este libro no solo explica los conceptos clave, sino que también busca fomentar un pensamiento crítico y una capacidad de análisis que permita a los lectores diseñar y desarrollar arquitecturas eficientes y escalables.

Confío en que esta obra sea una referencia valiosa para todos aquellos interesados en el mundo de la integración de servicios y las APIs, esperando que esta lectura inspire a estudiantes y profesionales a enfrentar con confianza los retos tecnológicos del presente y del futuro.

CAPÍTULO 1

Fundamentos de SOA y APIs

Un Vistazo a la Integración Moderna

El objetivo de este capítulo es comprender los principios básicos de SOA y su propósito en la integración de sistemas y aplicaciones.



En el entorno tecnológico actual, las empresas deben integrar diversas aplicaciones y servicios que operan de manera aislada. Para facilitar esta integración y la interoperabilidad entre plataformas y tecnologías heterogéneas, se han desarrollado dos enfoques clave: la Arquitectura Orientada a Servicios (SOA) y las Interfaz de Programación de Aplicaciones (APIs). Aunque ambos conceptos están estrechamente relacionados, cada uno tiene sus propias características y beneficios en la construcción de sistemas modulares y escalables.

SOA es un enfoque arquitectónico que permite que los servicios, representados como unidades funcionales independientes, interactúen entre sí a través de una red.

Este modelo se centra en la creación de servicios reutilizables y desacoplados que pueden ser consumidos por diferentes aplicaciones. Por otro lado, las APIs actúan como contratos de comunicación que permiten la interacción entre servicios, facilitando la integración entre aplicaciones de diferentes orígenes y tecnologías.

En este capítulo, se exploran las bases de SOA y las APIs, analizando sus diferencias, complementos y la manera en que transforman el desarrollo de software.

Orígenes de SOA y APIs

La evolución de la arquitectura de software ha estado marcada por la necesidad de mejorar la interoperabilidad, reutilización y escalabilidad de los sistemas. En este contexto, la Arquitectura Orientada a Servicios (SOA) y las Interfaces de Programación de Aplicaciones (APIs) han desempeñado un papel clave en la transformación digital de las organizaciones.



<<Gráfica - Evolución de Arquitecturas>>

La Arquitectura Orientada a Servicios (SOA) surge a finales de la década de 1990 y principios de los 2000 como un paradigma de diseño de software que busca modularizar sistemas mediante la creación de servicios autónomos y reutilizables que interactúan a través de interfaces bien definidas (Erl, 2005). Su desarrollo fue impulsado por la creciente complejidad de los sistemas empresariales y la necesidad de integrar aplicaciones heterogéneas dentro de un mismo ecosistema tecnológico.

SOA se basa en principios fundamentales como la interoperabilidad, la abstracción de servicios y la comunicación mediante protocolos estándar, como SOAP (Simple Object Access Protocol) y WSDL (Web Services Description Language). Estos estándares permitieron la interacción entre sistemas de distintos fabricantes y tecnologías, lo que facilitó la integración de aplicaciones en entornos empresariales (Papazoglou, 2008).

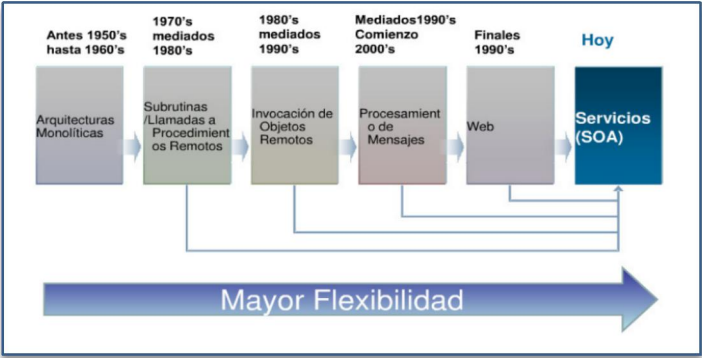
A su vez las APIs han existido desde las primeras etapas del desarrollo de software, con interfaces diseñadas para facilitar la comunicación entre módulos de un mismo programa o entre sistemas diferentes. Sin embargo, el concepto moderno de API, especialmente en su versión web, comenzó a ganar relevancia en la década de 2000 con el auge de la web y los servicios en la nube.

Las APIs RESTful (basadas en el modelo Representational State Transfer, REST) fueron propuestas por Fielding (2000) como un enfoque más ligero y flexible en comparación con SOAP. REST permite la comunicación entre sistemas utilizando protocolos estándar como HTTP, lo que facilitó la adopción de APIs en entornos web.

Empresas como Amazon, Google y Facebook fueron pioneras en la implementación de APIs públicas para permitir la integración con sus plataformas y fomentar la innovación en el ecosistema digital (Richardson & Ruby, 2007).

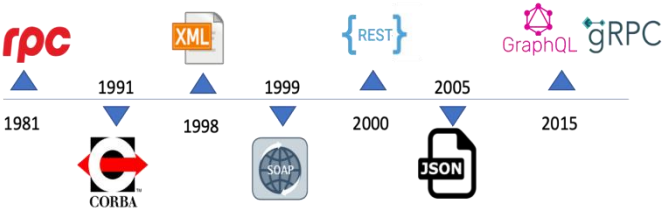
Aunque SOA y las APIs han evolucionado de manera independiente, existe una relación intrínseca entre ambos conceptos. SOA introdujo el enfoque de servicios como una unidad modular de software, mientras que las APIs han sido el medio a través del cual estos servicios se exponen e interactúan con otros sistemas.

Con el tiempo, el uso de APIs ha evolucionado más allá del contexto de SOA. Mientras que SOA se centra en la arquitectura empresarial y la integración de sistemas complejos, las APIs modernas (especialmente REST y GraphQL) han facilitado el desarrollo ágil de aplicaciones, la interoperabilidad en la web y la creación de ecosistemas digitales abiertos (Newman, 2015).



<<Gráfica - Evolución SOA>>

En la actualidad, el enfoque de microservicios toma elementos de SOA y los combina con APIs modernas para construir aplicaciones escalables y distribuidas. Los microservicios utilizan APIs como mecanismo de comunicación entre servicios independientes, permitiendo una mayor flexibilidad y resiliencia en comparación con las arquitecturas monolíticas tradicionales (Fowler & Lewis, 2014).



<<Gráfica - Evolución nuevas tecnologías>>

En la evolución de las arquitecturas van apareciendo nuevos conceptos y nuevas tecnologías desde lo monolítico pasando por arquitecturas cliente-servidor, APIs, hasta lo que conocemos como SOA

Característica	SOA	APIs
Propósito	Integrar servicios de manera flexible y reutilizable	Facilitar la comunicación entre aplicaciones y servicios
Enfoque	Desacoplamiento de servicios	Exposición de interfaces para consumo
Protocolos comunes	SOAP, XML-RPC, REST	REST, GraphQL, gRPC
Escalabilidad	Alta (pero más compleja en configuración)	Alta (escalabilidad dependiente de la API)
Interoperabilidad	Alta, con diferentes sistemas y plataformas	Alta, especialmente en entornos web y móviles

<<Tabla comparativa de SOA y APIs>>

Para entender el entorno de SOA y API, revisaremos algunos conceptos y tecnologías y su interacción, como lo son modelos y estándares, los cuales son importantes en la arquitectura de soluciones informáticas.

Conceptos Y Tecnologías

Internet y la Web

El desarrollo de la tecnología de la información y las comunicaciones ha llevado a la interconexión global de computadoras y dispositivos, facilitando el intercambio de información y servicios digitales, en este contexto Internet y Web, aunque a menudo se usan indistintamente, representan conceptos distintos con funciones específicas dentro del ecosistema digital, y son conceptos estrechamente relacionados pero distintos.

Internet infraestructura global que permite la comunicación de dispositivos, mientras que la Web es un sistema de información basado en hipertexto que utiliza Internet como plataforma.

La evolución de ambos ha transformado la sociedad moderna, impulsando la globalización, la educación, el comercio y la comunicación digital.

Internet es una red global de computadoras interconectadas que permite la comunicación y el intercambio de datos mediante protocolos estándar, principalmente el Protocolo de Internet (**IP**) y el Protocolo de Control de Transmisión (**TCP**). Surgió como un proyecto del Departamento de Defensa de los Estados Unidos en la década de 1960, bajo el nombre de ARPANET, con el objetivo de desarrollar una infraestructura de comunicación descentralizada y resistente a fallos (Leiner et al., 2009). Internet es, por tanto, la base sobre la que se construyen múltiples servicios digitales, como el correo electrónico, la mensajería instantánea, la transmisión de archivos y la World Wide Web (WWW).

Web o WWW es un sistema de información basado en hipertexto que funciona sobre Internet. Fue desarrollada por **Tim Berners-Lee** en 1989 en el Consejo Europeo para la Investigación Nuclear (**CERN**). Se fundamenta en el uso de páginas web interconectadas mediante enlaces de hipertexto y accesibles a través de navegadores web (Berners-Lee, 1996), elementos de la Web incluyen:

HTML (HyperText Markup Language): Lenguaje de marcado utilizado para estructurar el contenido de las páginas web.

HTTP (HyperText Transfer Protocol): Protocolo que facilita la comunicación entre navegadores y servidores web.

URL (Uniform Resource Locator): Dirección única que identifica recursos en la Web.

Sin Internet no podría existir la Web, pero Internet puede existir sin la Web, ya que otros servicios (como el correo electrónico o la comunicación VoIP) también dependen de esta red global (Krol, 1993).

El entorno de Internet y Web abarca una serie de elementos clave que permiten su funcionamiento y expansión:

Infraestructura de redes: Incluye servidores, enrutadores, centros de datos y proveedores de servicios de Internet (ISP).

Protocolos de comunicación: Como TCP/IP, HTTP y HTTPS, que regulan la transferencia de datos.

Servicios y aplicaciones: Motores de búsqueda, plataformas de comercio electrónico, redes sociales, entre otros.

Usuarios y regulación: Implica la interacción de individuos, empresas y gobiernos en el uso, regulación y desarrollo de la Web e Internet.

Modelo

Los modelos permiten estructurar la forma en que los componentes de un sistema interactúan entre sí y ayudan a establecer principios para su desarrollo y mantenimiento. En los ambientes de software, los modelos pueden utilizarse para describir arquitecturas, patrones de diseño, procesos de desarrollo y paradigmas de programación.

Modelo abstracción que representa un sistema o un conjunto de procesos, facilitando su análisis, diseño y comprensión.

Los modelos en software pueden clasificarse en diversas categorías, tales como modelos arquitectónicos, modelos de datos, modelos de procesos y modelos de comportamiento. Cada tipo de modelo tiene un propósito específico dentro del ciclo de vida del software y contribuye a garantizar su calidad, escalabilidad y mantenibilidad (Bass, Clements & Kazman, 2012).

Uno de los modelos más relevantes en el desarrollo de software es el modelo **cliente-servidor**. Este modelo define una arquitectura en la que una entidad, denominada cliente, solicita servicios o recursos a otra entidad, el servidor, que los proporciona.

Esta estructura permite una distribución eficiente de las tareas y se emplea ampliamente en sistemas distribuidos y aplicaciones web y los clientes pueden ser dispositivos como computadoras personales, teléfonos móviles o aplicaciones web que envían peticiones al servidor mediante protocolos de comunicación, como HTTP o TCP/IP. El servidor, por su parte, procesa las solicitudes y devuelve la información o servicio requerido. Ejemplos comunes de este modelo incluyen servidores web, bases de datos centralizadas y servicios en la nube (Tanenbaum & Van Steen, 2017).

Un ejemplo típico de este modelo es un sistema de correo electrónico basado en la web. Cuando un usuario accede a su cuenta de correo mediante un navegador, este actúa como cliente y envía una solicitud al servidor de correo, el cual procesa la petición, verifica las credenciales y devuelve la bandeja de entrada del usuario con sus correos electrónicos. En este proceso, el cliente solo visualiza la interfaz, mientras que el servidor gestiona los datos y operaciones en segundo plano.

Para comprender este modelo de manera sencilla, podemos compararlo con un restaurante, en esta analogía:

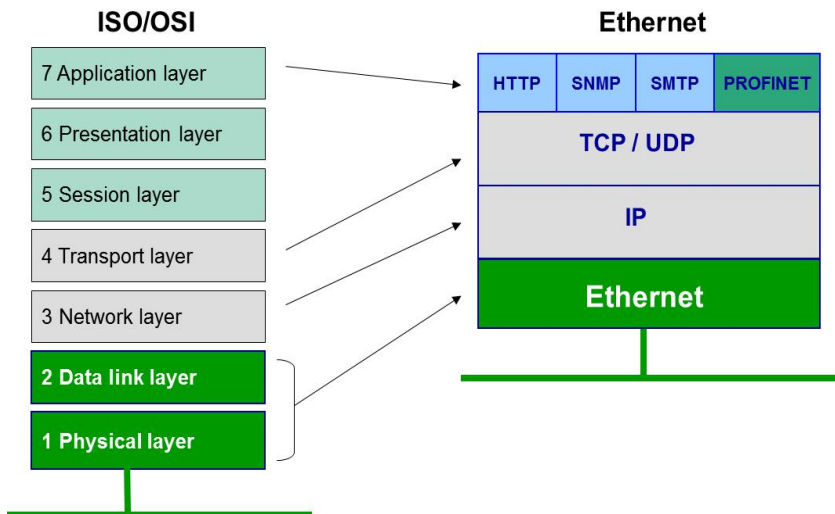
El **cliente** es el comensal que llega al restaurante y ordena un platillo.

El **servidor** es la cocina del restaurante, que recibe la orden, la prepara y la entrega al mesero para su distribución.

El **mesero** actúa como intermediario entre el cliente y la cocina, transmitiendo las solicitudes y entregando los resultados.

Así como en un restaurante, donde el cliente no necesita conocer el proceso de preparación de los alimentos, en el modelo cliente-servidor, el cliente solo recibe el servicio sin preocuparse por los detalles técnicos de su procesamiento.

Los modelos de referencia, son modelos normalizados que proporcionan una visión integrada de alto nivel de una tecnología y sus datos; se utilizan como referencia para la construcción de modelos similares.



<<Gráfica - Modelos de referencia>>

Protocolos

Otros conceptos importantes y que se acaban de mencionar son los protocolos, HTTP y TCP/IP, establecen cómo se transmiten, reciben y procesan los datos, asegurando la interoperabilidad entre sistemas heterogéneos. En informática y telecomunicaciones, los protocolos permiten que los dispositivos intercambien información de manera estructurada y eficiente.

***Protocolo** conjunto de reglas y estándares que regulan la comunicación entre dispositivos dentro de una red.*

El protocolo **HTTPS** (HyperText Transfer Protocol Secure) es una versión segura de HTTP que incorpora mecanismos de cifrado mediante el protocolo **TLS** (Transport Layer Security) o su versión predecesora SSL (Secure Sockets Layer). Su función principal es proteger la integridad y confidencialidad de la comunicación entre el cliente (navegador) y el servidor web.

Características principales de HTTPS:

Cifrado: HTTPS cifra los datos transmitidos, impidiendo que terceros intercepten información confidencial, como contraseñas o datos bancarios.

Integridad: Garantiza que los datos no sean alterados durante la transmisión.

Autenticación: Utiliza certificados digitales emitidos por entidades certificadoras (CA) para verificar la identidad del servidor y evitar ataques como el **Man-in-the-Middle (MitM)**.

Característica	HTTP	HTTPS
Seguridad	No cifra los datos	Cifra los datos mediante TLS/SSL
Uso recomendado	Sitios públicos sin información sensible	Sitios que manejan datos privados o transacciones
Certificado SSL/TLS	No requerido	Requerido
Protección contra ataques	Vulnerable a interceptaciones	Protección contra ataques de intermediarios

<<Tabla: Diferencia entre HTTP y HTTPS>>

HTTPS es fundamental en arquitecturas orientadas a servicios (SOA) y aplicaciones web, especialmente en aquellas que manejan datos sensibles, como comercio electrónico y banca en línea. Su implementación se basa en un modelo cliente-servidor, donde el navegador realiza una solicitud segura al servidor web, el cual responde con la página cifrada. Por ejemplo cuando un usuario accede a su banca en línea, el navegador inicia una conexión HTTPS, asegurando que las credenciales y transacciones sean cifradas y protegidas de accesos no autorizados.

El **TCP/IP** es un conjunto de protocolos fundamentales en la comunicación a través de Internet y redes privadas. Se compone de dos protocolos principales, **IP** se encarga de la dirección y el enrutamiento de los paquetes de datos a través de la red, y **TCP** asegura que los datos sean enviados y recibidos en el orden correcto, garantizando la entrega completa y sin errores.

TCP/IP opera en una arquitectura de cuatro capas:

Capa de Acceso: Gestiona la conexión física con la infraestructura de la red.

Capa de Internet: Permite enrutamiento de paquetes mediante direcciones IP.

Capa de Transporte: TCP maneja la transmisión confiable de datos.

Capa de Aplicación: Facilita la interacción con el usuario mediante protocolos como HTTP, FTP o SMTP.

Quando un usuario envía un correo electrónico, TCP/IP descompone el mensaje en paquetes, los envía a través de la red y luego los reensambla en el destino. Si algún paquete se pierde, TCP solicita su retransmisión para garantizar la entrega completa.

Abstracción

Este concepto es un principio fundamental en la ingeniería de software y en el diseño de arquitecturas. Su correcta aplicación permite la construcción de sistemas robustos, escalables y mantenibles. Desde la orientación a objetos hasta las arquitecturas empresariales modernas, la abstracción es clave para manejar la complejidad y promover la reutilización del software.

En el ámbito de la informática y las arquitecturas de software, abstracción se refiere al proceso de ocultar detalles irrelevantes y resaltar las características esenciales de un sistema o componente (Gamma et al., 1994), y es fundamental para la gestión de la complejidad en el diseño y desarrollo de software, permitiendo a los ingenieros enfocarse en los aspectos más relevantes de una solución sin verse abrumados por los detalles de implementación subyacentes.

***Abstracción** mecanismo para definir modelos de alto nivel que encapsulan la funcionalidad sin necesidad de exponer la implementación interna (Larman, 2004).*

A través de este enfoque, se logra modularidad, mantenibilidad y reutilización del código en los sistemas informáticos, y se manifiesta en diferentes niveles, desde la separación de responsabilidades en capas arquitectónicas hasta la implementación de interfaces y patrones de diseño, algunas formas en que la abstracción se aplica en las arquitecturas de software son:

Abstracción en Capas: Las arquitecturas de software suelen estructurarse en capas, cada una con una responsabilidad bien definida. Ejemplos comunes incluyen la arquitectura en tres capas (presentación, lógica de negocio y datos) y la arquitectura en capas de sistemas operativos (Tanenbaum & Bos, 2014).

Patrones de Diseño: Algunos patrones de diseño, como el **Patrón de Fábrica** (FACTORY PATTERN) o el **Patrón de Adaptador** (ADAPTER PATTERN), utilizan la abstracción para desacoplar componentes y facilitar la extensibilidad del software (Gamma et al., 1994).

Interfaces y Programación Orientada a Objetos (POO): En la POO, la abstracción se logra mediante interfaces y clases abstractas que permiten definir contratos de comportamiento sin revelar detalles de implementación (Meyer, 1997).

Arquitecturas Basadas en Servicios (SOA) y Microservicios: En SOA y microservicios, los servicios se exponen a través de interfaces bien definidas, ocultando la lógica interna y permitiendo la interoperabilidad entre sistemas heterogéneos (Erl, 2005).

El uso de la abstracción en arquitecturas de software proporciona múltiples ventajas:

Reducción de la complejidad: Permite descomponer sistemas complejos en componentes manejables.

Reutilización de componentes: Facilita la creación de software modular que puede reutilizarse en distintos contextos.

Mantenimiento y escalabilidad: Favorece la evolución del software sin afectar la funcionalidad existente.

Independencia tecnológica: Separa la implementación de la interfaz, permitiendo cambios en la tecnología subyacente sin afectar a los consumidores del servicio.

Interoperabilidad

La **interoperabilidad** se refiere a la capacidad de diferentes sistemas, aplicaciones o componentes de software para comunicarse, intercambiar datos y operar en conjunto de manera eficiente, independientemente de sus diferencias tecnológicas, arquitectónicas o de implementación (Khosrow-Pour, 2018).

Esta propiedad es fundamental en entornos distribuidos donde múltiples sistemas heterogéneos necesitan interactuar sin dependencia de una plataforma específica.

En el contexto de la arquitectura orientada a servicios (**SOA**) y las interfaces de programación de aplicaciones (**APIs**), la interoperabilidad permite la integración y el funcionamiento coordinado de servicios,

aplicaciones y plataformas a través de estándares comunes y protocolos bien definidos (Papazoglou, 2012).

La interoperabilidad en SOA se vuelve crucial debido a la diversidad de tecnologías y plataformas que pueden coexistir en un ecosistema empresarial.

Algunos aspectos clave incluyen:

Independencia Tecnológica: SOA permite que sistemas desarrollados en diferentes lenguajes y entornos tecnológicos interactúen mediante protocolos estándar como **SOAP (Simple Object Access Protocol)** y **REST (Representational State Transfer)** (Erl, 2017).

Reutilización de Servicios: La interoperabilidad garantiza que los servicios puedan ser utilizados por distintas aplicaciones sin necesidad de modificaciones extensas.

Flexibilidad y Escalabilidad: Facilita la evolución de los sistemas sin interrumpir su operatividad, permitiendo la incorporación de nuevos servicios sin afectar los existentes.

Interacción entre Organizaciones: En entornos empresariales, diferentes sistemas internos y externos pueden compartir servicios mediante estándares abiertos, promoviendo la colaboración y la eficiencia operativa (Josuttis, 2007).

Las **APIs** son interfaces que permiten la comunicación entre aplicaciones y servicios. La interoperabilidad en APIs se refiere a la capacidad de estas interfaces para facilitar la integración sin restricciones tecnológicas. Su importancia radica en:

Conectividad Universal: APIs bien diseñadas permiten la comunicación entre sistemas desarrollados en distintas tecnologías, promoviendo la interoperabilidad.

Uso de Estándares Abiertos: Protocolos como **HTTP**, **JSON**, **OAuth** y **OpenAPI** estandarizan la forma en que las aplicaciones interactúan, asegurando compatibilidad entre distintos sistemas (Fielding, 2000).

Evolución y Mantenimiento: APIs interoperables permiten actualizaciones y mejoras sin afectar las aplicaciones que las consumen, garantizando estabilidad y continuidad operativa.

La interoperabilidad es un pilar fundamental en los entornos de **SOA y APIs**, ya que permite la integración eficiente de servicios y aplicaciones en un ecosistema distribuido y heterogéneo. Su implementación mediante estándares abiertos y protocolos bien definidos facilita la comunicación entre sistemas, optimiza la reutilización de servicios y garantiza la evolución tecnológica sin interrupciones.

Ambientes Distribuidos

El crecimiento exponencial del volumen de datos y la necesidad de procesamiento eficiente han impulsado la evolución de los ambientes distribuidos. Estos entornos permiten la distribución de tareas

computacionales entre varios nodos, ya sea para optimizar el rendimiento, aumentar la disponibilidad o mejorar la tolerancia a fallos. Sin embargo, dentro de estos ambientes es esencial diferenciar entre dos conceptos clave: el cómputo distribuido y los sistemas distribuidos. Mientras que el primero se centra en la ejecución coordinada de procesos en múltiples nodos, el segundo abarca una infraestructura más amplia que integra diversos recursos y servicios en una red heterogénea.

Cómputo Distribuido El cómputo distribuido se refiere a la ejecución de procesos computacionales en múltiples unidades de procesamiento interconectadas (Tanenbaum & Van Steen, 2017). Este modelo busca mejorar el desempeño mediante la paralelización de tareas, dividiéndolas en sub tareas que se ejecutan simultáneamente en distintos nodos. Algunos ejemplos clásicos de cómputo distribuido incluyen los sistemas de clúster y las arquitecturas de procesamiento paralelo, como MapReduce en Hadoop (Dean & Ghemawat, 2008), una de las principales ventajas del cómputo distribuido es la eficiencia en el procesamiento de grandes volúmenes de datos. No obstante, también enfrenta desafíos como la sincronización de tareas, la comunicación entre nodos y la tolerancia a fallos. Además, su éxito depende en gran medida de algoritmos de distribución y balanceo de carga efectivos.

Sistemas Distribuidos Se definen como un conjunto de computadores independientes que aparecen ante los usuarios como un sistema único y cohesionado (Coulouris, Dollimore, & Kindberg, 2013), que proporcionan servicios a través de redes interconectadas, permitiendo la interoperabilidad entre distintos componentes.

Los sistemas distribuidos incluyen una variedad de modelos, como sistemas de almacenamiento distribuido (como Google File System), bases de datos distribuidas (como Cassandra) y plataformas en la nube (como AWS y Google Cloud). una ventaja de los sistemas distribuidos es su capacidad para ofrecer alta disponibilidad y escalabilidad. Sin embargo, presentan desafíos en cuanto a consistencia de datos, seguridad y latencia de red.

Aunque ambos conceptos comparten la idea de distribución de recursos, sus enfoques difieren significativamente. El cómputo distribuido se enfoca en el procesamiento concurrente de tareas computacionales, los sistemas distribuidos engloban un ecosistema que involucra almacenamiento, comunicación y gestión de recursos. Además, el cómputo distribuido es utilizado en aplicaciones específicas de alto rendimiento, y los sistemas distribuidos abarcan una gama más amplia de aplicaciones, desde bases de datos hasta servicios en la nube.

Los ambientes distribuidos representan un paradigma fundamental en la computación actual, proporcionando soluciones escalables y eficientes a problemas complejos.

La distinción entre cómputo distribuido y sistemas distribuidos es crucial para comprender sus aplicaciones y optimizar su diseño e implementación. Mientras que el cómputo distribuido se orienta hacia el procesamiento paralelo, los sistemas distribuidos ofrecen una infraestructura integral para la gestión de recursos y servicios en red. El entendimiento de estos conceptos permitirá a los futuros profesionales de la computación diseñar soluciones más eficientes y resilientes para los desafíos del mundo digital.

Interfaces

Las interfaces en la arquitectura de software definen los contratos de comunicación entre diferentes módulos o sistemas. En el contexto de SOA, una interfaz es una especificación que expone las funcionalidades de un servicio sin revelar su implementación interna. Por otro lado, en el ámbito de las APIs, las interfaces permiten la interacción entre aplicaciones a través de protocolos estándar como REST, SOAP y GraphQL.

SOA se enfoca en la creación de servicios empresariales reutilizables que son orquestados para formar aplicaciones más complejas. Las APIs, en cambio, proporcionan un medio para que las aplicaciones interactúen con servicios y datos de manera más flexible y accesible.

La principal diferencia entre las interfaces en SOA y en APIs radica en su propósito y alcance.

Las interfaces en SOA son esenciales para garantizar la interoperabilidad entre distintos sistemas y plataformas. Gracias a la abstracción y el desacoplamiento que proporcionan, los servicios pueden evolucionar independientemente sin afectar a los consumidores permitiendo una mayor flexibilidad y adaptabilidad en entornos empresariales dinámicos. Además, las interfaces en SOA fomentan la reutilización de servicios, reduciendo la duplicación de código y optimizando los recursos de desarrollo. También facilitan la implementación de estándares y protocolos como SOAP y WSDL, asegurando una comunicación eficiente y estructurada entre los servicios.

Las APIs han evolucionado como un mecanismo clave para la integración de sistemas en entornos modernos. A diferencia de SOA, que se basa en protocolos más rígidos, las APIs ofrecen mayor flexibilidad mediante arquitecturas RESTful, que utilizan HTTP como protocolo de comunicación y JSON como formato de datos ligero.

REST, SOAP y GraphQL son ejemplos de enfoques utilizados para diseñar interfaces de APIs. REST se ha convertido en el estándar

predominante debido a su simplicidad y eficiencia, mientras que GraphQL permite consultas más dinámicas y personalizadas. La seguridad y gobernanza en APIs también juegan un papel crucial, garantizando la protección de datos y el control de acceso a los servicios.

Un ejemplo de la importancia de las interfaces en SOA y APIs es la transición de una empresa de servicios financieros desde una arquitectura monolítica a una basada en microservicios y APIs. Mediante la definición de interfaces bien estructuradas, la empresa logra mejorar la escalabilidad, reducir el tiempo de desarrollo y aumentar la interoperabilidad entre sus sistemas internos y externos.

Otro caso es el de una plataforma de comercio electrónico que expone sus funcionalidades a través de APIs públicas, permitiendo que terceros integren sus servicios y amplíen el ecosistema digital de la empresa.

Las interfaces desempeñan un rol fundamental en SOA y APIs al permitir la comunicación efectiva entre sistemas y la reutilización de componentes de software. Su correcta implementación contribuye a la escalabilidad, seguridad y eficiencia de las aplicaciones modernas. En un mundo cada vez más interconectado, el diseño y gestión adecuados de interfaces seguirán siendo una pieza clave en la evolución del desarrollo de software.

Así podemos concluir que la evolución de SOA y APIs refleja el avance continuo en la forma en que los sistemas de software se diseñan e integran. Mientras que SOA sentó las bases de la modularidad y la comunicación basada en servicios, las APIs han permitido la democratización del acceso a datos y funcionalidades, impulsando la transformación digital en diversos sectores. La convergencia de estos enfoques en arquitecturas modernas, como los microservicios, demuestra la importancia de la interoperabilidad y la flexibilidad en el desarrollo de software actual.

Referencias

- Erl, T. (2005). SERVICE-ORIENTED ARCHITECTURE: CONCEPTS, TECHNOLOGY, AND DESIGN. Prentice Hall.
- Fielding, R. (2000). ARCHITECTURAL STYLES AND THE DESIGN OF NETWORK-BASED SOFTWARE ARCHITECTURES. Doctoral Dissertation, University of California, Irvine.
- Fowler, M., & Lewis, J. (2014). MICROSERVICES: A DEFINITION OF THIS NEW ARCHITECTURAL TERM. ThoughtWorks.
- Newman, S. (2015). BUILDING MICROSERVICES: DESIGNING FINE-GRAINED SYSTEMS. O'Reilly Media.
- Papazoglou, M. P. (2008). WEB SERVICES: PRINCIPLES AND TECHNOLOGY. Pearson.
- Richardson, L., & Ruby, S. (2007). RESTFUL WEB SERVICES. O'Reilly Media.
- Erl, T. (2005). SOA: PRINCIPLES OF SERVICE DESIGN. Prentice Hall.
- Fielding, R. (2000). ARCHITECTURAL STYLES AND THE DESIGN OF NETWORK-BASED SOFTWARE ARCHITECTURES. Doctoral dissertation, University of California, Irvine.
- Berners-Lee, T. (1996). THE WORLD WIDE WEB: PAST, PRESENT AND FUTURE. World Wide Web Consortium (W3C).
- Krol, E. (1993). THE WHOLE INTERNET: USER'S GUIDE & CATALOG. O'Reilly Media.
- Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., ... & Wolff, S. (2009). A BRIEF HISTORY OF THE INTERNET. ACM SIGCOMM Computer Communication Review, 39(5), 22-31.
- Bass, L., Clements, P., & Kazman, R. (2012). SOFTWARE ARCHITECTURE IN PRACTICE. Addison-Wesley.
- Tanenbaum, A. S., & Van Steen, M. (2017). DISTRIBUTED SYSTEMS: PRINCIPLES AND PARADIGMS. Pearson.
- Comer, D. E. (2018). INTERNETWORKING WITH TCP/IP: PRINCIPLES, PROTOCOLS, AND ARCHITECTURE (6th ed.). Pearson.
- Stallings, W. (2020). CRYPTOGRAPHY AND NETWORK SECURITY: PRINCIPLES AND PRACTICE (8th ed.). Pearson.
- Rescorla, E. (2000). SSL AND TLS: DESIGNING AND BUILDING SECURE SYSTEMS. Addison-Wesley.
- Erl, T. (2005). SERVICE-ORIENTED ARCHITECTURE: CONCEPTS, TECHNOLOGY, AND DESIGN. Prentice Hall.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE. Addison-Wesley.
- Larman, C. (2004). APPLYING UML AND PATTERNS: AN INTRODUCTION TO OBJECT-ORIENTED ANALYSIS AND DESIGN AND ITERATIVE DEVELOPMENT. Prentice Hall.
- Meyer, B. (1997). OBJECT-ORIENTED SOFTWARE CONSTRUCTION. Prentice Hall.
- Tanenbaum, A. S., & Bos, H. (2014). MODERN OPERATING SYSTEMS. Pearson.
- Coulouris, G., Dollimore, J., & Kindberg, T. (2013). DISTRIBUTED SYSTEMS: CONCEPTS AND DESIGN (5th ed.). Pearson.

- Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. COMMUNICATIONS OF THE ACM, 51(1), 107-113.
- Erl, T. (2016). SERVICE-ORIENTED ARCHITECTURE: CONCEPTS, TECHNOLOGY, AND DESIGN. Pearson Education.
- Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures (Doctoral dissertation, University of California, Irvine).
- Fowler, M. (2019). PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE. Addison-Wesley Professional.
- Newman, S. (2021). BUILDING MICROSERVICES: DESIGNING FINE-GRAINED SYSTEMS. O'Reilly Media.
- Richardson, L., & Amundsen, M. (2020). RESTFUL WEB APIS. O'Reilly Media.
- Erl, T. (2017). SERVICE-ORIENTED ARCHITECTURE: CONCEPTS, TECHNOLOGY, AND DESIGN. Prentice Hall.
- Fielding, R. (2000). ARCHITECTURAL STYLES AND THE DESIGN OF NETWORK-BASED SOFTWARE ARCHITECTURES (Doctoral dissertation, University of California, Irvine).
- Josuttis, N. M. (2007). SOA IN PRACTICE: THE ART OF DISTRIBUTED SYSTEM DESIGN. O'Reilly Media.
- Khosrow-Pour, M. (Ed.). (2018). ENCYCLOPEDIA OF INFORMATION SCIENCE AND TECHNOLOGY. IGI Global.
- Papazoglou, M. P. (2012). WEB SERVICES AND SOA: PRINCIPLES AND TECHNOLOGY. Pearson Education.

CAPÍTULO 2

Diseño de Servicios

Creando APIs Efectivas

El objetivo de este capítulo es comprender principios fundamentales para el diseño de APIs, que faciliten la interoperabilidad y la reutilización de servicios.



El diseño de APIs es una de las piezas clave para asegurar una integración efectiva entre servicios y aplicaciones.

Una API bien diseñada no solo facilita la interacción entre sistemas, sino que también optimiza la usabilidad, mejora el rendimiento y garantiza la seguridad. Sin embargo, requiere una comprensión profunda de las mejores prácticas, los principios de diseño y las necesidades del consumidor de la API.

Este capítulo, hará énfasis en los principios esenciales para crear APIs efectivas, abordará temas como la elección adecuada de métodos HTTP, el uso de estándares de autenticación y la documentación clara, además, se explicará cómo construir APIs escalables y mantenibles, diseñadas para evolucionar junto con las necesidades del sistema.

API Interfaz de Programación de Aplicaciones, puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato , que es el que define cómo se comunican entre sí mediante solicitudes y respuestas.

Importancia de las APIs

En la era digital, las APIs juegan un papel crucial en la interoperabilidad entre sistemas y aplicaciones. Su correcto diseño no solo facilita la integración de servicios, sino que también garantiza la seguridad, eficiencia y mejora el mantenimiento del software, por consiguiente puede facilitar la integración entre diferentes sistemas, promover la reutilización de código y mejorar la experiencia del usuario.

Sin embargo, para lograr estos beneficios, es fundamental seguir una serie de principios y mejores prácticas, como requisitos esenciales para la creación de APIs

1. **Definición clara de objetivos y alcance.** Antes de iniciar el desarrollo de una API, es fundamental definir claramente sus objetivos y alcance. Esto implica identificar el público objetivo de la API, los servicios que ofrecerá y las funcionalidades que expondrá. Una definición clara de los objetivos permitirá establecer una hoja de ruta para el desarrollo y garantizar que la API cumpla con las necesidades de sus usuarios (Richardson & Ruby, 2013).
2. **Diseño centrado en el usuario.** El diseño de una API efectiva debe centrarse en la experiencia del usuario, es decir, en cómo los desarrolladores interactúan con ella. Esto implica utilizar convenciones de nomenclatura claras y coherentes, proporcionar documentación completa y precisa, y garantizar que la API sea fácil de usar e intuitiva. Un diseño centrado en el usuario fomentará la adopción y el uso de la API (Red Hat, 2023).
3. **Arquitectura robusta y escalable.** La arquitectura de una API debe ser robusta y escalable para garantizar su rendimiento y disponibilidad. Esto implica utilizar tecnologías y patrones de diseño adecuados, implementar medidas de seguridad para proteger los datos y garantizar que la API pueda

manejar un volumen creciente de solicitudes. Una arquitectura sólida permitirá que la API evolucione y se adapte a las necesidades futuras (Aster Software, 2023).

4. **Seguridad.** La seguridad es un aspecto fundamental en la creación de APIs efectivas. Se deben implementar mecanismos de autenticación y autorización para proteger los datos y garantizar que solo los usuarios autorizados puedan acceder a los recursos de la API. Además, se deben tomar medidas para prevenir ataques comunes, como la inyección de SQL y las vulnerabilidades de cross-site scripting (OWASP, 2021).
5. **Documentación completa y actualizada.** La documentación es un elemento clave para el éxito de una API. Debe ser completa, precisa y estar actualizada. La documentación debe incluir información sobre los endpoints de la API, los parámetros de entrada y salida, los códigos de error y ejemplos de uso. Una documentación clara y completa facilitará el uso de la API y reducirá la necesidad de soporte (Belvo, 2024).
6. **Pruebas exhaustivas.** Antes de lanzar una API, es fundamental realizar pruebas exhaustivas para garantizar su calidad y funcionamiento. Esto implica realizar pruebas unitarias, pruebas de integración y pruebas de rendimiento. Las pruebas permitirán identificar y corregir errores antes de que la API sea utilizada por los usuarios (Bambu Mobile, 2023).
7. **Versiónamiento.** A medida que una API evoluciona, es importante implementar un sistema de versionamiento. Esto permitirá mantener la compatibilidad con versiones anteriores y evitar que los cambios en la API afecten a los usuarios existentes. El versionamiento también facilitará la introducción de nuevas funcionalidades y mejoras (Aster Software, 2023).
8. **Monitoreo y mantenimiento.** Una vez que una API ha sido lanzada, es importante monitorearla y mantenerla. Esto implica realizar un seguimiento del rendimiento, identificar y corregir errores, y actualizar la documentación. El monitoreo y el mantenimiento garantizarán que la API siga funcionando de manera óptima y satisfaciendo las necesidades de sus usuarios.

Las Interfaces de Programación de Aplicaciones (APIs) son herramientas fundamentales en el desarrollo de software moderno, permitiendo la comunicación e intercambio de datos entre diferentes sistemas o aplicaciones y existen diversos tipos de APIs, cada una con características y funcionalidades específicas,

Clasificación de APIs:

1. Según su acceso.
 - a) APIs públicas (abiertas): Son accesibles para cualquier persona o empresa que desee utilizarlas, sin necesidad de autorización previa. Suelen ser gratuitas y están documentadas para facilitar su uso. Ejemplos: APIs de redes sociales (como Twitter API), APIs de mapas (como Google Maps Platform), APIs de información meteorológica (como OpenWeatherMap API). Musser, G. L., & Turner, R. (2018). RESTful web APIs. O'Reilly Media.
 - b) APIs privadas (internas): Son utilizadas internamente por una empresa u organización para conectar sus propios sistemas y aplicaciones. No están disponibles para el público externo y suelen requerir autenticación para acceder a ellas. Fowler, M. (2018). Microservices. Addison-Wesley Professional.

- c) APIs de partners (asociadas): Son compartidas con socios comerciales o desarrolladores autorizados que tienen una relación comercial con la empresa u organización. Suelen requerir un acuerdo o contrato para acceder a ellas. Richardson, L., & Ruby, S. (2013). RESTful web services. O'Reilly Media.
- d) APIs compuestas: Combinan varias APIs diferentes para ofrecer una funcionalidad más completa o un servicio más complejo.

2. Según su arquitectura

- a) APIs REST (Representational State Transfer): Son las más utilizadas actualmente. Se basan en el protocolo HTTP y utilizan métodos como GET, POST, PUT y DELETE para interactuar con los recursos. Son ligeras, escalables y fáciles de usar. Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. University of California, Irvine.
- b) APIs SOAP (Simple Object Access Protocol): Utilizan el protocolo SOAP para intercambiar mensajes en formato XML. Son más complejas que las APIs REST y suelen utilizarse en entornos empresariales donde la seguridad y la fiabilidad son importantes. W3C. (2007). SOAP Version 1.2 Part 1: Messaging Framework.
- c) APIs RPC (Remote Procedure Call): Permiten ejecutar funciones o procedimientos en un servidor remoto. Son más antiguas que las APIs REST y SOAP, pero aún se utilizan en algunos casos.
- d) APIs GraphQL: Permiten a los clientes solicitar datos específicos de un servidor, en lugar de recibir todos los datos disponibles. Son más eficientes que las APIs REST en algunos casos, ya que reducen la cantidad de datos transferidos.
- e) APIs WebSocket: Permiten la comunicación bidireccional en tiempo real entre el cliente y el servidor. Son ideales para aplicaciones que requieren actualizaciones instantáneas, como chats, juegos en línea y aplicaciones de colaboración. IETF. (2014). The WebSocket Protocol.

3. Según su funcionalidad

- a) APIs de datos: Permiten acceder a datos específicos, como información de usuarios, productos, transacciones, etc.
- b) APIs de servicios: Permiten acceder a funcionalidades específicas, como procesamiento de pagos, envío de correos electrónicos, generación de informes, etc.
- c) APIs de dispositivos: Permiten interactuar con dispositivos físicos, como sensores, actuadores, cámaras, etc.
- d) APIs de inteligencia artificial: Permiten acceder a modelos de aprendizaje automático y funcionalidades de IA, como reconocimiento de voz, procesamiento de lenguaje natural, visión por computadora, etc.

Existen otros tipos de APIs. La elección del tipo de API adecuado dependerá de las necesidades específicas del proyecto, los requisitos de seguridad, la escalabilidad y otros factores.

Por otro lado existen herramientas para hacer pruebas de APIs antes de ponerlas en producción, algunas de las cuales son:



Postman: Una de las herramientas más populares y utilizadas para probar APIs REST. Ofrece una interfaz intuitiva y fácil de usar, lo que la hace ideal tanto para principiantes como para usuarios avanzados. Postman permite crear y enviar solicitudes HTTP, validar respuestas, automatizar pruebas y documentar APIs. www.postman.com



SoapUI: Una herramienta potente y versátil que admite pruebas de APIs REST y SOAP. SoapUI ofrece una amplia gama de funciones, incluyendo pruebas funcionales, pruebas de rendimiento, pruebas de seguridad y simulación de servicios. www.soapui.org



Rest Assured: Una biblioteca Java que simplifica las pruebas de APIs REST. Rest Assured proporciona un lenguaje sencillo y expresivo para escribir pruebas y se integra fácilmente con otras herramientas de prueba. www.jadedglobal.com



JMeter: Una herramienta de código abierto utilizada principalmente para pruebas de rendimiento y carga. JMeter puede simular múltiples usuarios y solicitudes para evaluar el rendimiento de una API bajo diferentes condiciones. www.edureka.co



Insomnia: Una alternativa a Postman que ofrece una interfaz moderna y fácil de usar. Insomnia admite pruebas de APIs REST, GraphQL y gRPC. docs.insomnia.rest

Algunas son más adecuadas para probar APIs REST, mientras que otras son más adecuadas para APIs SOAP o GraphQL, así que se debe elegir una herramienta que sea compatible con el tipo de API que se está probando, tomando en cuenta:

1. **Funcionalidades:** como pruebas de rendimiento, pruebas de seguridad y simulación de servicios, mientras que otras se centran en pruebas funcionales básicas. Es importante elegir una herramienta que ofrezca las funcionalidades necesarias para el proyecto.
2. **Facilidad de uso:** Es importante elegir una herramienta que sea cómoda para el equipo de desarrollo y que no requiera una curva de aprendizaje pronunciada.
3. **Integración:** Algunas herramientas se integran con otras herramientas de desarrollo, como herramientas de CI/CD y herramientas de gestión de pruebas. Es importante elegir una herramienta que se integre con el flujo de trabajo existente.

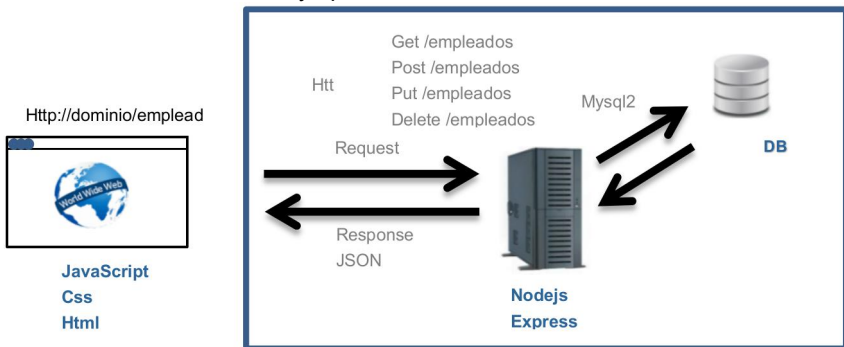
Diseñando de una api rest simple

Para entender el funcionamiento de lo que representa una API en este punto se plasmará el diseño de una api rest simple, contemplando algunos de los principios en la implementación como lo es la arquitectura escalable, la interacción con el usuario, las convenciones claras y la seguridad entre otras.

Dado lo anterior lo siguiente será una creación e implementación de *API Rest simple* a manera de ejemplo, si el lector la implementa siguiendo la explicación seguramente entenderá mas claramente la interoperabilidad y reutilización en el diseño de APIs.

Las herramientas y tecnologías:

Lenguaje	Javascript
Framework	NodeJs
Editor	Visual Studio Code
Motor de datos	Mysql.



<<Esquema - API utilizando base de datos con tabla empleados>>

Primero Instalar y configurar las herramientas (puede ser localmente), VScode, NodeJs y Mysql.

Consideraciones importantes:

JavaScript es un lenguaje interpretado que puede ser utilizado para diversos propósitos, pero que se considera muchas veces como solo un complemento de HTML y CSS. los nuevos motores de procesamiento incorporados por las recientes versiones de los navegadores más populares fueron desarrollados para acelerar el procesamiento del código Javascript convirtiéndolo en código máquina y así alcanzar velocidades de ejecución similares a las aplicaciones de escritorio.

Framework es una estructura o conjunto de componentes que sirve de base para la construcción de aplicaciones, ofrece una serie de funciones y elementos comunes que se utilizan con frecuencia en el desarrollo web,

se clasifican según el tipo de lenguaje de programación que utilizan o el lado del desarrollo web al que se enfocan.

1. **Back End:** facilitan desarrollo del lado del servidor, es la parte de la aplicación web que se encarga de la lógica de negocio, gestionar las bases de datos y comunicarse con el cliente, usan lenguajes de programación como Python, Ruby, PHP, Javascript, Java, etc.
2. **Front End:** aumentan funcionalidad de aplicaciones web del lado del cliente, es la parte de la aplicación web que se encarga de presentar la interfaz de usuario y responder a sus acciones, usan lenguajes de programación como HTML, CSS y Javascript. (Bootstrap y React son unos ejemplos de estos Frameworks).
3. **Full Stack:** Están diseñados para abarcar tanto el desarrollo del lado del servidor como el del cliente, integrando varios Frameworks de Back End y Front End, usan lenguajes de programación como Javascript, Python, Ruby, Node.js, etc.

Visual Studio Code es un editor de código re-definido y optimizado para crear y depurar aplicaciones web y en la nube modernas.

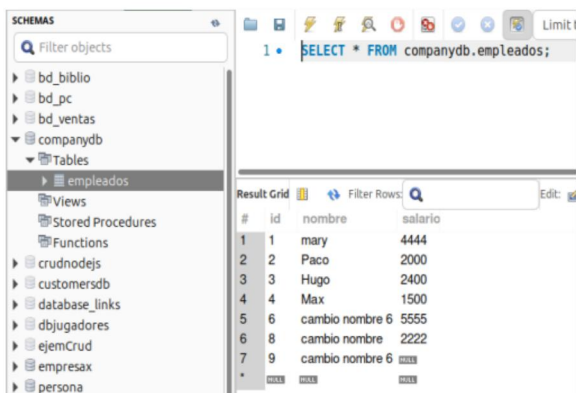


Nodejs es un entorno que trabaja en tiempo de ejecución, de código abierto, multiplataforma, que permite a los desarrolladores crear toda clase de herramientas de lado servidor y aplicaciones en JavaScript.



npm(Node Package Manager) es un gestor de paquetes que permite crear los archivos necesarios para gestionar los desarrollos con Node o JavaScript en general, viene incluido y genera un archivo llamado **package.json**, que contiene todos los metadatos acerca del proyecto tal como descripción, licencia, dependencias y scripts.

Antes de iniciar, se crea la base de datos “**companydb**” con la tabla “**empleados**”, y tres columnas “**id**, **nombre**, **salario**”, en Mysql para usarla en el diseño



Creación del servidor WEB

Se crea un folder para el proyecto, y se abre VScode y en una terminal se inicializa el proyecto ejecutando **npm init -y** para crear el archivo **package.json**.

A medida que un aplicación crece, se divide en múltiples archivos, llamados “módulos”. Un módulo puede contener una clase o una biblioteca de funciones para un propósito específico. Cuando instalamos paquetes o dependencias mediante npm, se crea un directorio en la carpeta raíz de nuestro proyecto con el nombre **node_modules** y el archivo **package-lock.json**.

Se instalan módulos: con **npm i nodemon (-D), express, mysql2**.

En archivo **package.json** se agrega la propiedad “**type**”, para poder importar módulos, y los scripts “**start** y **dev**” para desarrollo y producción.

```
"type": "module",

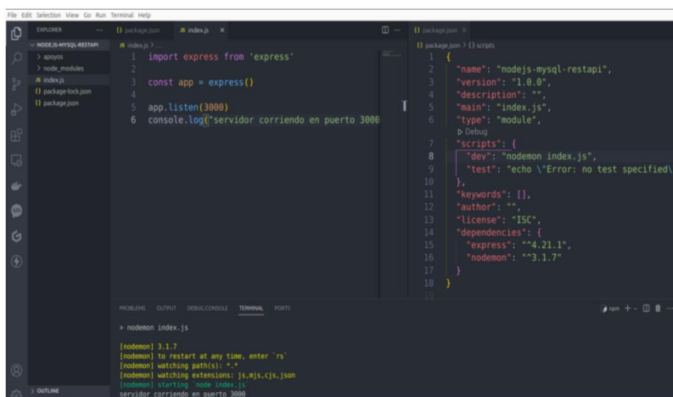
"scripts": {
  "start": "node ./src/index.js",
  "dev": "nodemon ./src/index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Se crea un archivo **index.js** con el código:

```
import express from 'express'
const app = express()

app.listen(3000)
console.log("servidor corriendo en puerto 3000")
```

En terminal se ejecuta **npm run dev**.



Hasta aquí el servidor listo escuchando en el puerto 3000 !!.

Ahora se Inicia el diseño de API's (endpoints)

***Endpoint**, conjunto de protocolos y reglas que definen cómo un cliente API (el software que realiza una solicitud) puede comunicarse con un servidor para acceder a datos y funciones*

En archivo **index.js** se agrega el código para las consultas desde los clientes.

```
import express from 'express'
const app = express()

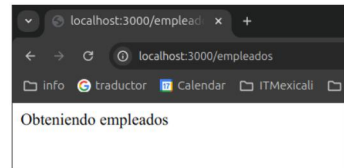
// endpoints
app.get('/empleados', (req, res) => res.send("Obteniendo empleados"))
app.post('/empleados', (req, res) => res.send("creando empleado"))
app.put('/empleados', (req, res) => res.send("actualizando empleado"))
app.delete('/empleados', (req, res) => res.send("eliminando empleado"))

app.listen(3000)
console.log("servidor corriendo en puerto 3000")
```

Los principales métodos soportados por HTTP y por ello usados por una API REST son:

- POST: crear un recurso nuevo.
- PUT: modificar un recurso existente.
- GET: consultar información de un recurso.
- DELETE: eliminar un recurso determinado.
- PATCH: modificar solamente un atributo de un recurso.

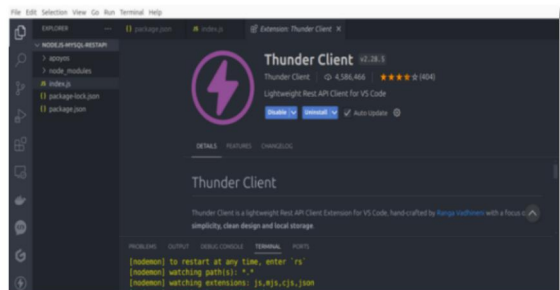
Ya se puede probar la ruta **“/empleados”** con el método **GET**

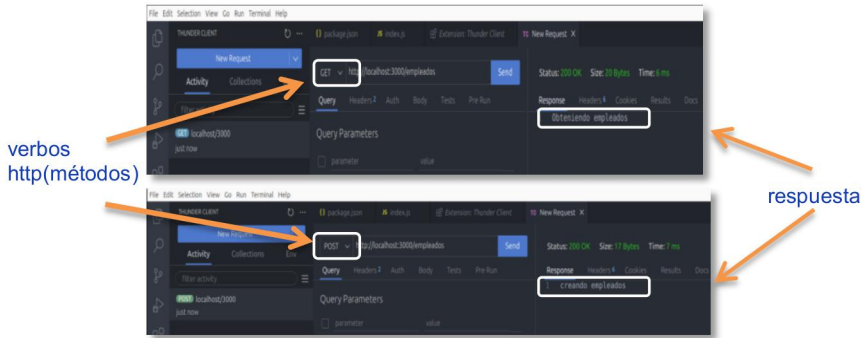


Los otros endpoint's no se pueden enviar en la url, por lo que se usa un programa (rest cliente), postman, thunder, etc....

En este caso se instala la extensión de **Thunder Client** en vscode, para probar las api's

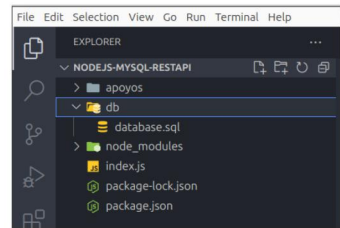
Se simulan peticiones desde una aplicación (móvil o web), y ya que estén listos en desarrollo se podrían desplegar para su consumo.





Ahora se agrega la conexión a la base de datos, se organiza y se estructura el código en capas.

Se crea folder (**db**) para tener el sql de la base de datos y dentro se crea el archivo **database.sql** y se agrega el código que se usará.



```
CREATE DATABASE IF NOT EXISTS companydb;
```

```
use companydb;
```

```
CREATE TABLE empleados(
  id INT(11) NOT NULL AUTO_INCREMENT,
  nombre VARCHAR(45) DEFAULT NULL,
  salario INT DEFAULT NULL,
  PRIMARY KEY (id)
);
```

```
DESCRIBE empleados;
```

En mysql se visualiza ...

```
mysql> use companydb
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> describe empleados;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id    | int           | NO   | PRI | NULL    | auto_increment |
| nombre | varchar(45)   | YES  |     | NULL    |                |
| salario | int          | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

Para probar la conexión se usará un pool de conexiones y una ruta **“/ping”** con el método GET, y se utilizarán promesas para pausar por lo asíncrono en la función.

Se agrega el código en archivo **db.js**

```
import { createPool } from 'mysql2/promise'

export const pool = createPool({
  host: 'localhost',
  user: 'root',
  password: 'root',
  port: '3306',
  database: 'companydb'
})
```

Se agrega el código para la consulta de prueba desde los clientes.

```
index.js
import express from 'express'
import { pool } from './db.js';

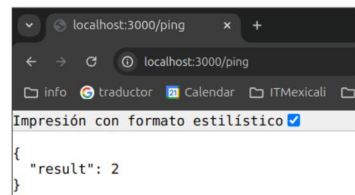
const app = express()

//consulta de prueba
app.get('/ping', async (req, res) => {
  const [result] = await pool.query('SELECT 1+1 AS result')
  res.json(result[0])
});

//api's o endpoints
app.get('/empleados', (req, res) => res.send("Obteniendo empleados"))
app.post('/empleados', (req, res) => res.send("creando empleado"))
app.put('/empleados', (req, res) => res.send("actualizando empleado"))
app.delete('/empleados', (req, res) => res.send("eliminando empleado"))

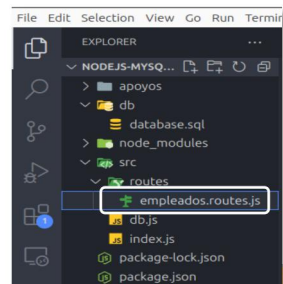
app.listen(3000)
console.log("servidor corriendo en puerto 3000")
```

Se prueba y visualizamos el resultado.



Ahora se inicia la estructura creando el folder **src** y dentro un folder **routes** para rutas de las APIs

La lógica de negocio, se divide en múltiples archivos para después importarlos desde **index.js**, por lo que se crea un archivo para las rutas dentro del folder routes.



Archivo `empleados.routes.js` para las rutas

```
// de express solo importamos router
import { Router } from "express"

//se crea el enrutador
const router = Router()

// traemos las rutas que estaban en index.js, y en el index se importaran
// Indican que cuando se visiten las rutas se ejecutan las funciones
router.get('/empleados', getEmpleados)
router.get('/empleados/:id', getEmpleado)
router.post('/empleados', createEmpleado)
router.patch('/empleados/:id', updateEmpleado)
router.delete('/empleados/:id', deleteEmpleado)

// y se exporta
export default router
```

En `index.js` se importan las rutas.

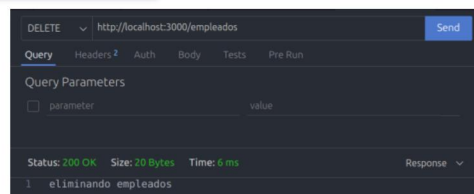
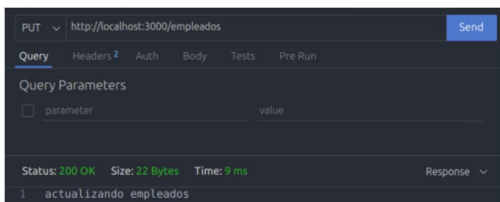
```
import express from 'express'
import { pool } from './db.js'
// Se importan las rutas
import empleadosRoutes from './routes/empleados.routes.js'

const app = express()

app.get('/ping', async (req, res) => {
  const [result] = await pool.query('SELECT 1+1 AS result')
  res.json(result[0])
});
//se indica el uso de las rutas
app.use(empleadosRoutes)

app.listen(3000)
console.log("servidor corriendo en puerto 3000")
```

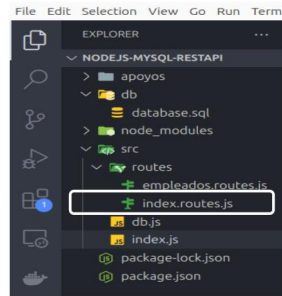
Prueba con Thunder y siguen funcionando



Se hace lo mismo con la **ruta de prueba “/ping”**

Se crea el archivo para esa ruta **index.routes.js**

Se pasa la ruta de prueba **“/ping”** que estaba en **index.js** a **routes.js** y se cambia **app.get** por **router.get**, y pasamos también la importación de **pool** dado que aquí lo usaremos y no en **index.js**



index.routes.js

```
import { Router } from "express"
import { pool } from '../db.js'
```

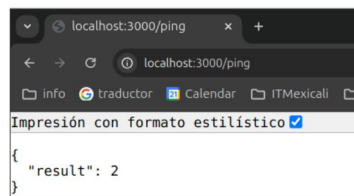
```
const router = Router();
```

//consulta de prueba

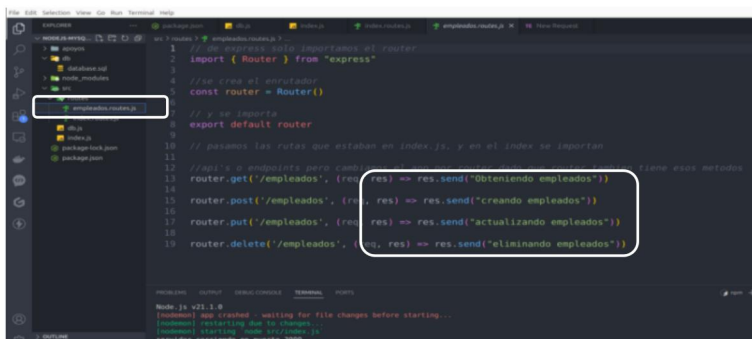
```
router.get('/ping', async (req, res) => {
  const [result] = await pool.query("SELECT 1+1 AS result")
  res.json(result[0])
});
```

```
export default router
```

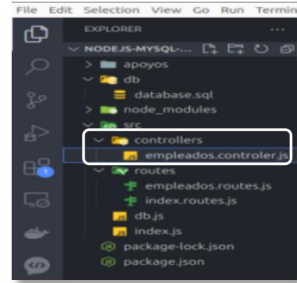
Se prueba y visualizamos el resultado.



En **empleados.routes.js** esta la lógica cuando esta debería estar en el controlador.



Para organizar la lógica se crea un folder para controladores y el archivo controlador de empleados para las consultas, validaciones, etc...



Se agregan las funciones que estaban en **empleados.routes.js** en **empleados.controllers.js** y se exportan

empleados.controllers.js

```
export const getEmpleados = (req, res) => res.send("obteniendo empleados")
export const createEmpleados = (req, res) => res.send("creando empleados")
export const updateEmpleados = (req, res) => res.send("actualizar empleados")
export const deleteEmpleados = (req, res) => res.send("eliminar empleados")
```

Se importan esas funciones

empleados.routes.js

// de express solo importamos el router

```
import { Router } from "express"
```

```
import { getEmpleados, createEmpleados, updateEmpleados, deleteEmpleados } from
'../controllers/empleados.controller.js'
```

//se crea el enrutador

```
const router = Router()
```

// traemos las rutas que estaban en index.js, y en el index se importan

// Indican que cuando se visiten las rutas se ejecutan las funciones

```
router.get('/empleados', getEmpleados)
```

```
router.post('/empleados', createEmpleados)
```

```
router.put('/empleados', updateEmpleados)
```

```
router.delete('/empleados', deleteEmpleados)
```

// y se exporta

```
export default router
```

Ahora con la **ruta de prueba "/ping"**. En **index.controllers.js** se importa el pool que esta en **index.js** y se exporta la función ping.

index.controllers.js

// se importa el pool, porque aqui se requiere

```
import { pool } from './db.js'
```

//traemos la funcion get que estaba en index.routes.js y se exporta

```
export const ping = async (req, res) => {
  const [result] = await pool.query("SELECT 1+1 AS result")
  res.json(result[0])
}
```

Se importa la función ping y se invoca en la ruta con GET.

index.routes.js

```
import { Router } from "express"
import { ping } from "../controllers/index.controller.js"
```

```
const router = Router()
router.get('/ping', ping)
```

```
export default router
```

Lo siguiente, será estructurar e implementar la lógica de negocio, iniciando con el **POST** y el **GET** empleados.

Se requiere usar la función de **json() de express** en **index.js** para probar desde Thunder

```
import express from 'express'
// Se importan las rutas
import empleadosRoutes from './routes/empleados.routes.js'
import indexRoutes from './routes/index.routes.js'
```

```
const app = express()
app.use(express.json())
```

```
//se indica el uso de las rutas y se recomienda anteponer el prefijo /api a todas las rutas
```

```
app.use('/api', empleadosRoutes)
app.use('/api', indexRoutes)
```

```
app.listen(3000)
console.log("servidor corriendo en puerto 3000")
```

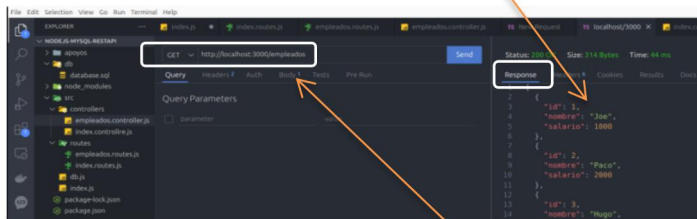
En controlador de empleados, se implementa la lógica para consultar empleados **getEmpleados** y se importa el pool

empleados.controllers.js (consulta empleados)

```
import { pool } from '../db.js'
```

```
//traemos las funciones que estaban en empleados.routes.js, y se exportan
```

```
export const getEmpleados = async (req, res) => {
  const [rows] = await pool.query('SELECT * FROM empleados')
  res.send(rows)
}
```



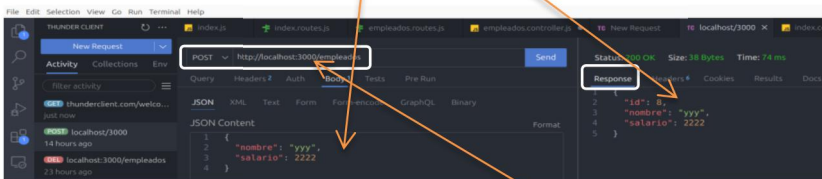
En **empleados.routes.js** esta la ruta **router.get('/empleados', getEmpleados)** y llama la función para consultar empleados

Ahora en el mismo archivo de controlador de empleados, se implementa la lógica para crear empleados `createEmpleados` seguido del código para consultar

empleados.controllers.js (crear empleados)

```
export const createEmpleados = async (req, res) => {
  const { nombre, salario } = req.body
  //aquí se pueden validar los datos...
  const [rows] = await pool.query('INSERT INTO empleados (nombre, salario)
VALUES (?, ?)', [nombre, salario])
  //res.send({ rows })
  res.send({
    id: rows.insertId,
    nombre,
    salario,
  })
}

export const updateEmpleados = (req, res) => res.send("actualizar empleados")
export const deleteEmpleados = (req, res) => res.send("eliminar empleados")
```



En `empleados.routes.js` esta la ruta **`router.post('/empleados', createEmpleados)`** y llama la función para crear empleados.

Para ubicar **un solo empleado** se implementa la lógica de negocio (**GET**) usando el ID agregando la ruta e importando su función, que sería `getEmpleado`, esto servirá mas adelante para actualizar o borrar un empleado.

```
// de express solo importamos el router
import { Router } from "express"
import { getEmpleados, createEmpleados, updateEmpleados, deleteEmpleados,
getEmpleado } from '../controllers/empleados.controller.js'
```

```
//se crea el enrutador
const router = Router()
```

```
// traemos las rutas que estaban en index.js, y en el index se importan
// Indican que cuando se visiten las rutas se ejecutan las funciones
router.get('/empleados', getEmpleados)
router.get('/empleados/:id', getEmpleado)
router.post('/empleados', createEmpleados)
router.put('/empleados', updateEmpleados)
router.delete('/empleados', deleteEmpleados)
```

```
// y se exporta
export default router
```

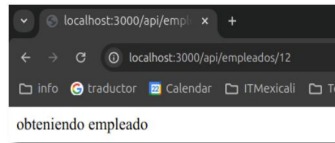
Nuevamente en el mismo archivo de controlador de empleados, se implementa la lógica para consultar un empleado `getEmpleado`

empleados.controllers.js (consulta empleado)

```
import { pool } from './db.js'
```

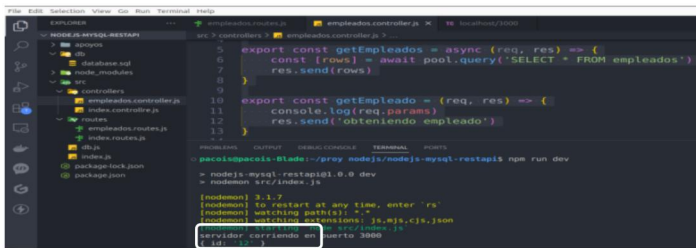
```
//traemos las funciones que estaban en empleados.routes.js, y se exportan
export const getEmpleados = async (req, res) => {
  const [rows] = await pool.query('SELECT * FROM empleados')
  res.send(rows)
}
```

```
export const getEmpleado = (req, res) => {
  console.log(req.params)
  res.send('obteniendo empleado')
}
```



Cuando se llama al API

Si se ejecuta en consola



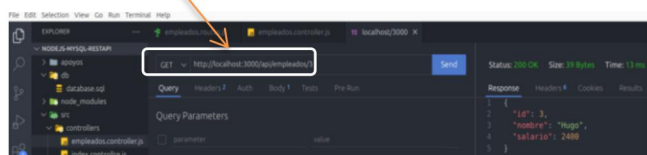
Ahora se implementa una consulta a DB (un solo empleado)

empleados.controllers.js

```
export const getEmpleados = async (req, res) => {
  const [rows] = await pool.query('SELECT * FROM empleados')
  res.send(rows)
}
```

```
export const getEmpleado = async (req, res) => {
  const [rows] = await pool.query('SELECT * FROM empleados WHERE id = ?',
  [req.params.id])
  if (rows.length <= 0) return res.status(404).json({ message: 'Empleado no
  encontrado' })
  res.json(rows[0])
}
```

Cuando se llama al API



Ahora se estructura e implementa la lógica de negocio (**DELETE**) para eliminar empleados

empleadosControllers.js

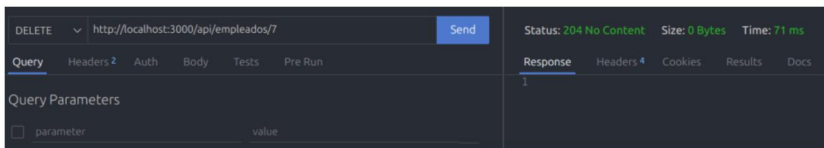
```
export const deleteEmpleados = async (req, res) => {
  const [result] = await pool.query('DELETE FROM empleados WHERE id = ?',
    [req.params.id])
  if (result.affectedRows <= 0) return res.status(404).json({ message: 'Empleado no encontrado' })

  res.send(204)
}
```

Si se consulta empleados, para eliminar el "id 7"

```
mysql> select * from empleados
+----+-----+-----+
| id | nombre | salario |
+----+-----+-----+
| 1  | Joe    | 1000    |
| 2  | Paco   | 2000    |
| 3  | Hugo   | 2400    |
| 4  | Max    | 1500    |
| 5  | xxx    | 5555    |
| 7  | yyy    | 2222    |
+----+-----+-----+
7 rows in set (0.00 sec)
```

Se llama la función en Thunder



Si se consulta nuevamente empleados, ya no existe "id 7"

```
mysql> select * from empleados
+----+-----+-----+
| id | nombre | salario |
+----+-----+-----+
| 1  | Joe    | 1000    |
| 2  | Paco   | 2000    |
| 3  | Hugo   | 2400    |
| 4  | Max    | 1500    |
| 5  | xxx    | 5555    |
| 8  | yyy    | 2222    |
+----+-----+-----+
6 rows in set (0.00 sec)
```

El siguiente a estructurar e implementar será la lógica de negocio (**PUT**) para **actualizar** empleados, ya con la consulta implementada

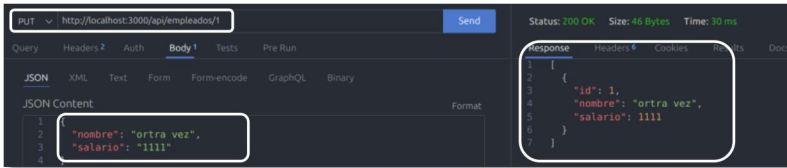
empleadosControllers.js

```
export const updateEmpleados = async (req, res) => {
  const { id } = req.params // parametro en la URL --> equivale a (const id = req.params.id)
  const { nombre, salario } = req.body // datos en formulario
  const [result] = await pool.query('UPDATE empleados SET nombre = ?, salario = ? WHERE id = ?', [nombre, salario, id])
  console.log(result)
  if (result.affectedRows <= 0) return res.status(404).json({ message: 'Empleado no encontrado' })
  const [rows] = await pool.query('SELECT * FROM empleados WHERE id = ?', [id])
  // res.json(rows) -- devuelve un arreglo
  res.json(rows[0]) // devuelve un objeto
}
```

```
ResultSetHeader {
  fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  info: 'Rows matched: 1 Changed: 1 Warnings: 0',
  serverStatus: 2,
  warningStatus: 0,
  changedRows: 1
}
```

```
mysql> select * from empleados
+----+-----+-----+
| id | nombre | salario |
+----+-----+-----+
| 1  | otra vez | 1111    |
| 2  | Paco   | 2000    |
| 3  | Hugo   | 2400    |
| 4  | Max    | 1500    |
| 5  | xxx    | 5555    |
| 8  | yyy    | 2222    |
+----+-----+-----+
6 rows in set (0.00 sec)
```

Cuando se llama al API



El último a estructurar e implementar es la lógica de negocio (**PATCH**) para **actualizar** empleados parcialmente, ya con la consulta implementada, en la ruta cambiamos el metodo put por patch

empleadosRoutes.js

```
router.get('/empleados', getEmpleados)
router.get('/empleados/:id', getEmpleado)
router.post('/empleados', createEmpleados)
router.patch('/empleados', updateEmpleados)
router.delete('/empleados', deleteEmpleados)
```

empleadosControllers.js

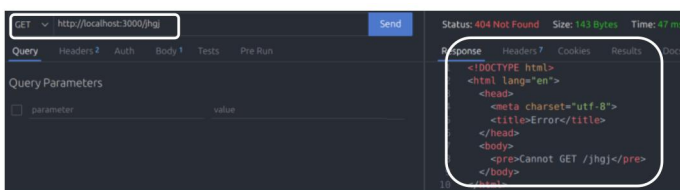
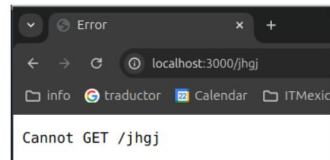
```
export const updateEmpleados = async (req, res) => {
  const { id } = req.params; // parametro en la URL --> equivale a (const id = req.params.id)
  const { nombre, salario } = req.body; // datos en formulario
  // IFNULL permite que se quede el mismo dato
  const [result] = await pool.query(
    "UPDATE empleados SET nombre = IFNULL(?, nombre), salario = IFNULL(?, salario) WHERE id = ?", [nombre, salario, id]
  );
  console.log(result);

  if (result.affectedRows === 0)
    return res.status(404).json({ message: "Empleado no encontrado" });

  const [rows] = await pool.query("SELECT * FROM empleados WHERE id = ?", [id]);

  // res.json(rows) -- devuelve un arreglo
  res.json(rows[0]); // devuelve un objeto
};
```

Si no se encuentra una ruta, la respuesta será un html (Cannot GET /jhgj)



Para manejar errores, se agrega una función middleware en index.js para respuesta cuando no hay ruta

index.js

```
import express from 'express'
```

```
// Se importan las rutas
```

```
import empleadosRoutes from './routes/empleados.routes.js'
```

```
import indexRoutes from './routes/index.routes.js'
```

```
const app = express()
```

```
app.use(express.json())
```

```
//se indica el uso de las rutas y se recomienda anteponer el prefijo /api a todas las rutas
```

```
app.use('/api', empleadosRoutes)
```

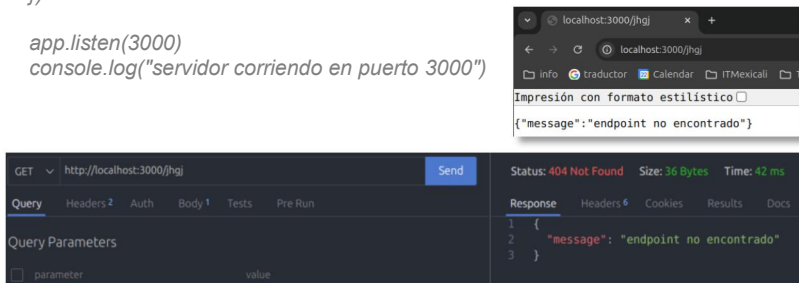
```
app.use('/api', indexRoutes)
```

```
//funcion middleware, por si no encuentra nada en la ruta (404)
```

```
app.use((req, res, next) => {res.status(404).json({message: 'endpoint no encontrado'})})
```

```
app.listen(3000)
```

```
console.log("servidor corriendo en puerto 3000")
```

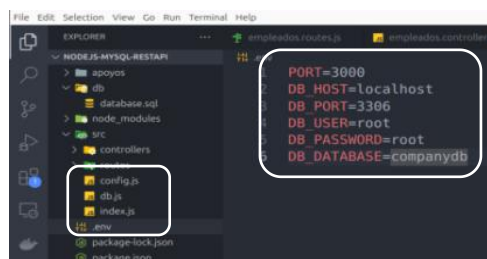


Hasta aquí el API y las rutas están listos para que clientes los llamen y son escuchados en el puerto 3000 !!.

La preparación adecuada del entorno de producción es una inversión esencial para garantizar el éxito a largo plazo de una aplicación. Permite minimizar riesgos, optimizar el rendimiento, mejorar la seguridad y facilitar el mantenimiento y actualización, lo que se traduce en una mejor experiencia para los usuarios y una mayor confianza en la aplicación.

Dado lo anterior se prepara lo básico en un entorno para producción, usando variables de entorno, quitándolas de la programación y almacenarlas en un archivo.

Primero se instala módulo **dotenv** y creamos los archivos **.env** para las variables de entorno y **config.js** para usar las variables y exportarlas.



Si las mostramos en consola

```

1 import { config } from "dotenv"
2
3 config()
4
5 console.log(process.env.PORT)
6 console.log(process.env.DB_HOST)
7 console.log(process.env.DB_USER)
8 console.log(process.env.DB_PASSWORD)
9 console.log(process.env.DB_DATABASE)
10 // process = objeto global de node
11 // env (almacena las variables del host)
12 // PORT, DB_HOST etc... con las variables

```

Se verá

```

[nodemon] starting `node src/index.js`
3000
localhost
root
root
companydb
servidor corriendo en puerto 3000

```

Se importan las variables de entorno desde dotenv y se exportan como constantes

config.js

```
import { config } from "dotenv"
```

```
config()
```

```
//exportando constantes con las variables de entorno
```

```
export const PORT = process.env.PORT || 3000
```

```
export const DB_HOST = process.env.DB_HOST || localhost
```

```
export const DB_PORT = process.env.DB_PORT || 3306
```

```
export const DB_PASSWORD = process.env.DB_PASSWORD || root
```

```
export const DB_USER = process.env.DB_USER || root
```

```
export const DB_DATABASE = process.env.DB_DATABASE || companydb
```

Se importan las variables de entorno desde **config.js** y se exporta el pool con esas variables

config.js

```
import { createPool } from 'mysql2/promise'
```

```
import {
```

```
  DB_HOST,
```

```
  DB_PORT,
```

```
  DB_USER,
```

```
  DB_PASSWORD,
```

```
  DB_DATABASE
```

```
} from './config.js'
```

```
export const pool = createPool({
```

```
  host: DB_HOST,
```

```
  user: DB_USER,
```

```
  password: DB_PASSWORD,
```

```
  port: DB_PORT,
```

```
  database: DB_DATABASE
```

```
})
```

Debería seguir funcionando

```

{
  "result": 2
}

```

Una **buena práctica** es no colocar toda la aplicación en el **index.js**, por si después se requiere trabajar mas módulos, por lo que se crea el archivo **app.js** y se le pasa el código de **index.js** sin la inicialización.

app.js

```
import express from 'express'
// Se importan las rutas
import empleadosRoutes from './routes/empleados.routes.js'
import indexRoutes from './routes/index.routes.js'

const app = express()

app.use(express.json())

//se indica el uso de las rutas
app.use('/api', empleadosRoutes)
app.use(indexRoutes)

//funcion middleware, por si no encuentra nada en la ruta (404)
app.use((req, res, next) => {
  res.status(404).json({
    message: 'endpoint no encontrado'
  })
})
export default app
```

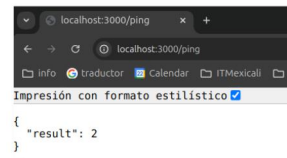
En el archivo **index.js** se importa **app** y se inicializa con **app.listen(3000)**.

index.js

```
import app from './app.js'
import { PORT } from './config.js'

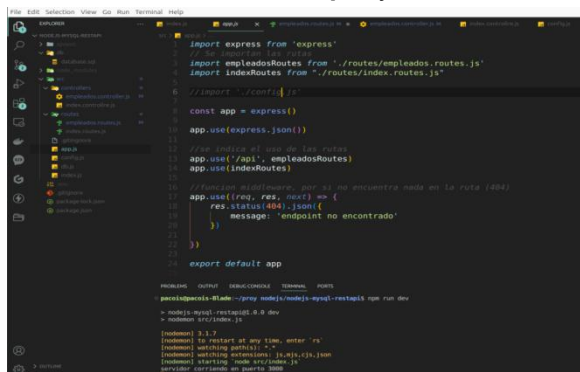
app.listen(3000)
console.log("servidor corriendo en puerto 3000")
```

Sigue funcionando



Con esto ya se habrá implementado una **API rest simple** y la estructura en VScode.

Se verá así...



El diseño de APIs debe abordar múltiples dimensiones, como la usabilidad, seguridad y el rendimiento y requiere un enfoque holístico que abarque desde la definición de objetivos y alcance hasta el monitoreo y mantenimiento.

Al seguir los principios y mejores prácticas mencionados anteriormente, se puede garantizar que una API sea robusta, escalable, segura y fácil de usar, lo que facilitará su adopción y contribuirá al éxito del proyecto.

Referencias

Astera Software. (2023). ¿Qué es el diseño API? Definición, proceso y mejores prácticas.

Bambu Mobile. (2023). 10 Mejores prácticas para el desarrollo de APIs.

Belvo. (2024). Cómo construir una API que enamore a los desarrolladores.

OWASP. (2021). OWASP Top 10:2021.

Red Hat. (2023). ¿En qué consiste el diseño de API?.

Richardson, L., & Ruby, S. (2013). RESTful web services. O'Reilly Media.

CAPÍTULO 3

Integración de aplicaciones con tecnologías actuales

Herramientas para SOA y APIs

El objetivo de este capítulo es mostrar de manera pragmática un entorno sencillo implementando el uso de SOAP además de una revisión de tecnologías APIs, como GraphQL y gRPC.



Existe una tendencia en las empresas por el desarrollo de aplicaciones que puedan trabajar sobre Internet, principalmente por la ventaja de la distribución global de la información.

Tecnologías en Integración de Servicios

Algunas tecnologías fueron muy utilizadas e incluso todavía se usan algunas para el desarrollo de aplicaciones, antes de SOA, por mencionar algunas está:

CORBA (Common Object Request Broker Architecture) de OMG (Object Management Group)

COM - Microsoft Component Object Model de Microsoft

EJB (Enterprise JavaBeans) de Sun Microsystems.

Cada una proporciona un marco de trabajo para la activación de objetos remotos, mediante la solicitud a un servidor de aplicaciones para la ejecución de servicios de aplicación, han probado ser efectivas para el establecimiento de sitios Web corporativos; sin embargo, presentan algunas desventajas como la falta de interoperabilidad.

La dependencia a la arquitectura de trabajo COM está muy ligado a Windows, mientras que CORBA tiene muchas implementaciones de diversos fabricantes, así como el lenguaje de programación (COM usa primordialmente C++ y Visual Basic, mientras que EJB usa Java).

Actualmente los departamentos de tecnología se han caracterizado moverse hacia la apertura de sus bases de datos y aplicaciones previamente almacenadas, para que se pueda acceder a los datos y la funcionalidad fuera de los límites de la organización o reutilizarse en nuevos sistemas. La manifestación inicial de esta tendencia vino con la Arquitectura Orientada a Servicios (SOA) y la más reciente ha sido la explosión de las API de las aplicaciones web.

Con la rápida evolución de las tecnologías de integración, nuevos enfoques están transformando la forma en que se diseñan y gestionan los servicios. **REST**, **GraphQL** y **Microservicios** son algunas de las herramientas emergentes más relevantes que están redefiniendo el panorama de la integración y las APIs.

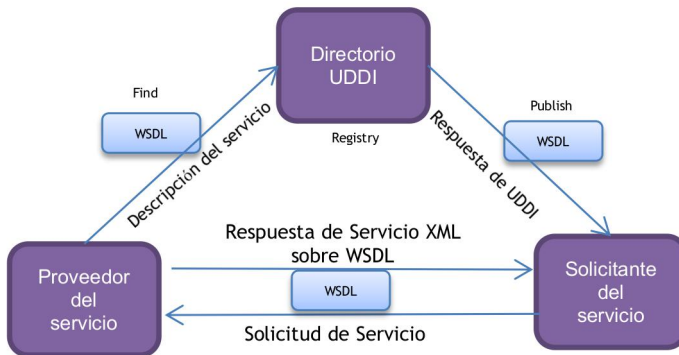
Cada una de estas tecnologías ofrece ventajas únicas para abordar los desafíos de la escalabilidad, la flexibilidad y el rendimiento en sistemas distribuidos.

Aunque en la actualidad **REST** y **GraphQL** han ganado popularidad debido a su simplicidad, **SOAP sigue siendo fundamental en sectores como la banca y los servicios gubernamentales** donde se requieren garantías de seguridad y transacciones confiables.

En el capítulo anterior se hizo referencia entorno a las APIs, y se tocó el tema de REST, en este capítulo se expondrá el protocolo **SOAP** además se hará una exploración de las tecnologías emergentes que están ganando popularidad como lo es **GraphQL** para consultar datos complejos.

SOAP (Simple Object Access Protocol)

SOAP es un protocolo de mensajería y se basa en estándares como **XML** (Extensible Markup Language) y **HTTP** (Hypertext Transfer Protocol) o **SMTP** (Simple Mail Transfer Protocol) para la transmisión de datos, permite la comunicación entre aplicaciones distribuidas a través de redes, como Internet y fue desarrollado por el W3C (World Wide Web Consortium).



<<Esquema - SOAP relación XML/WSDL/UDDI>>

Para ampliar conceptos podemos decir que el **XML** es un metalenguaje desarrollado por el W3C que permite la estructuración, almacenamiento y transporte de datos, su sintaxis se basa en etiquetas personalizables que organizan la información de manera jerárquica y estructurada.

Extensible Markup Language (XML)

XML desempeña un papel importante en el contexto de la **SOA**, debido a su capacidad para representar datos de manera interoperable entre sistemas heterogéneos, en SOA, los servicios necesitan comunicarse de manera flexible e independiente de plataformas y tecnologías. Xml facilita comunicación al proporcionar un formato estándar para representación de datos, y permite que diferentes aplicaciones intercambien información sin preocuparse por las diferencias en los sistemas operativos, los lenguajes de programación o las arquitecturas de hardware.

XML es un estándar de marcado que define reglas para la codificación de documentos en un formato que sea tanto legible por humanos y por sistemas informáticos(W3C, 2008).

En las APIs, se definen esquemas de datos mediante **XSD** (XML Schema Definition) usando XML, lo que permite una validación estricta de los documentos XML antes de ser procesados por un servicio o aplicación.

Además, XML se usa para describir servicios con **WSDL** (Web Services Description Language), que permite especificar las operaciones, formatos de mensajes y protocolos de comunicación en servicios web basados en SOA (Alonso et al., 2004).

XML es fundamental en la evolución de las arquitecturas orientadas a servicios por su capacidad de representar datos de manera estructurada y estandarizada. A pesar de formatos más ligeros como JSON, XML sigue siendo relevante en entornos empresariales y en servicios web que requieren altos niveles de interoperabilidad y validación de datos. Su adopción en estándares como SOAP y WSDL demuestra su impacto en la construcción de arquitecturas escalables y orientadas a servicios.

A diferencia de otros enfoques más ligeros, como **REST**, **SOAP** proporciona una comunicación **estrictamente estructurada y segura**, lo que lo hace ideal para entornos empresariales donde se requiere **transacciones confiables y estándares de seguridad elevados**.

SOAP encapsula mensajes en formato XML para su transmisión a través de protocolos como HTTP y SMTP, esto garantiza la interoperabilidad de los servicios al definir estructuras estándar para los mensajes y al proporcionar mecanismos para la validación, seguridad y transacciones distribuidas. Su arquitectura se compone de tres elementos principales:

- **Envoltorio SOAP (SOAP Envelope):** Contiene la estructura del mensaje, incluyendo encabezados y cuerpo.
- **Encabezado SOAP (SOAP Header):** Opcional, utilizado para metadatos como seguridad o control de transacciones.
- **Cuerpo SOAP (SOAP Body):** Contiene la información principal de la solicitud o respuesta.

SOAP opera sobre múltiples protocolos de transporte, siendo el más común **HTTP**. También puede utilizar **SMTP** para la mensajería asincrónica. Además, suele emplearse XML con **WSDL**, que define cómo interactuar con un servicio web SOAP, permitiendo su integración en distintos entornos tecnológicos.

Su uso es común en sistemas empresariales que requieren:

- **Alta seguridad** (mediante WS-Security).
- **Mensajería confiable** (garantía de entrega y orden de los mensajes).
- **Operaciones transaccionales complejas**.

SOAP se comunica a través de una petición-respuesta. Un cliente envía una solicitud SOAP a un servidor, el cual procesa el mensaje y devuelve una respuesta estructurada en XML.

Ejemplo un sistema bancario permite a los usuarios consultar su saldo a través de un servicio web SOAP.

El cliente envía una solicitud SOAP al servidor bancario con su número de cuenta.

Mensaje de Solicitud SOAP

```
xml
CopiarEditar
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:bank="http://www.banco.com/servicios">
  <soapenv:Header/>
  <soapenv:Body>
    <bank:ConsultaSaldo>
      <bank:NumeroCuenta>123456789</bank:NumeroCuenta>
    </bank:ConsultaSaldo>
  </soapenv:Body></soapenv:Envelope>
```

El servidor valida la solicitud, consulta el saldo en la base de datos y se genera un mensaje de respuesta SOAP con el saldo y se devuelve al cliente.

Mensaje de Respuesta SOAP

```
xml
CopiarEditar
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:bank="http://www.banco.com/servicios">
  <soapenv:Header/>
  <soapenv:Body>
    <bank:ConsultaSaldoResponse>
      <bank:Saldo>5000.00</bank:Saldo>
    </bank:ConsultaSaldoResponse>
  </soapenv:Body></soapenv:Envelope>
```

En el ejemplo, el cliente envía solicitud al servicio bancario y recibe una respuesta en **XML**, permite una comunicación estándar y comprensible para diferentes plataformas, y esta se da en la interface WSDL.

Interfaz WSDL, descripción de la forma en que un cliente puede acceder a un servicio web, un esquema XML que describe los protocolos, formatos y firmas de las operaciones de un servicio web.

Para consumir un Servicio Web

- Primero se toma la definición del archivo WSDL.
- El servidor le entrega el archivo WSDL. Este archivo contiene los métodos y propiedades (para entendernos) de ese servicio que están disponibles.
- El cliente hace la petición en el formato que espera el servidor según las especificaciones del archivo WSDL en el que se dice qué parámetros acepta y de qué tipo.
- El servidor le entrega el resultado de la consulta.

Web Services Description Language

Contenido de una Interface **WSDL**.

<?xml version="1.0">

Este es un documento XML, por eso que debe comenzar con el tag **<?xml .. ?>**

<definitions> Comienzo del documento, este tag agrupa a todos los demás.

<types> </types>

Se definen los tipos de datos utilizados en el Web Service.

<message> </message>

Se definen los métodos y parámetros para realizar la operación. Cada message puede consistir en una o más partes (parámetros).

<portType> </portType>

Esta sección es la más importante, ya que se definen las operaciones que pueden ser realizadas, y los mensajes que involucran (por ejemplo el mensaje de petición y el de respuesta)

<binding> </binding>

Es el que establece la transición desde tipos de datos abstractos a tipos de datos concretos, para cada portType.

</definitions> Fin del documento WSDL

```
<?xml version="1.0">
<definitions>
  <types>
    ...
  </types>
  <message>
    ...
  </message>
  <portType>
    ...
  </portType>
  <binding>
    ...
  </binding>
</definitions>
```

SOAP juega un papel importante cuando un sistema quiere acceder a otro de manera ordenada y limitada, en lugar de darle al cliente solicitante acceso total al servidor, con un protocolo como SOAP puede limitarse el acceso a las funciones que necesita, la arquitectura del protocolo, al facilitar un marco al que la aplicación puede incorporarse, ofrece así la ventaja de que sistemas muy diferentes pueden cooperar entre sí.

En la versión 1.0 de la especificación de SOAP se establecía como mandatorio el uso de HTTP como protocolo de transporte, sin embargo a partir de la versión 1.1 se desacopla SOAP del uso de HTTP, lo cual permite una mayor variedad de protocolos de transporte como FTP, SMTP, etc. En definitiva la especificación no establece un protocolo de transporte particular pero si especifica como se realiza el transporte en caso de usar HTTP.

Diseñando una api soap simple

Para entender el funcionamiento, se contemplará la implementación de una app y su Interface con el usuario. Cabe aclarar que en REST la arquitectura utiliza los métodos HTTP, y en el caso de SOAP es un protocolo que implementa funcionalidad, por lo que para explicar el diseño de la api soap, primero se implementará la aplicación web para posteriormente implementar el servicio web soap.

Dado lo anterior lo siguiente será la creación e implementación de la aplicación y de la *API SOAP simple* a manera de ejemplo, si el lector la implementa siguiendo la explicación seguramente entenderá mas claramente la interoperabilidad y reutilización en el diseño de APIs.

Se usará el patrón MVC, y las siguientes herramientas y tecnologías:

Lenguaje

Java

IDE

NetBeans

Servicios de aplicación

Glassfish

Motor de datos

MySQL.



Consideraciones importantes:

Java es un lenguaje de programación ampliamente utilizado para codificar aplicaciones web, ha sido una opción popular entre los desarrolladores durante más de dos décadas, con millones de aplicaciones Java en uso en la actualidad. Java es un lenguaje multiplataforma, orientado a objetos y centrado en la red que se puede utilizar como una plataforma en sí mismo. Es un lenguaje de programación rápido, seguro y confiable para codificarlo todo, desde aplicaciones móviles y software empresarial hasta aplicaciones de macrodatos y tecnologías del servidor.

NetBeans IDE es un entorno de desarrollo integrado, gratuito y de código abierto para el desarrollo de aplicaciones en los sistemas operativos Windows, Mac, Linux y Solaris.

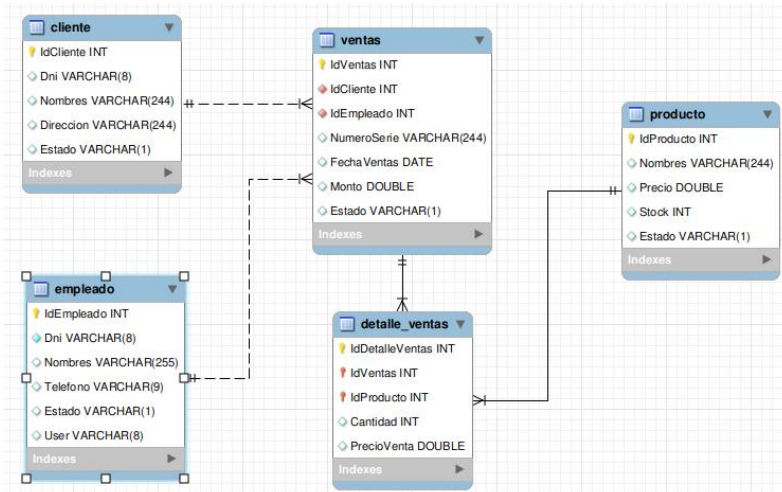
Este IDE simplifica el desarrollo de aplicaciones web, corporativas, de escritorio y móviles que utilizan plataformas Java y HTML5. El IDE también ofrece soporte para el desarrollo de aplicaciones PHP y C/C++ y contiene tecnologías innovadoras listas para usar y es el estándar en el desarrollo de aplicaciones, gracias a sus características integrales para el desarrollo de aplicaciones, las constantes mejoras en el editor de Java y el perfeccionamiento del rendimiento y la velocidad.

Glassfish es un servidor compatible con Java EE 5 para desarrollo e implementación de las aplicaciones y servicios web basados en la tecnología Java.

Primeramente instalar y configurar las herramientas (puede ser localmente), Java, NetBeans, Glassfish y MySql.

Publicación del servicio

Para ejemplificar se hará referencia a una aplicación de ventas que posteriormente publicará servicios, por lo que se crea un esquema de base de datos con 5 tablas (cliente, ventas, producto, empleado, detalle_ventas) en este caso será en MySql.



<<Esquema - Base de Datos>>

#	IdCliente	Dni	Nombres	Direccion	Estado
1	17	2	Juan Guerrero Solis	Los Alamos	1
2	18	1	Maria Rosas Villanueva	Los Laureles 234	1
3	19	3	Andres de Santa Cruz	Av. La Frontera 347	1
4	20	4	Andres Mendoza	Chosica, Lurigancho	1
5	27	5	fedee	rio santa	1

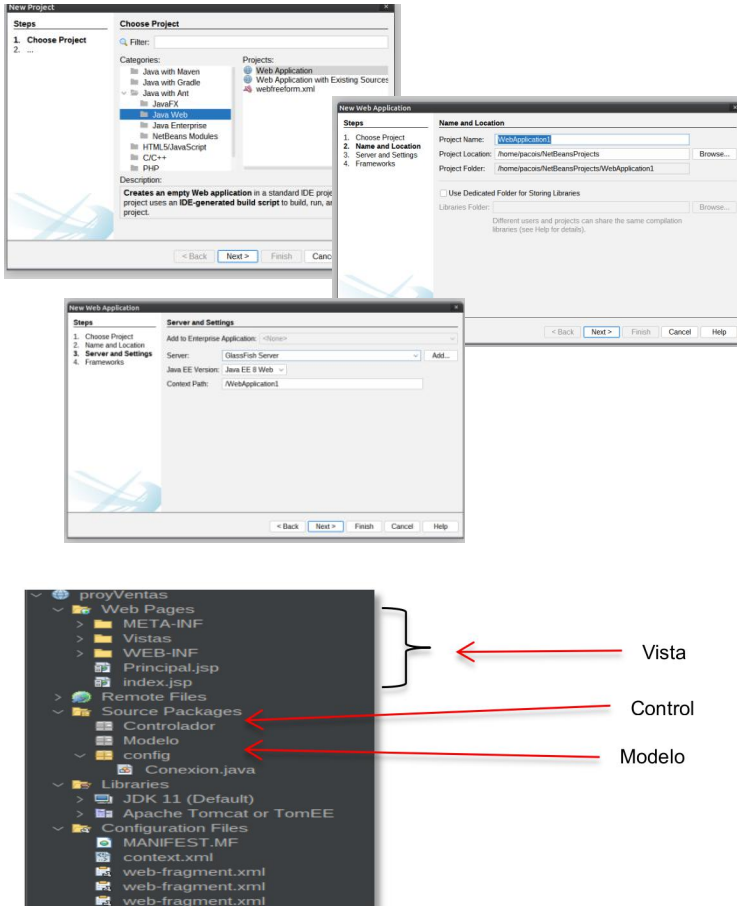
cliente 1

Columns:

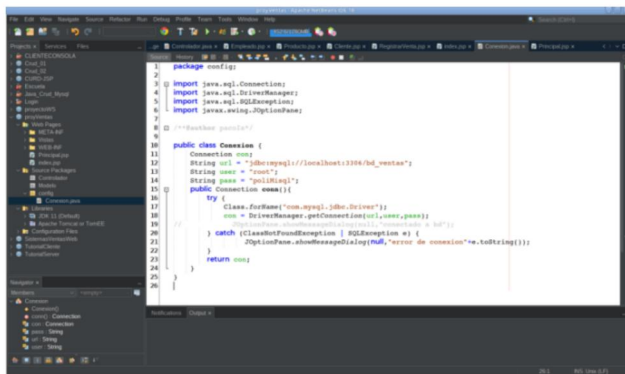
- IdCliente int(11) AI PK
- Dni varchar(8)
- Nombres varchar(244)
- Direccion varchar(244)
- Estado varchar(1)

<<Gráfica - MySql Base de Datos>>

Ahora se crea el proyecto en NetBeans, seleccionando el tipo de proyecto, nombre, localización y servidor.



Se programa la conexión.



<<Estructura Inicial en NetBeans>>

Ahora la entidad Cliente (JavaBean) y su DAO

JavaBean hace referencia a la definición de clases de negocio con propiedades privadas con métodos getters y setters además debe tener un constructor público sin parámetros.

DAO (Data Access Object) patrón de diseño para operar de persistencia en la base de datos, de la entidad cliente.

javabeen Cliente.java

```
package Modelo;

/** @author pacois */
public class Cliente {

    int id;
    String dni;
    String nom;
    String dir;
    String estado;

    public Cliente() {
    }

    public Cliente(int id, String dni, String nom, String dir, String estado) {
        this.id = id;
        this.dni = dni;
        this.nom = nom;
        this.dir = dir;
        this.estado = estado;
    }

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getDni() { return dni; }
    public void setDni(String dni) { this.dni = dni; }

    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }

    public String getDir() { return dir; }
    public void setDir(String dir) { this.dir = dir; }

    public String getEstado() { return estado; }
    public void setEstado(String estado) { this.estado = estado; }
}
```

ClienteDAO.java

```
package Modelo;

import ...7 lines

/** @author pacois */
public class ClienteDAO {

    Conexion claseConexion = new Conexion();
    Connection con;
    PreparedStatement pstmt;
    ResultSet rs;
    Cliente cl = new Cliente();
    int r;
    String msj;

    public Cliente Buscar(String dni) {...18 lines}

    // ===== Operaciones CRUD =====//
    // metodo para usar en webservice
    public List listar() {...21 lines}

    public int Agregar(Cliente cl) {...15 lines}

    // metodo para usar en webservice
    public String add(String Dni, String Nom, String Dir, String Estado) {...20 lines}

    // metodo para usar en webservice
    public Cliente listarId(int id) {...17 lines}

    public int Actualizar(Cliente cli) {...16 lines}

    // metodo (Actualizar) para usar en webservice
    public String edit(int Id, String Dni, String Nom, String Dir, String Estado) {...20 lines}

    // metodo para usar en webservice
    public Cliente Delete(int id) {...11 lines}
}
```

Métodos CRUD de DAO

Método Buscar

```

public Cliente Buscar(String dni) {
    String sql = "select * from cliente where Dni=" + dni;
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        rs = pstmt.executeQuery();
        while (rs.next()) {
            cl.setId( id:rs.getInt( columnIndex:1));
            cl.setDni( dni:rs.getString( columnIndex:2));
            cl.setNom( nom:rs.getString( columnIndex:3));
            cl.setDir( dir:rs.getString( columnIndex:4));
            cl.setEstado( estado:rs.getString( columnIndex:5));
        }
    } catch (SQLException e) {
        System.out.println("no se listo" + e.toString());
    }
    return cl;
}

```

Método Listar para usar en webService

```

public List Listar() {
    String sql = "select * from cliente";
    List<Cliente> lista = new ArrayList<>();
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        rs = pstmt.executeQuery();
        while (rs.next()) {
            Cliente c = new Cliente();
            c.setId( id:rs.getInt( columnIndex:1));
            c.setDni( dni:rs.getString( columnIndex:2));
            c.setNom( nom:rs.getString( columnIndex:3));
            c.setDir( dir:rs.getString( columnIndex:4));
            c.setEstado( estado:rs.getString( columnIndex:5));
            lista.add( e:c);
        }
    } catch (SQLException e) {
        System.out.println("no se listo" + e.toString());
    }
    return lista;
}

```

Método Agregar

```

public int Agregar(Cliente cl) {
    String sql = "insert into cliente (Dni, Nombres, Direccion, Estado) values (?, ?, ?, ?)";
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        pstmt.setString( parameterIndex:1, x:cl.getDni());
        pstmt.setString( parameterIndex:2, x:cl.getNom());
        pstmt.setString( parameterIndex:3, x:cl.getDir());
        pstmt.setString( parameterIndex:4, x:cl.getEstado());
        pstmt.executeUpdate();
    } catch (SQLException e) {
        System.out.println("no se agrego cliente" + e.toString());
    }
    return r;
}

```

Método add para usar en webService

```

public String add(String Dni, String Nom, String Dir, String Estado) {
    String sql = "insert into cliente (Dni, Nombres, Direccion, Estado) values (?, ?, ?, ?)";
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        pstmt.setString(1, x:Dni);
        pstmt.setString(2, x:Nom);
        pstmt.setString(3, x:Dir);
        pstmt.setString(4, x:Estado);
        r = pstmt.executeUpdate();
        if (r == 1) {
            msj = "cliente agregado";
        } else {
            msj = "error";
        }
    } catch (SQLException e) {
        System.out.println("no se agrego cliente" + e.toString());
    }
    return msj;
}

```

Listar por ID Método Listar por ID para usar en webService

```

public Cliente listarId(int id) {
    String sql = "select * from cliente where IdCliente=" + id;
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        rs = pstmt.executeQuery();
        while (rs.next()) {
            cl.setDni( dni:rs.getString( columnIndex:2));
            cl.setNom( nom:rs.getString( columnIndex:3));
            cl.setDir( dir:rs.getString( columnIndex:4));
            cl.setEstado( estado:rs.getString( columnIndex:5));
        }
    } catch (SQLException e) {
        System.out.println("no se listo" + e.toString());
    }
    return cl;
}

```

Método Actualizar

```

public int Actualizar(Cliente cli) {
    String sql = "update cliente set Dni=?, Nombres=?, Direccion=?, Estado=? where IdCliente=?";
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        pstmt.setString(1, x:cli.getDni());
        pstmt.setString(2, x:cli.getNom());
        pstmt.setString(3, x:cli.getDir());
        pstmt.setString(4, x:cli.getEstado());
        pstmt.setInt(5, x:cli.getId());
        pstmt.executeUpdate();
    } catch (SQLException e) {
        System.out.println("no se actualizo cliente" + e.toString());
    }
    return r;
}

```

Método edit para usar en webService

```

public String edit(int Id, String Dni, String Nom, String Dir, String Estado) {
    String sql = "update cliente set Dni=?, Nombres=?, Direccion=?, Estado=? where IdCliente=" + Id;
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        pstmt.setString( parameterIndex:1, x:Dni);
        pstmt.setString( parameterIndex:2, x:Nom);
        pstmt.setString( parameterIndex:3, x:Dir);
        pstmt.setString( parameterIndex:4, x:Estado);
        r = pstmt.executeUpdate();
        if (r == 1) {
            msj = "cliente actualizado";
        } else {
            msj = "error";
        }
    } catch (SQLException e) {
        System.out.println("no se agrego cliente" + e.toString());
    }
    return msj;
}

```

Método Delete para usar en webService

```

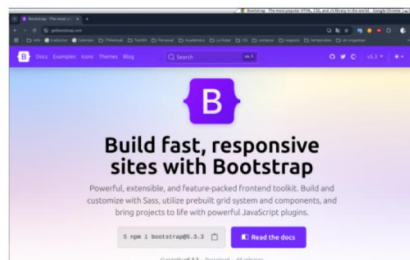
public Cliente Delete(int id) {
    String sql = "delete from cliente where IdCliente=" + id;
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        pstmt.executeUpdate();
    } catch (SQLException e) {
        System.out.println("no se borro cliente" + e.toString());
    }
    return cl;
}

```

Todas la entidades tendrán similitud, por lo que **solo se incluirá la entidad Cliente y su DAO.**

Siguiendo se construye las interfaces de inicio de sesión y vista principal iniciando con el archivo **index.jsp**, utilizando el framework Bootstrap

JSP (JavaServer Pages) es una tecnología que permite crear páginas web dinámicas. Se basa en el lenguaje de programación Java y se utiliza para generar contenido web dinámico, como archivos HTML, XML, XHTML y DHTML.



<<Gráfica - Framework Bootstrap>>

Se agregan los enlaces bootstrap de referencia CSS y los fuentes de JS

```

1 <!-- Author : pacols -->
2
3 <!page contentType="text/html" pageEncoding="UTF-8"%>
4 <DOCTYPE html>
5 <html>
6 <head>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-ndcywV6b06Z/p5kFyTf02zgjAjXO8N6l1y4G7J96h6mn2l7486yb8ffD4f2a" crossorigin="anonymous">
9 <title>JSP Page</title>
10 </head>
11 <body>
12
13 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-gW76RC66w4Pwv9tj0673Cm3Q634tWTmJGp382C8373270s36ylqoE4preHqD" crossorigin="anonymous"></script>
14 <script src="https://cdn.jsdelivr.net/npm/popper.js@2.11.0/dist/umd/popper.min.js" integrity="sha384-7876473074L48898299832E7684e9283176J5bbf443Jg9924563804150" crossorigin="anonymous"></script>
15 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.min.js" integrity="sha384-7876473074L48898299832E7684e9283176J5bbf443Jg9924563804150" crossorigin="anonymous"></script>
16
17 </body>
18 </html>

```

Y se agrega el código de la Interface para inicio de sesión en **index.jsp**.

```

1 <!-- Author : pacols -->
2
3 <!page contentType="text/html" pageEncoding="UTF-8"%>
4 <DOCTYPE html>
5 <html>
6 <head>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous">
9 <title>JSP Page</title>
10 </head>
11 <body>
12 <div class="container mt-4 col-lg-4">
13 <div class="card col-sm-10">
14 <div class="form-sign" action="Validar" method="POST">
15 <div class="form-group text-center">
16 
17 <label style="margin-left: 100px;">Todo en Electrónica</label><br>
18 <h3>Sistema Ventas (Practica)</h3>
19 </div>
20 <div class="form-group">
21 <label>Usuario:</label>
22 <input type="text" name="txtuser" class="form-control"></input>
23 </div>
24 <div class="form-group">
25 <label>Password:</label>
26 <input type="password" name="txtpass" class="form-control">
27 </div>
28 <input type="submit" name="accion" value="Ingresar" class="btn btn-primary">
29 </form>
30 </div>
31 </div>
32 <script src="css/js/bootstrap.bundle.min.js" crossorigin="anonymous"></script>
33 </body>
34 </html>

```

Se crea una clase para validar como parte del control.

```

1 package Controlador;
2
3 /**
4  * @author pacols
5  */
6 import ...9 lines
7
8 public class Validar extends HttpServlet {
9
10     EmpleadoDAO edao = new EmpleadoDAO();
11     Empleado em = new Empleado();
12
13     protected void processRequest(HttpServletRequest request, HttpServletResponse response)
14         throws ServletException, IOException {
15
16     }
17
18     @Override
19     protected void doPost(HttpServletRequest request, HttpServletResponse response)
20         throws ServletException, IOException {
21
22     }
23
24     String accion = request.getParameter("accion"); // parametro del submit de index.jsp
25     if (accion.equals("Ingresar")) {
26         String user = request.getParameter("txtuser");
27         String pass = request.getParameter("txtpass");
28         try {
29             em = edao.validar(user, pass);
30         } catch (SQLException e) {
31             JOptionPane.showMessageDialog(null, e.toString());
32         }
33         if (em.getUser() != null) {
34             request.getRequestDispatcher("index.jsp").forward(request, response);
35         } else {
36             request.setAttribute("usuario", em);
37             request.getRequestDispatcher("ControladorMenuPrincipal").forward(request, response);
38         }
39     } else {
40         request.getRequestDispatcher("index.jsp").forward(request, response); // se envia la peticion a index.jsp
41     }
42 }
43
44 }
45

```

Clase EmpleadoDAO.java

En el modelo de empleado se agrega el método para validar, dado que los empleados serian los que tienen acceso a la app.

```

1 package Modelo;
2
3 import ...0 lines
4
5 /**
6  * @author pascual
7  */
8 public class EmpleadoDAO {
9
10     Conexion claseConexion = new Conexion();
11     Conexion con;
12     PreparedStatement pstmt;
13     ResultSet rs;
14     Empleado emp = new Empleado();
15     int i;
16
17     public Empleado validar(String user, String dni) throws SQLException {
18         try {
19             con = claseConexion.conexion();
20             String sql = "select * from empleado where User=? and Dni=?";
21             pstmt = con.prepareStatement(sql);
22             pstmt.setString(1, user);
23             pstmt.setString(2, dni);
24             rs = pstmt.executeQuery();
25             while (rs.next()) {
26                 emp.setId(rs.getInt("idEmpleado"));
27                 emp.setUser(rs.getString("User"));
28                 emp.setDni(rs.getString("Dni"));
29                 emp.setNom(rs.getString("Nombre"));
30             }
31         } catch (SQLException e) {
32             JOptionPane.showMessageDialog(null, e.toString());
33         }
34         return emp;
35     }
36 }

```

Se usa Bootstrap para **generar la barra de navegación** en la interface principal, y se agrega el código en archivo **Principal.jsp**

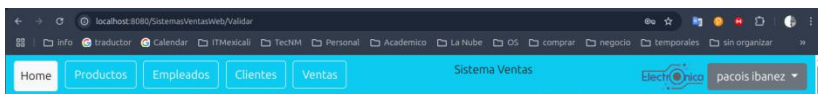
```

<!-- Author      : pacois -->

<doctype html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link href="css/js/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous">
<title>Sistema Ventas</title>
</head>
<body>
<nav class="navbar navbar-expand-lg navbar-light bg-info">
<div class="container-fluid">
<div class="collapse navbar-collapse id="navbarSupportedContent">
<ul class="navbar-nav me-auto mb-2 mb-lg-0">
<li class="nav-item">
<a class="nav-link active btn btn-outline-light" aria-current="page" href="/principal.jsp" id="home" />
</li>
<li class="nav-item">
<a style="margin-left: 10px;" class="btn btn-outline-light" href="/Controlador?menu=Productos&accion=Listar" target="myFrame">Productos</a>
</li>
<li class="nav-item">
<a style="margin-left: 10px;" class="btn btn-outline-light" href="/Controlador?menu=Empleados&accion=Listar" target="myFrame">Empleados</a>
</li>
<li class="nav-item">
<a style="margin-left: 10px;" class="btn btn-outline-light" href="/Controlador?menu=Clientes&accion=Listar" target="myFrame">Clientes</a>
</li>
<li class="nav-item">
<a style="margin-left: 10px;" class="btn btn-outline-light" href="/Controlador?menu=NuevaVenta&accion=default" target="myFrame">Ventas</a>
</li>
<li>
<a style="margin-left: 150px;" aria-current="page">Sistema Ventas</a>
</li>

</li>
</ul>
</div>
<div class="dropdown">
<button class="btn btn-secondary dropdown-toggle" type="button" data-bs-toggle="dropdown" aria-expanded="false">
<${usuario.getNom()}>
</button>
<ul class="dropdown-menu">
<li><a class="dropdown-item" href="#"></a></li>
<li><a class="dropdown-item" href="#">${usuario.getUser()}</a></li>
<li><a href="#" class="dropdown-divider"></a></li>
<li><a href="#">
<form action="/Validar" method="POST">
<button class="dropdown-item" name="accion" value="Salir">Salir</button>
</form></li>
</ul>
</div>
</div>
</nav>
<div class="ratio ratio-21x9">
<iframe name="myFrame" src="img/bufatec.jpg" style="height: 100%; width: 100%; border: none;"></iframe>
</div>
<!-- BOOTSTRAP 5 JS -->
<script src="css/js/js/bootstrap.bundle.min.js" crossorigin="anonymous"></script>
</body>
</html>

```



Se agregan las interfaces de entidades (solo Clientes para ejemplificar), la interface tendrá dos áreas una en el lado izquierdo y otra en el lado derecho.

Se agrega el código para la parte izquierda en archivo **Cliente.jsp**

```

<!-- Author : pacols -->
<? taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<@page contentType="text/html" pageEncoding="UTF-8"%>
<DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link href="/css/js/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous">
<title>CLIENTE</title>
</head>
<body>
<div class="d-flex">
<div class="col-sm-4 parte01 p-2">
<div class="card">
<form action="Controlador?menu=Cliente" method="POST">
<div class="card-body">
<div class="form-group d-grid">
<div class="col-sm-7 d-flex gap-2 p-1">
<label class="bg-light border form-control">=====CLIENTES=====</label>
<div class="form-group">
<label>Dni</label>
<input type="text" value="{cliente.getdni()}" name="txtDni" class="form-control">
</div>
<div class="form-group">
<label>Nombres</label>
<input type="text" value="{cliente.getNom()}" name="txtNombres" class="form-control">
</div>
<div class="form-group">
<label>Dirección</label>
<input type="text" value="{cliente.getDir()}" name="txtDir" class="form-control">
</div>
<div class="form-group">
<label>Estado</label>
<input type="text" value="{cliente.getEstado()}" name="txtEstado" class="form-control">
</div>
</div>
</div>
<div class="form-group p-1 bg-light border"></div>
</div>
<div>
<input type="submit" name="accion" value="Agregar" class="btn btn-info">
<input type="submit" name="accion" value="Actualizar" class="btn btn-success">
</div>
</div>
</div>

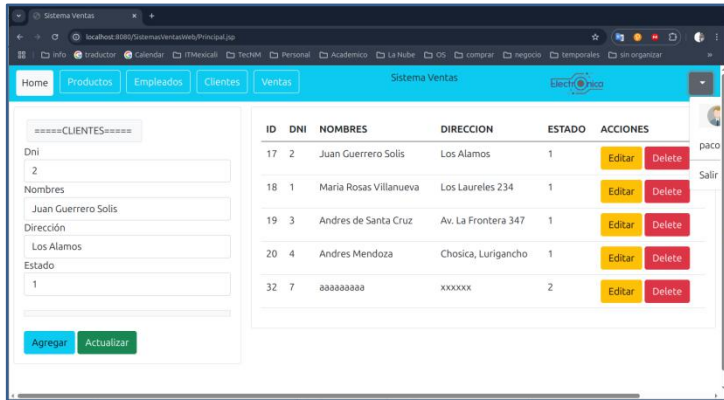
```

Se agrega el código para la parte derecho en archivo **Cliente.jsp**

```
<div class="col-sm-8 p-2">  
  <div class="card card-body">  
    <table class="table table-hover">  
      <thead>  
        <tr>  
          <th>ID</th>  
          <th>DNI</th>  
          <th>NOMBRES</th>  
          <th>DIRECCION</th>  
          <th>ESTADO</th>  
          <th>ACCIONES</th>  
        </tr>  
      </thead>  
      <tbody>  
        <c:forEach var="cl" items="{clientes}">  
          <tr>  
            <td>${cl.getId()}</td>  
            <td>${cl.getDni()}</td>  
            <td>${cl.getNom()}</td>  
            <td>${cl.getDir()}</td>  
            <td>${cl.getEstado()}</td>  
            <td>  
              <a class="btn btn-warning" href="/Controlador?menu=Cliente&accion=Editar&id=${cl.getId()}">Editar</a>  
              <a class="btn btn-danger" href="/Controlador?menu=Cliente&accion=Delete&id=${cl.getId()}">Delete</a>  
            </td>  
          </c:forEach>  
        </tbody>  
      </table>  
    </div>  
  </div>  
  
<!-- BOOTSTRAP 5 JS -->  
<script src="css/js/bootstrap.bundle.min.js" crossorigin="anonymous"></script>  
</html>
```

ID	DNI	NOMBRES	DIRECCION	ESTADO	ACCIONES
17	2	Juan Guerrero Solis	Los Alamos	1	Editar Delete
18	1	Maria Rosas Villanueva	Los Laureles 234	1	Editar Delete
19	3	Andres de Santa Cruz	Av. La Frontera 347	1	Editar Delete
20	4	Andres Mendoza	Chosica, Lurigancho	1	Editar Delete
32	7	aaaaaaaaaa	xxxxxxx	2	Editar Delete

La interface **Cliente.jsp** en la interface **Principal.jsp** se verá ...



A continuación de trabaja con el control para Cliente, creando la clase **Controlador.java**

```
package Controlador;

/**
 * @author paco1s
 */
import ...16 lines

public class Controlador extends HttpServlet {

    Empleado em = new Empleado();
    EmpleadoDAO edao = new EmpleadoDAO();
    Cliente cl = new Cliente();
    ClienteDAO cdao = new ClienteDAO();
    Producto pr = new Producto();
    ProductoDAO pdao = new ProductoDAO();
    int idc;
    int idd;
    int idp;

    Venta v = new Venta();
    List<Venta> lista = new ArrayList<>();
    int item;
    int cod;
    String descripcion;
    Double precio;
    int cant;
    Double subtotal;
    Double totalPagar;

    String numeroSerie;
    VentaDAO vdao = new VentaDAO();

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String menu = request.getParameter("menu");
        String opcion = request.getParameter("accion");
        if (menu.equals("Principal")) {
            request.getRequestDispatcher("Principal.jsp").forward(request, response);
        }
    }
}
```

Cliente - Casos(Listar, Agregar y Editar)

```
if (menu.equals("Cliente")) {
    switch (opcion) {
        case "Listar":
            List lista = cdao.Listar();
            request.setAttribute("clientes", lista);
            break;
        case "Agregar":
            String dni = request.getParameter("txtDni");
            String nom = request.getParameter("txtNombres");
            String dir = request.getParameter("txtDir");
            String est = request.getParameter("txtEstado");
            cl.setDni(dni);
            cl.setNom(nom);
            cl.setDir(dir);
            cl.setEstado(est);
            cdao.Agregar(cl);
            request.getRequestDispatcher("Controlador?menu=Cliente&accion=Listar").forward(request, response);
            break;
        case "Editar":
            idc = Integer.parseInt(request.getParameter("id"));
            Cliente cli = cdao.listarId(idc);
            request.setAttribute("cliente", cli);
            request.getRequestDispatcher("Controlador?menu=Cliente&accion=Listar").forward(request, response);
            break;
    }
}
```

Cliente - Casos(Actualizar y Delete)

```

case "Actualizar":
    String dni1 = request.getParameter( name:"txtDni");
    String nom1 = request.getParameter( name:"txtNombres");
    String tel1 = request.getParameter( name:"txtTel");
    String est1 = request.getParameter( name:"txtEstado");
    String user1 = request.getParameter( name:"txtUser");
    em.setDni( dni1.dni1);
    em.setNom( nom1.nom1);
    em.setTel( tel1.tel1);
    em.setEstado( est1.est1);
    em.setUser( user1.user1);
    em.setId( id1.id1);
    edao.Actualizar(em);
    request.getRequestDispatcher( path:"Controlador?menu=Empleado&accion=Listar").forward(request, response);
    break;
case "Delete":
    id1 = Integer.parseInt( s:request.getParameter( name:"id"));
    edao.Delete( id1.id1);
    request.getRequestDispatcher( path:"Controlador?menu=Empleado&accion=Listar").forward(request, response);
    break;
default:
    throw new AssertionError();
}
request.getRequestDispatcher( path:"Empleado.jsp").forward(request, response);
}

```

Hasta aquí se generan las entidades, recordando que solo se presenta la entidad **Cliente**, quedando como práctica para aquellos lectores que quieran implementar las otras entidades (**Producto y Empleado**) será similar.

Esta implementación tendría que atender las ventas (tanto la venta como atender el registro de las ventas), pero dado que lo que nos ocupa en este libro es el “diseño de una api SOAP simple”, ya se puede explicar dicho API.

Creación del servicio (webService)

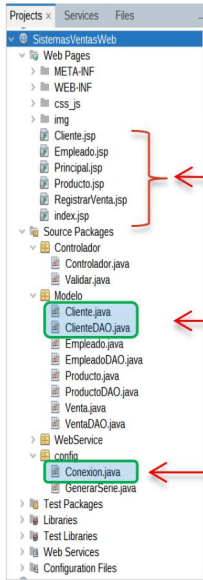
Es conveniente mencionar que se explicará creando interfaces diferentes a la app, para que se entienda mejor y se utilizara la misma base de datos, con la finalidad de dar mas atención en el objetivo de entender el diseño de la API.

Se utilizará la entidad Cliente

The screenshot shows the MySQL Workbench interface. In the 'SCHEMAS' panel on the left, the 'cliente' table is selected under the 'bd_ventas' database. The main editor displays the SQL query: `SELECT * FROM bd_ventas.cliente;`. Below the query, the 'Result Grid' shows the data returned by the query. The results are as follows:

#	IdCliente	Dni	Nombres	Direccion	Estado
1	17	2	Juan Guerrero Solis	Los Alamos	1
2	18	1	Maria Rosas Villanueva	Los Laureles 234	1
3	19	3	Andres de Santa Cruz	Av. La Frontera 347	1
4	20	4	Andres Mendoza	Chosica, Lurigancho	1
5	27	5	xxx	xxxx	2

Del proyecto ventas se usará la entidad Cliente y su DAO.



El proyecto consumidor no usará las vistas del proyecto, dado que se crearán sus propias vistas

El proyecto consumidor usará la entidad **Cliente** y su **DAO**

El consumidor usará la misma conexión, dado que se usará la misma base de datos

Clase Conexión

(Javabeen) del modelo en proyecto que genera servicio

```

Source History
package Modelo;

/** @author pascals */
public class Cliente {

    int id;
    String dni;
    String nom;
    String dir;
    String estado;

    public Cliente() {
    }

    public Cliente(int id, String dni, String nom, String dir, String estado) {
        this.id = id;
        this.dni = dni;
        this.nom = nom;
        this.dir = dir;
        this.estado = estado;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getDir() {
        return dir;
    }

    public void setDir(String dir) {
        this.dir = dir;
    }

    public String getEstado() {
        return estado;
    }

    public void setEstado(String estado) {
        this.estado = estado;
    }
}

```

```

Source History
package config;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

/** @author pascals */

public class Conexion {

    Connection con;
    String url = "jdbc:mysql://localhost:3306/bd_ventas?useSSL=false";
    String user = "root";
    String pass = "root";

    public Connection con() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            con = DriverManager.getConnection(url, user, password);
            Optional.ofNullable(con).orElseThrow(() -> new RuntimeException("No se pudo conectar a la BD"));
        } catch (ClassNotFoundException | SQLException e) {
            System.out.println("error conecta" + e.toString());
        }
        return con;
    }
}

```

Métodos que se usarán de la Clase ClienteDAO de la aplicación ventas

```

1 package Modelo;
2
3 import config.Conexion;
4 import java.sql.Connection;
5 import java.sql.PreparedStatement;
6 import java.sql.ResultSet;
7 import java.sql.SQLException;
8 import java.util.ArrayList;
9 import java.util.List;
10
11 /**
12  * @author Oscar W.
13  */
14 public class ClienteDAO {
15     Conexion con = new Conexion();
16     Connection con;
17     PreparedStatement pstmt;
18     ResultSet rs;
19     Cliente cli = new Cliente();
20     int r;
21     String msj;
22
23     public Cliente Buscar(String dni) [...18 lines]
24
25     // ===== Operaciones CRUD =====
26     // metodo para usar en webService
27     public List Listar() [...21 lines]
28
29     public int Agregar(Cliente cli) [...16 lines]
30
31     // metodo para usar en webService
32     public String add(String Dni, String Nom, String Dir, String Estado) [...20 lines]
33
34     // metodo para usar en webService
35     public Cliente ListarId(int id) [...17 lines]
36
37     public int Actualizar(Cliente cli) [...16 lines]
38
39     // metodo (Actualizar) para usar en webService
40     public String edit(int Id, String Dni, String Nom, String Dir, String Estado) [...2]
41
42     // metodo para usar en webService
43     public Cliente Delete(int id) [...11 lines]
44 }

```

- Listar
- add
- listarId
- edit
- delete

Ahora se utilizan los métodos de la implementación del CRUD en la app ventas, los cuales ya se mostraron anteriormente.

Se usa el método (listar) para servicio web

```

public List Listar() {
    String sql = "select * from cliente";
    List<Cliente> lista = new ArrayList<>();
    try {
        con = claseConexion.con();
        pstmt = con.prepareStatement(sql);
        rs = pstmt.executeQuery();
        while (rs.next()) {
            Cliente c = new Cliente();
            c.setId(rs.getInt( columnIndex:1));
            c.setDni( dni: rs.getString( columnIndex:2));
            c.setNom( nom: rs.getString( columnIndex:3));
            c.setDir( dir: rs.getString( columnIndex:4));
            c.setEstado( estado: rs.getString( columnIndex:5));
            lista.add( c );
        }
    } catch (SQLException e) {
        System.out.println("no se listo" + e.toString());
    }
    return lista;
}

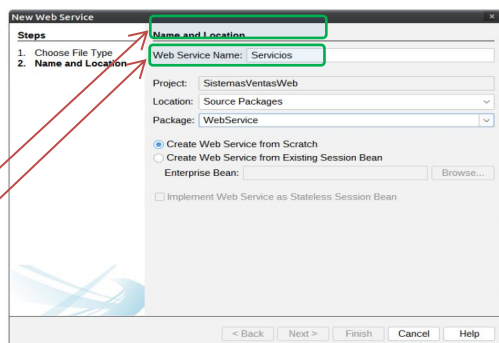
```



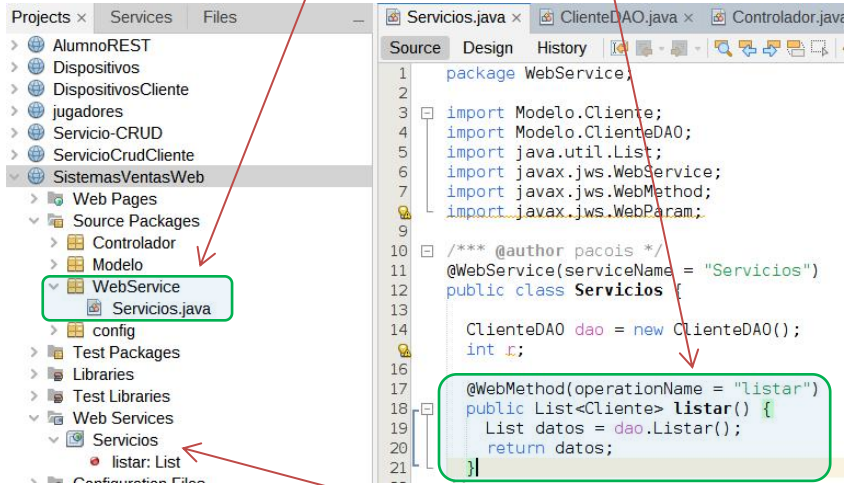
Se crea el servicio web listar, con el apuntador en el proyecto de la aplicación ventas, se da click derecho en mouse, seleccionar

1. New
2. WebServices

Se indica el nombre del servicio, y el paquete en el que estará.



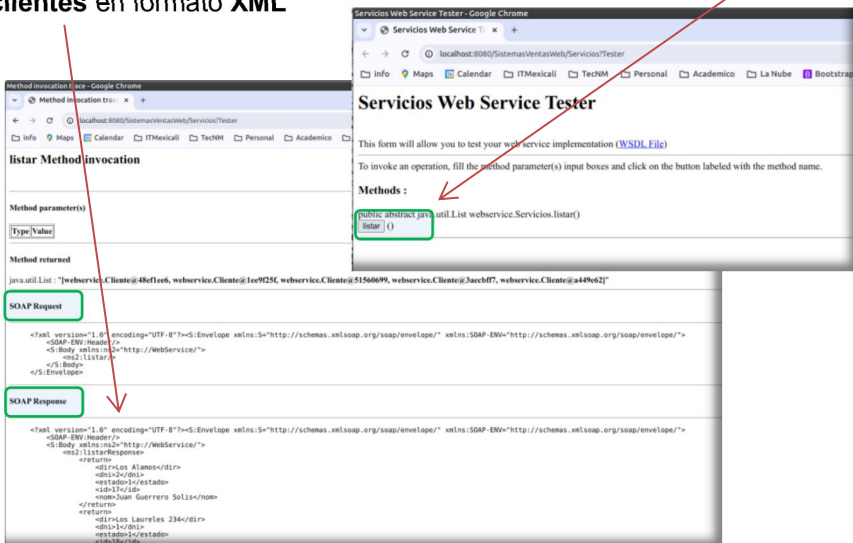
Lo anterior crea el archivo **Servicios.java** dentro del paquete **WebService** y se puede visualizar el método web **listar**



Ahora ya se puede probar el servicio web, desplegando el proyecto y posteriormente:

1. click derecho en mouse apuntando a **Servicios** en la referencia que se crea.
2. click en "Test Web Services"

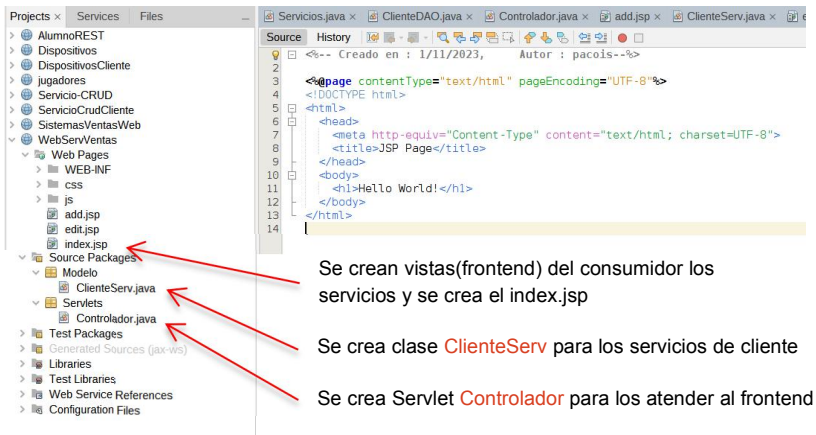
Se visualiza el endpoint para la prueba del servicio con el método **listar** y al dar click en el método, SOAP Request devolverá los datos de la tabla **clientes** en formato **XML**



En este punto ya se puede consumir el servicio.....

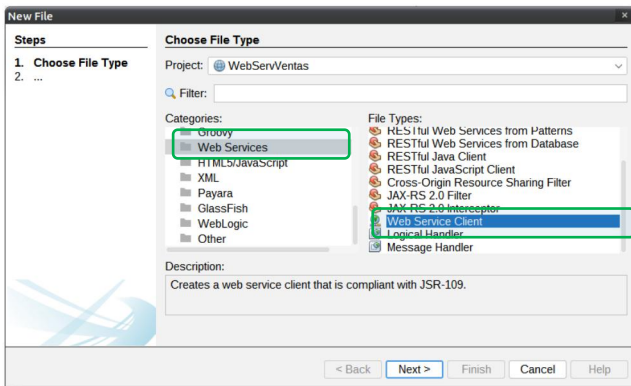
Consumiendo Servicios

Se implementa el método **listar** cliente en aplicación WebServVentas



Para obtener el servicio (WSDL),con el apuntador en el proyecto de la API, se da click derecho en mouse, seleccionar

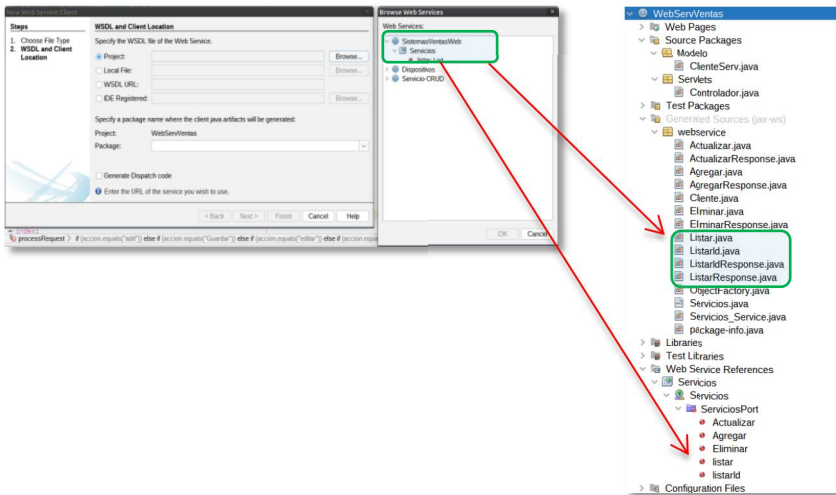
- New
- Web Services Client



Al continuar en la siguiente ventana se tratará de ubicar el WSDL en el UDDI, se proporciona una ventana que mostrará los servicios a los que se puede acceder.

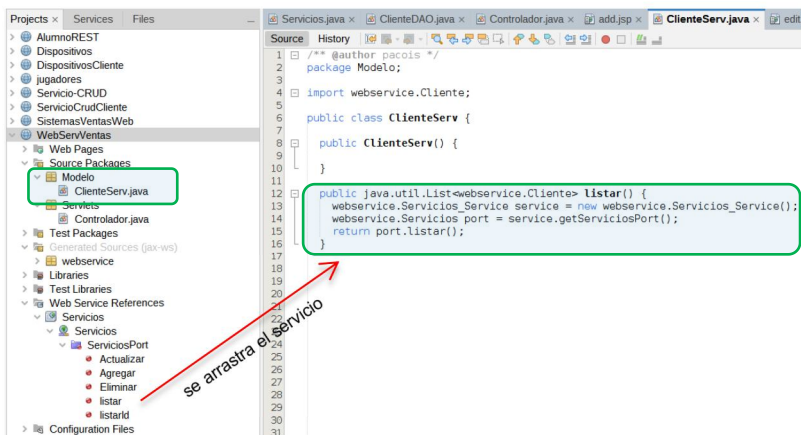


En Browse seleccionamos **Servicios**, para referenciar los servicios (por ahora listar).



Consumir el servicio.

Para usar el servicio (listar), dentro de la clase ClienteServ del servicio web, arrastramos el servicio usando el mouse, desde la referencia que se creó y posteriormente se cambia a público



Una vez que se tiene acceso al servicio web, se genera la vista en el archivo index.jsp, para listar los clientes obteniendo los datos desde la tabla Clientes.

Vista index.jsp (front)

esta vista es para listar los clientes, por lo que importamos el **webservice** y la clase **modelo**, para obtener la tabla de Clientes y mostrarla

codigo JSP (expression, declarations, etc...) para obtener datos de la sesión

```

<%page import="webservice.Cliente"%>
<%page import="Modelo.ClienteServ"%>
<%page import="java.util.List"%>
<%page contentType="text/html" pageEncoding="UTF-8"%>
<html>
<... 5 lines ...>
<body>
  <div class="container mt-4">
    <div class="card">
      <div class="card-header">
        <a href="#Controlador?accion=add" class="btn btn-primary">Nuevo Cliente</a>
      <div class="card-body">
        <table class="table table-hover">
          <thead>
            <tr>
              <th>ID</th>
              <th>DNI</th>
              <th>NOMBRES</th>
              <th>DIR</th>
              <th>ESTADO</th>
            </tr>
          </thead>
          <tbody>
            <%
              ClienteServ cli = new ClienteServ();
              List<Cliente> datos = cli.listar();
              for (Cliente c : datos) {
            <tr>
              <td><%= c.getId() %></td>
              <td><%= c.getDni() %></td>
              <td><%= c.getNom() %></td>
              <td><%= c.getDir() %></td>
              <td><%= c.getEstado() %></td>
              <td>
                <a href="#Controlador?accion=editar&id=<%= c.getId() %>" class="btn btn-warning">Edit</a>
                <a href="#Controlador?accion=eliminar&id=<%= c.getId() %>" class="btn btn-danger">Delete</a>
              </td>
            </tr>
            <%>
          </tbody>
        </table>
      </div>
    </div>
  </div>
</body>
</html>

```

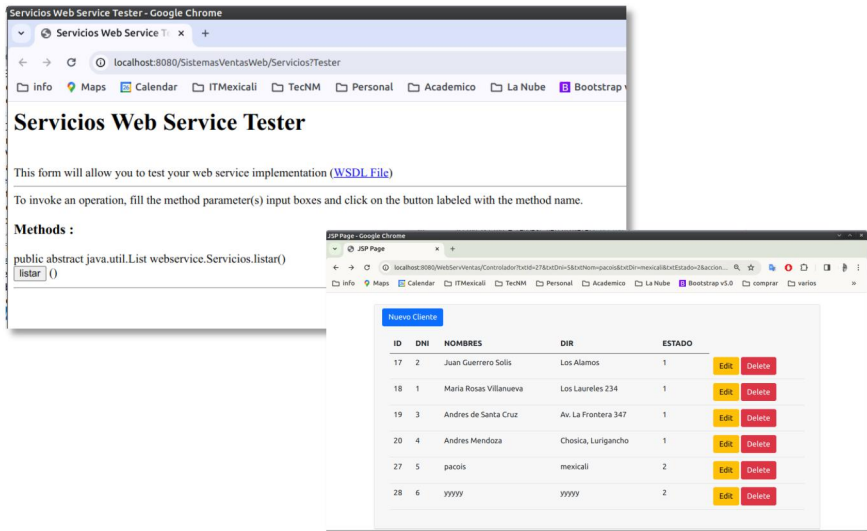
Debe estar en ejecución el proyecto de la aplicación dueña del servicio para poder usar (listar) **Listar Clientes**

Se visualiza la vista generada para mostrar el servicio web (listar) Listar Clientes, esta vista es diferente a la vista de la aplicación que generó el servicio

Nuevo Cliente

ID	DNI	NOMBRES	DIR	ESTADO	
17	2	Juan Guerrero Solis	Los Alamos	1	Edit Delete
18	1	Maria Rosas Villanueva	Los Laureles 234	1	Edit Delete
19	3	Andres de Santa Cruz	Av. La Frontera 347	1	Edit Delete
20	4	Andres Mendoza	Chosica, Lurigancho	1	Edit Delete
27	5	pacois	mexicali	2	Edit Delete
28	6	yyyyy	yyyyy	2	Edit Delete

Ya se tienen los dos proyectos (el que publica el servicio y el que consume)



Se usará el método (add), que se tiene en la clase ClienteDAO del proyecto ventas

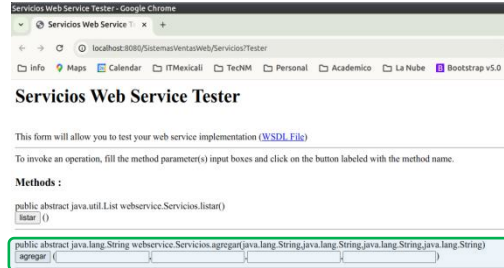
```
// metodo para usar en webService
public String add(String Dni, String Nom, String Dir, String Estado) {
    String sql = "insert into cliente (Dni, Nombres, Direccion, Estado) values (?, ?, ?, ?)";
    try {
        con = classConexion.con();
        pstmt = con.prepareStatement(sql);
        pstmt.setString(parameterIndex: 1, x Dni);
        pstmt.setString(parameterIndex: 2, x Nom);
        pstmt.setString(parameterIndex: 3, x Dir);
        pstmt.setString(parameterIndex: 4, x Estado);
        r = pstmt.executeUpdate();
        if (r == 1) {
            msj = "cliente agregado";
        } else {
            msj = "error";
        }
    } catch (SQLException e) {
        System.out.println("no se agrego cliente" + e.toString());
    }
    return msj;
}
```

Asi como se creo el servicio web listar, se crea el servicio web agregar (add).

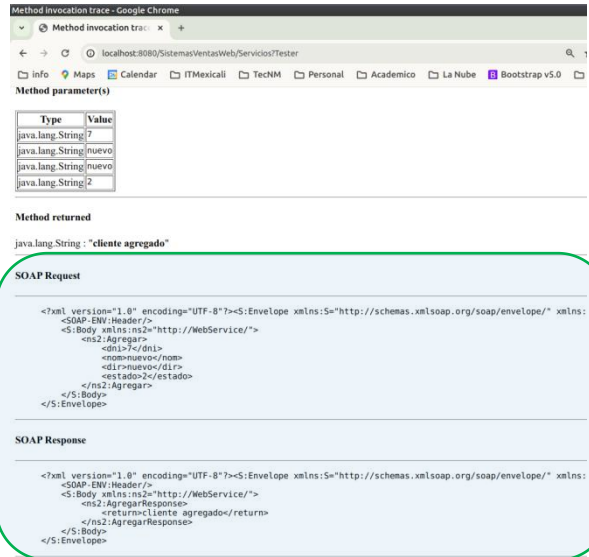


Se despliega el proyecto (deploy), y se prueba el servicio

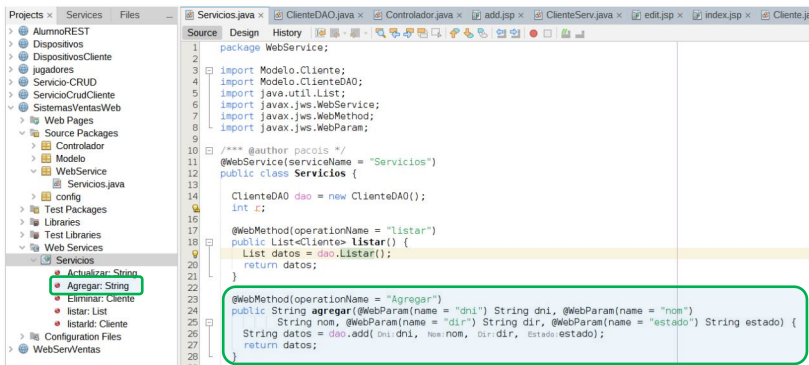
- Click derecho en Servicios
- Test Web Services



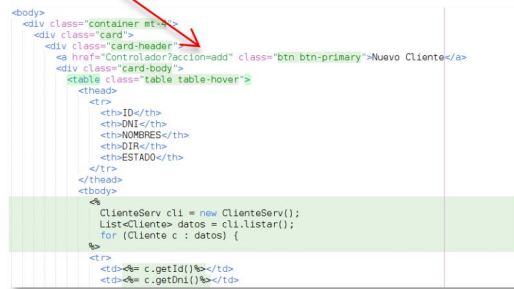
Se introducen los datos y se presiona click en agregar, se podrá verificar que se agregan los datos, señal que el servicio esta listo para su consumo



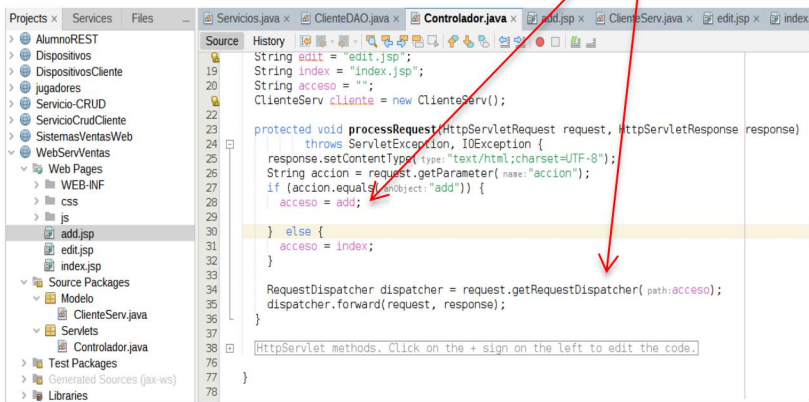
Ya esta el servicio web publicado agregar.



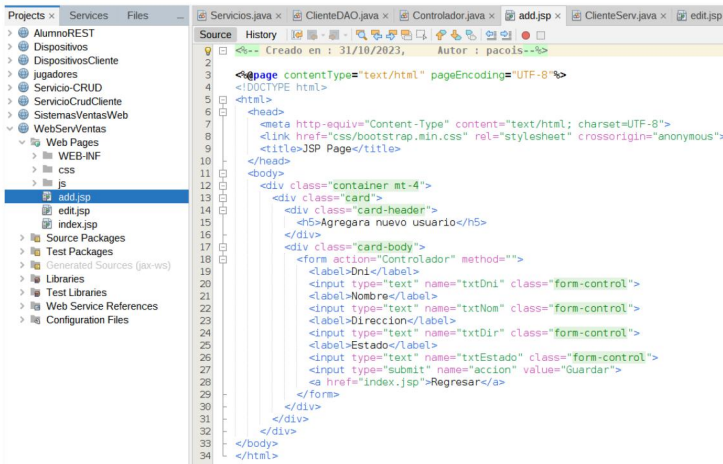
Se agrega la referencia en index.jsp



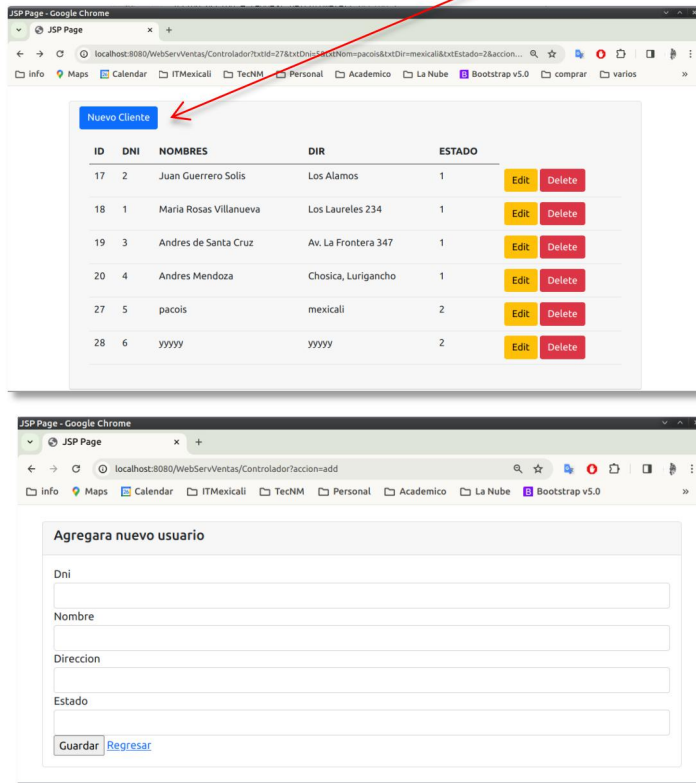
En el control, se obtiene el parámetro accion y si es add ? entonces se redirecciona a la vista add



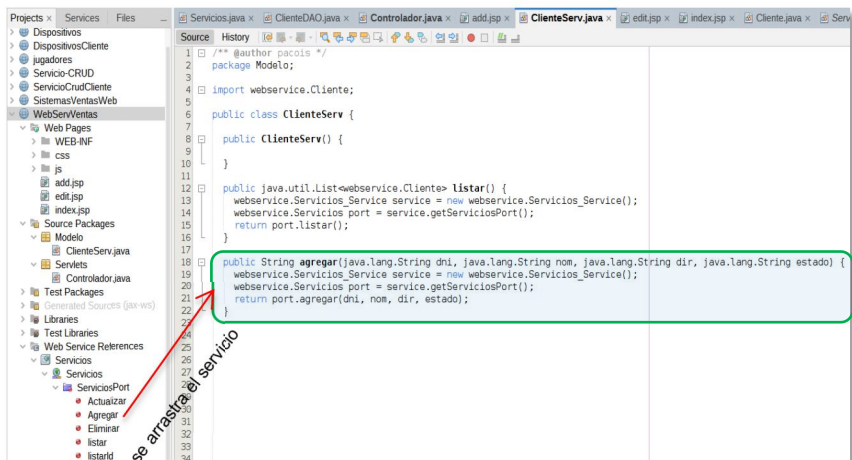
Ahora como se dijo anteriormente se genera una vista(front) para agregar, diferente a la del proyecto ventas.



Si se despliega el (consumidor) y click en botón nuevo cliente y muestra la vista add

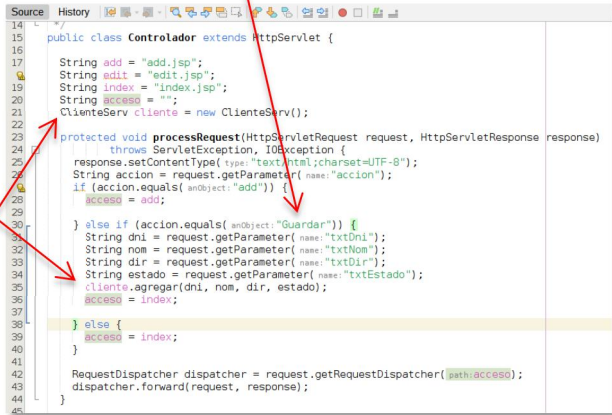


Para usar el servicio (Agregar), dentro de la clase ClienteServ arrastramos el servicio cambiando el acceso a público



En el control se implementa la acción Guardar, para atender botón de vista (Guardar) add.

Se instancia la clase ClienteServ, para usar el método agregar

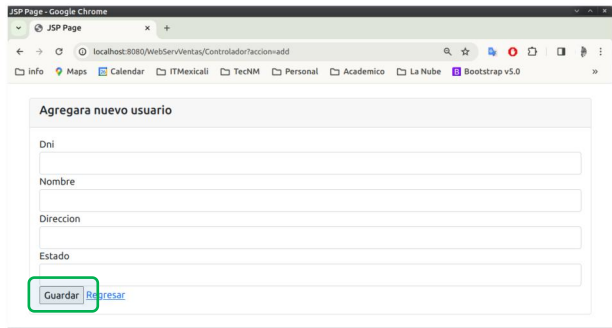


```

14
15
16 public class Controlador extends HttpServlet {
17
18     String add = "add.jsp";
19     String edit = "edit.jsp";
20     String index = "index.jsp";
21     String acceso = "";
22     ClienteServ cliente = new ClienteServ();
23
24     protected void processRequest(HttpServletRequest request, HttpServletResponse response)
25         throws ServletException, IOException {
26         response.setContentType("text/html;charset=UTF-8");
27         String accion = request.getParameter("accion");
28         if (accion.equals("add")) {
29             acceso = add;
30
31         } else if (accion.equals("Guardar")) {
32             String dni = request.getParameter("txtDni");
33             String nom = request.getParameter("txtNom");
34             String dir = request.getParameter("txtDir");
35             String estado = request.getParameter("txtEstado");
36             cliente.agregar(dni, nom, dir, estado);
37             acceso = index;
38
39         } else {
40             acceso = index;
41         }
42
43         RequestDispatcher dispatcher = request.getRequestDispatcher("path/acceso");
44         dispatcher.forward(request, response);
45     }
46 }

```

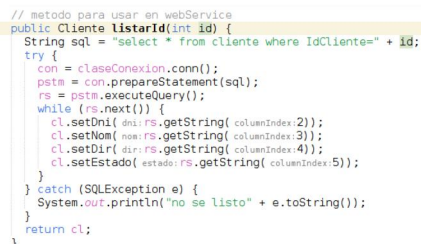
Con esto ya funciona el botón de guardar en la vista agregar.



Lo siguiente en el ejercicio, será la implementación método para editar clientes, para esto se requiere un método para buscar un cliente específico para poder editarlo, por lo que primero se implementa el método para buscar un cliente solamente (listarId) para posteriormente implementar el método para editarlo (edit).

Método de la clase ClienteDAO (listarId)

Se usará el método (listarId), que se tiene en la clase ClienteDAO del proyecto ventas



```

// metodo para usar en webService
public Cliente listarId(int id) {
    String sql = "select * from cliente where IdCliente=" + id;
    try {
        con = claseConexion.conn();
        pstmt = con.prepareStatement(sql);
        rs = pstmt.executeQuery();
        while (rs.next()) {
            cl.setDni( dni:rs.getString( columnIndex:2));
            cl.setNom( nom:rs.getString( columnIndex:3));
            cl.setDir( dir:rs.getString( columnIndex:4));
            cl.setEstado( estado:rs.getString( columnIndex:5));
        }
    } catch (SQLException e) {
        System.out.println("no se listo" + e.toString());
    }
    return cl;
}

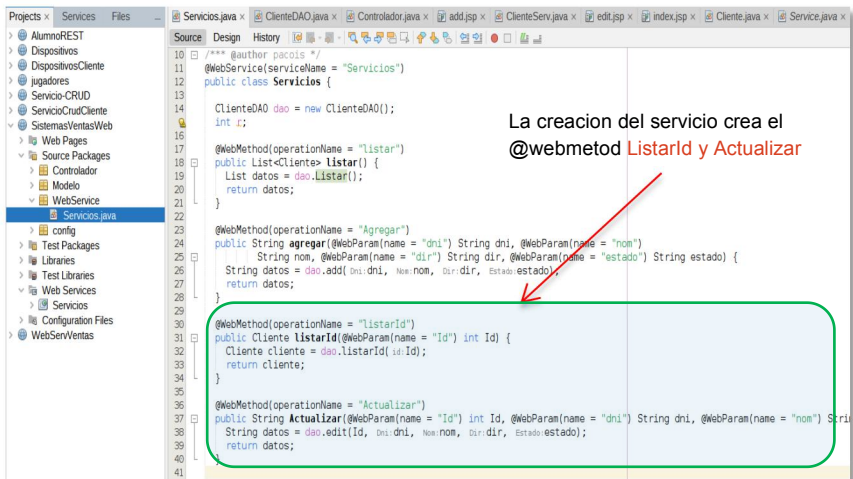
```

Método de la clase ClienteDAO (edit)

Se usará el método (edit), que se tiene en la clase ClienteDAO del proyecto ventas

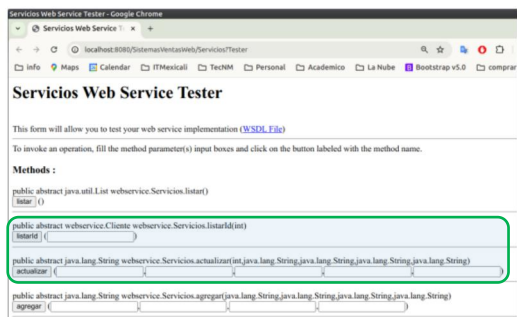
```
// metodo (Actualizar) para usar en webService
public String edit(int Id, String Dni, String Nom, String Dir, String Estado) {
    String sql = "update cliente set Dni=?, Nombre=?, Direccion=?, Estado=? where IDcliente=" + Id;
    try {
        con = claseConexion.con();
        pstn = con.prepareStatement(sql);
        pstn.setString(1, Id);
        pstn.setString(2, Dni);
        pstn.setString(3, Nom);
        pstn.setString(4, Dir);
        pstn.setString(5, Estado);
        r = pstn.executeUpdate();
        if (r == 1) {
            msj = "cliente actualizado";
        } else {
            msj = "error";
        }
    } catch (SQLException e) {
        System.out.println("no se agrego cliente" + e.toString());
    }
    return msj;
}
```

Asi como se creó el servicio web listar, se crea el servicio web para buscar y actualizar (ListarId y Actualizar).



Se despliega el proyecto (deploy), y se prueba el servicio

- Click derecho en Servicios
- Test Web Services



Se introducen los datos y se presiona click en agregar, se podrá verificar que se agregan los datos, señal que el servicio esta listo para su consumo

ListarId

Method parameter(s)

Type	Value
int	27

Method returned

webservice.Ciente: "webservice.Ciente@3c1f9d8e"

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body xmlns:ns2="http://WebService/">
    <ns2:ListarId>
      <Id>27</Id>
    </ns2:ListarId>
  </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body xmlns:ns2="http://WebService/">
    <ns2:ListarIdResponse>
      <return>
        <dir>mexicali</dir>
      </return>
    </ns2:ListarIdResponse>
  </S:Body>
</S:Envelope>
```

Method parameter(s)

Type	Value
int	27
java.lang.String	8
java.lang.String	actualizado
java.lang.String	aaaa
java.lang.String	3

Method returned

java.lang.String: "cliente actualizado"

SOAP Request

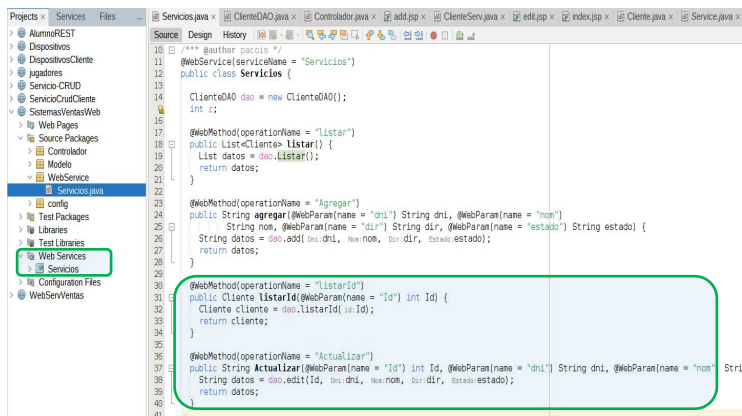
```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body xmlns:ns2="http://WebService/">
    <ns2:Actualizar>
      <Id>27</Id>
      <dni>8</dni>
      <nom>actualizado</nom>
      <dir>aaaa</dir>
      <estado>3</estado>
    </ns2:Actualizar>
  </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body xmlns:ns2="http://WebService/">
    <ns2:ActualizarResponse>
      <return>cliente actualizado</return>
    </ns2:ActualizarResponse>
  </S:Body>
</S:Envelope>
```

Actualizar

Se visualiza el Servicio web publicado ListarId y Actualizar.



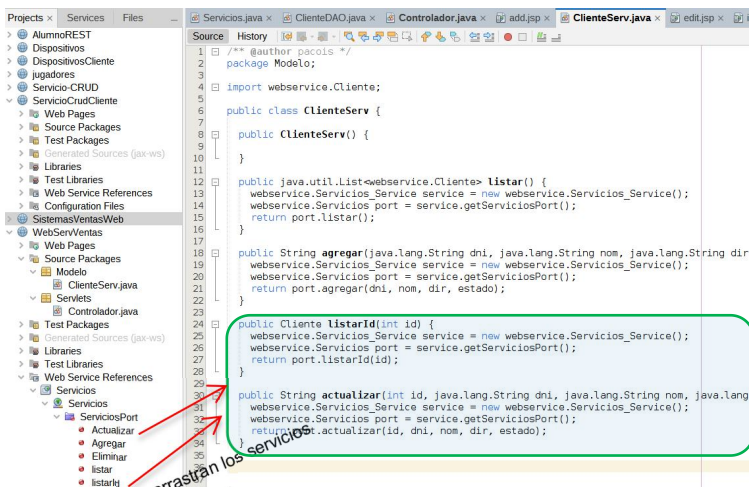
Se agrega la referencia en index.jsp.

```
<tbody>
  <tr>
    <td>
      ClienteServ cli = new ClienteServ();
      List<Cliente> datos = cli.listar();
      for (Cliente c : datos) {
        <!--
        <td>=<%= c.getId()>%></td>
        <td>=<%= c.getDir()>%></td>
        <td>=<%= c.getNom()>%></td>
        <td>=<%= c.getDir()>%></td>
        <td>=<%= c.getEstado()>%></td>
        <td>
          <a href="Controlador?accion=editar&id=<%= c.getId()>%>" class="btn btn-warning">Edit</a>
          <a href="Controlador?accion=eliminar&id=<%= c.getId()>%>" class="btn btn-danger">Delete</a>
        </td>
      </tr>
    </td>
  </tbody>
</table>
```

En la vista edit.jsp, Se bloquea id para no modificarlo.

```
<%page import="Modelo.ClienteServ"%>
<%page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link href="css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous">
<title>JSP Page</title>
</head>
<body>
<div class="container mt-4">
<div class="card">
<div class="card-header">
<h5>Actualizar usuario</h5>
</div>
<div class="card-body">
<form action="Controlador" method="">
<label>Id</label>
<input type="text" name="txtId" value="<%= id %>" readonly="" class="form-control">
<label>Dni</label>
<input type="text" name="txtDni" value="<%= c.getDni() %>" class="form-control">
<label>Nombre</label>
<input type="text" name="txtNom" value="<%= c.getNom() %>" class="form-control">
<label>Direccion</label>
<input type="text" name="txtDir" value="<%= c.getDir() %>" class="form-control">
<label>Estado</label>
<input type="text" name="txtEstado" value="<%= c.getEstado() %>" class="form-control">
<input type="submit" name="accion" value="Actualizar" class="btn btn-success">
<a href="index.jsp" Regresar</a>
</form>
</div>
</div>
</body>
</html>
```

Para usar los servicios (listarId y actualizar), dentro de la clase ClienteServ los arrastramos cambiando el acceso a público



En el control se implementa la acción editar y Actualizar.

```
public class Controlador extends HttpServlet {

    String add = "add.jsp";
    String edit = "edit.jsp";
    String index = "index.jsp";
    String acceso = "";
    ClienteServ cliente = new ClienteServ();

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        String accion = request.getParameter("accion");
        if (accion.equals("add")) {
            acceso = add;
        } else if (accion.equals("guardar")) {
            String dni = request.getParameter("txtDni");
            String nom = request.getParameter("txtNom");
            String dir = request.getParameter("txtDir");
            String estado = request.getParameter("txtEstado");
            cliente.agregar(dni, nom, dir, estado);
            acceso = index;
        } else if (accion.equals("editar")) {
            acceso = edit;
            request.setAttribute("idCliente", request.getParameter("id"));
        } else if (accion.equals("Actualizar")) {
            String dni = request.getParameter("txtDni");
            String nom = request.getParameter("txtNom");
            String dir = request.getParameter("txtDir");
            String estado = request.getParameter("txtEstado");
            int id = Integer.parseInt(request.getParameter("txtId"));
            cliente.actualizar(id, dni, nom, dir, estado);
            acceso = index;
        }
    }
}
```

Si se despliega el (consumidor) y click en botón edit para mostrar la vista de edición

ID	DNI	NOMBRES	DIR	ESTADO	
17	2	Juan Guerrero Solis	Los Alamos	1	Edit Delete
18	1	Maria Rosas Villanueva	Los Laureles 234	1	Edit Delete
19	3	Andres de Santa Cruz	Av. La Frontera 347	1	Edit Delete
20	4	Andres Mendoza	Chosica, Lurigancho	1	Edit Delete
27	5	pacois	mexicali	2	Edit Delete
28	6	yyyyy	yyyyy	2	Edit Delete

Actualizar usuario

ID: 28

Dni: 6

Nombre: nuevo

Direccion: nuevo

Estado: 2

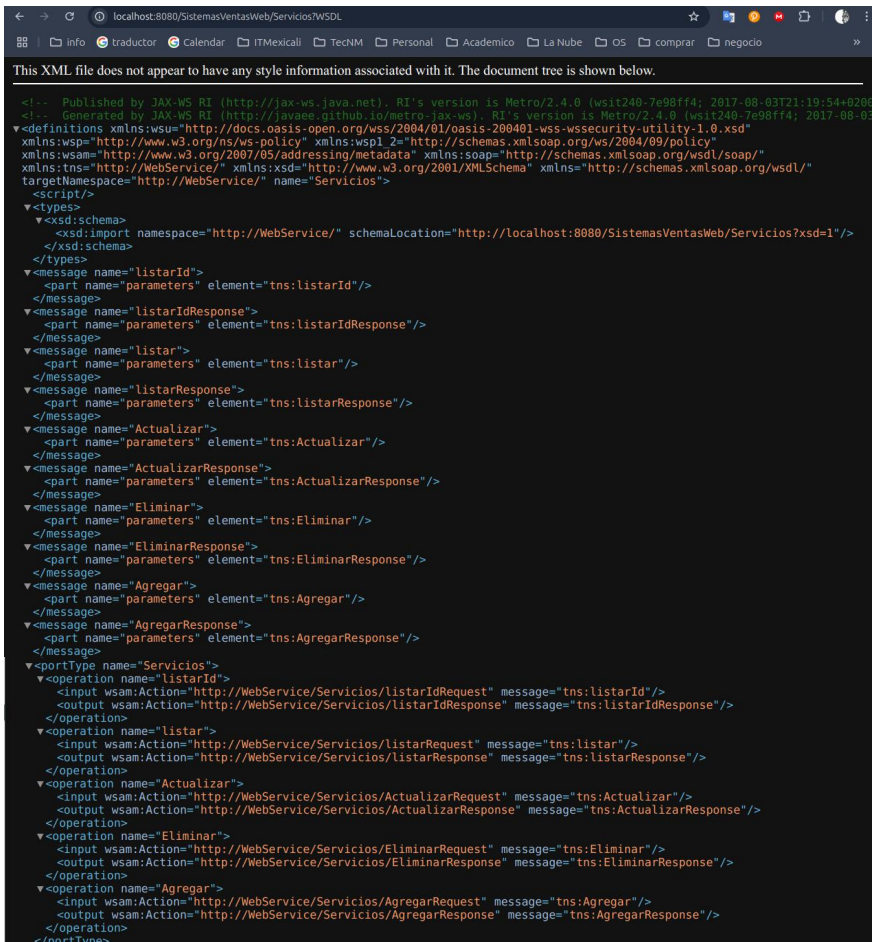
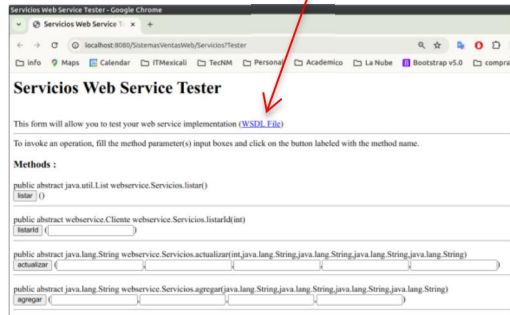
[Actualizar](#) [Regresar](#)

Ejemplo de edición del Id 28 y se revisa el funcionamiento al dar click en botón Actualizar

ID	DNI	NOMBRES	DIR	ESTADO	
17	2	Juan Guerrero Solis	Los Alamos	1	Edit Delete
18	1	Maria Rosas Villanueva	Los Laureles 234	1	Edit Delete
19	3	Andres de Santa Cruz	Av. La Frontera 347	1	Edit Delete
20	4	Andres Mendoza	Chosica, Lurigancho	1	Edit Delete
27	8	actualizado	aaaa	3	Edit Delete
28	6	nuevo	nuevo	2	Edit Delete

Como se mencionó al inicio de este capítulo podemos visualizar el descriptor del servicio, acccesando el descriptor **WSDL** file, con sus apartados

```
<?xml version="1.0">
<definitions>
  <types>
    ...
  </types>
  <message>
    ...
  </message>
  <portType>
    ...
  </portType>
  <binding>
    ...
  </binding>
</definitions>
```



```

<?xml version="1.0">
<definitions>
  <types>
    ...
  </types>
  <message>
    ...
  </message>
  <portType>
    ...
  </portType>
  <binding>
    ...
  </binding>
</definitions>

```

```

▼<binding name="ServiciosPortBinding" type="tns:Servicios">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  ▼<operation name="listarId">
    <soap:operation soapAction=""/>
    ▼<input>
      <soap:body use="literal"/>
    </input>
    ▼<output>
      <soap:body use="literal"/>
    </output>
  </operation>
  ▼<operation name="listar">
    <soap:operation soapAction=""/>
    ▼<input>
      <soap:body use="literal"/>
    </input>
    ▼<output>
      <soap:body use="literal"/>
    </output>
  </operation>
  ▼<operation name="Actualizar">
    <soap:operation soapAction=""/>
    ▼<input>
      <soap:body use="literal"/>
    </input>
    ▼<output>
      <soap:body use="literal"/>
    </output>
  </operation>
  ▼<operation name="Eliminar">
    <soap:operation soapAction=""/>
    ▼<input>
      <soap:body use="literal"/>
    </input>
    ▼<output>
      <soap:body use="literal"/>
    </output>
  </operation>
  ▼<operation name="Agregar">
    <soap:operation soapAction=""/>
    ▼<input>
      <soap:body use="literal"/>
    </input>
    ▼<output>
      <soap:body use="literal"/>
    </output>
  </operation>
</bindings>
▼<service name="Servicios">
  ▼<port name="ServiciosPort" binding="tns:ServiciosPortBinding">
    <soap:address location="http://localhost:8080/SistemasVentasWeb/Servicios"/>
  </port>
</service>
</definitions>

```

Tecnologías actuales en patrones de arquitecturas de APIs

Las API van más allá de un simple intercambio de datos son partes fundamentales en la arquitectura de microservicios, permitiendo que las aplicaciones sean más modulares y escalables. Han evolucionado significativamente en las últimas décadas, convirtiéndose en elementos clave en el desarrollo de software moderno.

Las arquitecturas de APIs han pasado de enfoques monolíticos y propietarios a paradigmas más flexibles, escalables y eficientes.

1. **REST** (Representational State Transfer), es uno de los patrones más utilizados para el diseño de APIs debido a su simplicidad y compatibilidad con HTTP. Se basa en principios como la separación entre cliente y servidor, la ausencia de estado y una estructura bien definida de recursos. Tecnologías actuales asociadas:

Frameworks: Express.js (Node.js), Spring Boot (Java), Flask (Python)
Herramientas: Postman, Swagger (OpenAPI)
Protocolos: HTTP, HTTPS

2. **GraphQL**, desarrollado por Facebook, permite a los clientes definir la estructura de los datos que desean recibir, mejorando la eficiencia en comparación con REST al reducir la sobrecarga de datos innecesarios. Tecnologías actuales asociadas:

Frameworks: Apollo Server, GraphQL.js, Hasura
Lenguajes compatibles: JavaScript, Python, Java, Go
Herramientas: GraphQL, GraphQL Playground

3. **gRPC** (Google Remote Procedure Call), es un framework de RPC moderno que utiliza HTTP/2 para mejorar la eficiencia en la comunicación entre servicios. Es especialmente útil en arquitecturas de microservicios debido a su alto rendimiento y compatibilidad con múltiples lenguajes. Tecnologías actuales asociadas:

Implementaciones: gRPC (Google), grpc-java, grpc-go
Protocolos: Protocol Buffers (protobuf)
Integraciones: Kubernetes, Envoy Proxy

4. **API Gateway**, actúa como un intermediario entre los clientes y los servicios backend, proporcionando funcionalidades como autenticación, balanceo de carga y gestión de solicitudes. Tecnologías actuales asociadas:

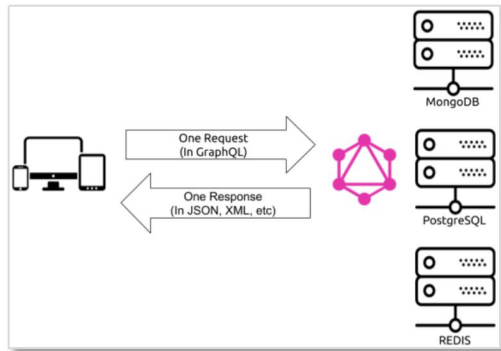
Herramientas populares: Kong, Apigee, AWS API Gateway, NGINX
Integraciones: OAuth 2.0, JWT (JSON Web Tokens)

Aunque en el capítulo anterior se trató el tema de REST, vale la pena revisar GraphQL, dado que es una tecnología que sigue las huellas de los patrones arquitectónicos de las APIs RESTful

GraphQL fue tecnología emergente hace algunos años, y actualmente está en un proceso de consolidación dentro de las arquitecturas de APIs modernas. No es una novedad, pero su adopción sigue creciendo y evolucionando

Ventajas de GraphQL:

- "Evita la sobrecarga de datos (overfetching) y la falta de datos (underfetching)."
- "Proporciona un esquema claro y predecible, facilitando el desarrollo y la documentación."
- "Permite a los clientes solicitar solo los datos que necesitan, adaptándose a diferentes necesidades."
- "Se reducen los tiempos de desarrollo dado que el front end, define los datos que necesita."
- "El cliente define exactamente los datos que necesita"



Desventajas de GraphQL:

- "Puede requerir un mayor esfuerzo inicial para aprender y configurar."
- "Las consultas muy anidadas pueden volverse difíciles de manejar."

Si se compara Rest y GraphQL por ejemplo para obtener información de un profesor, sus cursos y lista de alumnos inscritos en ellos, se verá que :

REST se basa en endpoints fijos que representan recursos y para obtener datos relacionados, se suelen hacer múltiples llamadas, se tendrían 5 peticiones HTTP

GET /profesores/123 → obtiene los datos del profesor.

GET /profesores/123/cursos → obtiene los cursos del profesor.

GET /cursos/101/alumnos

GET /cursos/102/alumnos

GET /cursos/103/alumnos

GraphQL permite consultas estructuradas donde se puede solicitar todo lo necesario en una sola petición.

Consulta típica:

```

query {
  profesor(id: "123") {
    nombre
    cursos {
      nombre
      alumnos {
        nombre
        matricula
      }
    }
  }
}

```

Característica	REST	GraphQL
Peticiones	Múltiples (5 en el ejemplo)	Una sola
Control sobre los datos	Límite lo que expone cada endpoint	se define lo que se quiere
Overfetching (datos de más)	Común, porque los endpoints retornan todo el objeto	solo se pide lo que se necesita
Underfetching (datos de menos)	Común: requiere más llamadas	se especifican todos los datos necesarios

<<Tabla comparativa de Rest y GraphQL>>

En REST, cada llamada implica latencia y headers repetidos.

En GraphQL se definen los tipos

```

type Profesor {
  id: ID!
  nombre: String!
  cursos: [Curso!]!
}

type Estudiante {
  id: ID!
  nombre: String!
  cursos: [Curso!]!
}

type Curso {
  id: ID!
  titulo: String!
  profesor: Profesor!
  estudiantes: [Estudiante!]!
}

```

En GraphQL, una sola llamada puede ahorrar mucho en contextos de red lenta, implicando escalabilidad y rendimiento de una API en aplicaciones móviles o con mala conexión a internet.

Otra arquitectura es **gRPC** es un framework de comunicación RPC

(Remote Procedure Call) de alto rendimiento desarrollado por Google, permite que servicios en diferentes máquinas se comuniquen directamente mediante llamadas a procedimientos como si fueran funciones locales, se basa en HTTP/2 como protocolo de transporte así como Protocol Buffers (Protobuf) como lenguaje de definición de interfaces (IDL) y formato de serialización de datos.

Características principales:

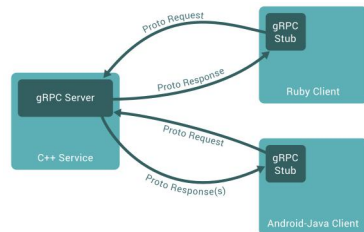
- Comunicación eficiente y rápida: gracias a Protobuf y HTTP/2.
- Soporta streaming nativo unidireccional y bidireccional.
- Su interoperabilidad permite que funcione en múltiples lenguajes (Java, Python, Go, C#, etc.).
- Escalabilidad y balanceo de carga es Ideal para microservicios modernos.
- Seguridad integrada al usar TLS por defecto.
- Tipado fuerte que valida contratos a nivel de compilación.

Característica	REST	gRPC
Protocolo	HTTP/1.1	HTTP/2
Formato de datos	JSON (texto, menos eficiente)	Protobuf (binario, eficiente)
Streaming	Limitado	Soportado nativamente
Rendimiento	Medio a bajo	Alto
Soporte de herramientas	Amplio ecosistema	Más limitado, pero robusto
Humanamente legible	Sí (JSON)	No (binario)

<<Tabla: Diferencia entre SOA y APIs>>

Ventajas de gRPC

- Más rápido y eficiente que REST.
- Excelente para comunicación entre servicios internos.
- Contratos bien definidos mediante proto.
- Fácil generación de clientes/servidores.



Desventajas de gRPC

- No es tan legible ni amigable como JSON.
- No es ideal para aplicaciones públicas expuestas por navegador (sin adaptación).
- Menor soporte de herramientas en comparación con REST.

Si se compara Rest y gRPC, ejemplo si es una empresa de envíos y necesita que la app de logística consulte la base central de paquetes se vera que :

REST: hace una petición “GET /paquetes/123” y se recibe JSON.

gRPC: llama al método ObtenerPaquete(id) y se recibe un objeto Paquete, con validación de tipos en tiempo de compilación y menor tiempo de respuesta.

Tendencias Emergentes en APIs

- a) **APIs Serverless**, eliminan la necesidad de gestionar infraestructura, permitiendo a los desarrolladores enfocarse en la lógica del negocio.

Tecnologías asociadas:

AWS Lambda, Azure Functions, Google Cloud Functions

Frameworks: Serverless Framework, AWS SAM

- b) **APIs Basadas en Eventos**, este enfoque utiliza arquitecturas orientadas a eventos para mejorar la reactividad y eficiencia del sistema.

Tecnologías asociadas:

Apache Kafka, AWS EventBridge, Google Cloud Pub/Sub

Modelos de mensajería: AMQP, MQTT

Las arquitecturas de APIs continúan evolucionando con tecnologías más eficientes y seguras. REST sigue siendo dominante, pero GraphQL, gRPC y los API Gateways han ganado relevancia en escenarios específicos. La integración con servicios serverless y eventos está marcando la próxima generación de arquitecturas de APIs.

Referencias

Hostinger. (s.f.). GraphQL vs REST: ¿cuál es el mejor para el desarrollo API?.
<https://www.hostinger.es/blog/graphql-vs-rest-en-hostinger>

GraphQL frente a REST: ¿Qué arquitectura de diseño de API es adecuada para su negocio?.
<https://www.astera.com/es/type/blog/graphql-vs-rest-api/>

¿Qué es GraphQL? Ejemplos y ¿por qué cambiar de API REST?.
<https://lab.wallarm.com/what/protocolo-graphql/?lang=es>

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). WEB SERVICES: CONCEPTS, ARCHITECTURES AND APPLICATIONS. Springer.

Erl, T. (2005). SERVICE-ORIENTED ARCHITECTURE: CONCEPTS, TECHNOLOGY, AND DESIGN. Prentice Hall.

Papazoglou, M. P. (2012). WEB SERVICES: PRINCIPLES AND TECHNOLOGY. Pearson Education.

World Wide Web Consortium (W3C). (2008). EXTENSIBLE MARKUP LANGUAGE (XML) 1.0 (FIFTH EDITION).
<https://www.w3.org/TR/xml/>

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. PROCEEDINGS OF THE 17TH INTERNATIONAL WORLD WIDE WEB CONFERENCE (WWW 2008).

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. J., & Nielsen, H. F. (2007). SOAP Version 1.2 Part 1: Messaging Framework. W3C RECOMMENDATION. Disponible en:
<https://www.w3.org/TR/soap12/>

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.

GraphQL Foundation. (2024). GraphQL Specification.
<https://graphql.org>

Google. (2024). gRPC: A High Performance, Open Source RPC Framework.
<https://grpc.io>

AWS. (2024). Amazon API Gateway.
<https://aws.amazon.com/api-gateway>

NGINX. (2024). API Gateway Solutions.
<https://www.nginx.com>

Oracle.
<https://www.oracle.com/mx/tools/technologies/netbeans-ide.html>

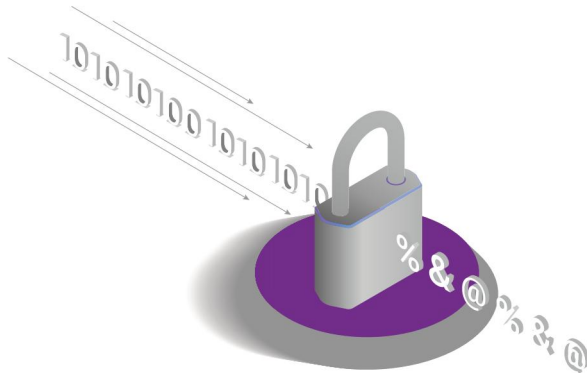
AWS what is java
<https://aws.amazon.com/es/what-is/java/>

CAPÍTULO 4

Seguridad en la Integración de Servicios

Protegiendo Servicios y Datos

El objetivo de este capítulo es revisar estrategias de cifrado y medidas de protección de datos sensibles durante las transacciones entre sistemas y APIs.





En la actualidad hay muchas empresas que utilizan APIs, diseñadas para ser canales de comunicación sencillos y rápidos entre distintas plataformas, y los delincuentes digitales las han identificado como posible vector de ataque, usándolas con frecuencia para llevar a cabo afectaciones en línea, maximizando la exposición de vulnerabilidades en la seguridad de muchas empresas.

Los incidentes de seguridad de API han aumentado significativamente en los últimos años, el 84% de los profesionales de seguridad han experimentado un ataque de API en el último año, este aumento se debe a la creciente dependencia de las APIs en las aplicaciones web y móviles, lo que las convierte en un objetivo atractivo para la delincuencia digital.

El sector financiero es el más afectado por el aumento en los ataques a APIs, la explotación de vulnerabilidades en seguridad y el abuso de la lógica empresarial, ponen en riesgo a las organizaciones y la confidencialidad de la información.

Tipos de ataques:

- **Ataques de Inyección:** Los atacantes aprovechan vulnerabilidades en las APIs para inyectar código malicioso.
- **Ataques de Máquina Intermediaria:** Los atacantes interceptan la comunicación entre APIs para robar datos.
- **Ataques DDoS:** Las APIs se inundan con tráfico para interrumpir el servicio.
- **Abuso de la Lógica Empresarial:** Los atacantes manipulan las APIs para obtener acceso no autorizado o beneficios económicos.
- **Ataques a la Autenticación:** Se aprovechan debilidades en los mecanismos de autenticación para obtener acceso a las APIs.
- **Consumo de Recursos sin Restricciones:** Los atacantes consumen grandes cantidades de recursos de las APIs, afectando su rendimiento.

Vulnerabilidades comunes en APIs

- Inyección SQL, los atacantes insertan comandos SQL maliciosos a través de las APIs, se mitiga con validación y saneamiento de entradas.
- Cross-Site Request Forgery (CSRF), Los atacantes inducen a los usuarios a ejecutar acciones no deseadas, se mitiga con el uso de tokens CSRF en formularios y validaciones.

Riesgos de los ataques:

- **Robo de Datos,** acceso a información confidencial, como datos de clientes o información financiera.
- **Interrupción del Servicio,** los ataques pueden interrumpir las operaciones de las organizaciones, generando pérdidas económicas y daños a la reputación.

- **Afectación al Rendimiento**, los ataques pueden consumir recursos de la API, afectando su rendimiento y disponibilidad.
- **Apertura de Puertas a Otros Ataques**, los ataques a APIs pueden facilitar el acceso a otros sistemas y redes.

Más del 70% del tráfico web está enfocado a llamadas de APIs, muchas organizaciones protegen el tráfico de APIs de la misma manera como protegen sus aplicaciones web o aplicaciones legacy, siendo que el enfoque de seguridad para las APIs es diferente, dado que las arquitecturas son diferentes.

a) La arquitectura de la web estándar, funciona a través de peticiones Http, donde hay un perímetro controlado (acceso de usuarios), se utilizan protocolos y puertos estándar (tcp 80 y 443), hay mejor control de acceso a la información (menor exposición) y los WAF (web application firewall), permiten mas facilidad de configuración funcionan sobre firmas, por lo que se tiene patrones pre-identificados con los ataques mas comunes y se pueden bloquear fácilmente.

b) En la arquitectura de APIs existen muchas vulnerabilidades ya que con un llamado a una api se puede exponer mucha información por lo que hay deficiencia en el monto de información que se debe exponer, además se habla de otros protocolos diferentes no tan estándar (SOAP, GraphQL, Rest, JSON) por lo que exige conocimiento adicional, tampoco se requiere una interface para exposición visual (browser) son peticiones y se pueden hacer a través de plataformas que trabajan APIs por ejemplo postman para poder validar o hacer auditorias, con esto aumenta el número de puntos de entrada hacia los backend, al tener exposición de una api se tienen varios endpoints públicos y se puede facilitar el consumo, si se exponen endpoints aumenta la posibilidad de que alguien intente hacer peticiones a través de los mismos.



Con la diferencia de arquitecturas se puede decir que por lo menos un 40% de APIs en la web no son gestionadas adecuadamente, y por tanto son potencialmente vulnerables.

Mitos de seguridad en APIs

Api Gateways, Web Applications Firewalls y otras herramientas son suficientes para la seguridad de APIs.

Api Gateway facilita ciertas características, y WAF está basado en firmas pero no ofrece total visibilidad de endpoints, estas facilitan la gestión (manejo de inventario, control de peticiones, etc..), pero no son herramientas que están enfocadas 100% a seguridad.

La seguridad en APIs es simple

El concepto va mas allá de conectar aplicaciones, como atender protocolos adecuadamente y esto puede abrir puertas a retos mayores en la parte de seguridad.

La seguridad es parte del diseño y desarrollo de las APIs

Las herramientas que normalmente se utilizan en un desarrollo estándar, como las de análisis:

SAST (Static Application Security Testing) son herramientas que analizan el código fuente o binario de una aplicación para identificar posibles vulnerabilidades de seguridad sin ejecutarla.

DAST (Dynamic Application Security Testing) es el proceso de analizar una aplicación web a través del front-end para encontrar vulnerabilidades mediante ataques simulados.

Tienen deficiencia ya que no están diseñadas para escanear APIs, están enfocadas a aplicaciones web.

Los proveedores de nube aseguran las APIs

Los proveedores ofrecen soluciones, pero la responsabilidad de la configuración es del propietario de las APIs.

zero trust es la solución

Los proveedores de servicios de nube hablan de zero trust pero no es la solución al 100% ya que hablar de APIs security incluye diferentes etapas, muchas están enfocadas en conceptos de diseño, desarrollo, arquitectura y todo es un ambiente integral que permite hablar de tener un nivel de seguridad en las APIs.

Protección de datos sensibles

Este capítulo se centra en las mejores prácticas de seguridad para APIs, desde la autenticación y autorización hasta el cifrado de datos, así como las técnicas para proteger las APIs contra ataques de inyección, y otros riesgos comunes en aplicaciones distribuidas.

En los entornos de sistemas distribuidos, el intercambio de información entre APIs y servicios es una actividad crítica. Sin embargo, este intercambio involucra datos sensibles que deben ser protegidos contra accesos no autorizados, alteraciones maliciosas y fugas accidentales.

Los DATOS SENSIBLES son los que no se deben divulgar sin consentimiento ya que se verían comprometidos, y pondrán en riesgo la privacidad de las personas, la seguridad de una organización o el cumplimiento normativo.

Los datos sensibles incluyen información personal (como nombres, direcciones, números de identificación), credenciales de autenticación, detalles financieros y datos de salud.

El cifrado de datos es un proceso que protege los datos sensibles, desde tiempos antiguos, las personas convertían mensajes en códigos (o texto cifrado) para proteger la información oculta. La única forma de leerlo era encontrar la clave para decodificar el texto. Este enfoque se usa hoy en día y se llama cifrado de datos.

La seguridad es uno de los aspectos más críticos cuando se trata de integrar servicios a través de APIs. Dado que las APIs permiten la exposición de datos y funcionalidades a través de internet, se convierten en un objetivo atractivo para ataques. Por esta razón, garantizar que las APIs sean seguros y confiables es esencial para proteger tanto los datos sensibles como las operaciones críticas de los servicios.

Los algoritmos de cifrado se desarrollan continuamente para proporcionar protección segura para datos sensibles y abordar amenazas modernas, además, existen regulaciones de protección de datos que controlan cómo se gestionan los datos personales, por mencionar algunos están:

- El Reglamento General de Protección de Datos (GDPR) en Europa
- La Ley Federal de Protección de Datos Personales en Posesión de los Particulares en México
- La Ley de Privacidad del Consumidor de California (CCPA) en Estados Unidos.

Este capítulo es sobre la seguridad en la integración por lo tanto se atenderán las estrategias de Cifrado y Protección de Datos Sensibles en Transacciones entre Sistemas y APIs.

Tipos de cifrado de datos:

- Cifrado simétrico, que utiliza la misma clave tanto para cifrar como para descifrar datos. Se usa ampliamente en múltiples aplicaciones, incluyendo VPNs, intercambio de archivos, almacenamiento de datos, y más.
- Cifrado asimétrico, que implica un par de claves: una clave pública y una clave privada. Este tipo de cifrado se usa en diferentes contextos que el cifrado simétrico, como para firmas digitales y blockchains.

La protección de datos sensibles durante las transacciones entre sistemas y APIs es un tema crucial en el diseño y operación de arquitecturas orientadas a servicios (SOA) y servicios basados en APIs. Las organizaciones deben garantizar que la información intercambiada entre servicios esté protegida contra accesos no autorizados, manipulaciones o divulgación.

Para proteger adecuadamente los datos durante las transacciones entre sistemas y APIs, se deben seguir los siguientes principios fundamentales:

- **Cifrado de Datos en Tránsito:** Utilizar protocolos seguros como HTTPS (TLS/SSL) para cifrar los datos mientras se transmiten entre sistemas.
- **Cifrado de Datos en Reposo:** Asegurar que los datos almacenados en bases de datos u otros sistemas de almacenamiento estén cifrados.
- **Autenticación y Autorización:** Implementar controles robustos de autenticación (como OAuth 2.0) y autorización para garantizar que solo las entidades autorizadas puedan acceder a las APIs.
- **Gestión de Credenciales Seguras:** Utilizar sistemas seguros para almacenar y gestionar claves API, contraseñas y otros secretos (por ejemplo, gestores de secretos como AWS Secrets Manager o Azure Key Vault).
- **Registro y Monitoreo de Actividades:** Implementar un sistema de monitoreo y registro de las actividades de las APIs para detectar posibles accesos no autorizados o comportamientos anómalos.
- **Control de Acceso Basado en Políticas:** Aplicar políticas de control de acceso que definan qué usuarios o sistemas pueden interactuar con las APIs y bajo qué condiciones.
- **Evaluación Continua de Vulnerabilidades:** Realizar pruebas de penetración y evaluaciones de seguridad periódicas para identificar y mitigar vulnerabilidades.

Las organizaciones ya se encaminan hacia la transformación digital y necesitan una sólida base de TI para empoderar a sus empleados, mejorar la satisfacción del cliente y destacarse de la competencia.

En el mundo digital actual, el rol de un proveedor de servicios es más crucial que nunca. Facilitan el funcionamiento fluido de las operaciones en todos los sectores, apoyan a las empresas con experiencia y empoderan a las personas con soluciones que mejoran su calidad de vida.

Los servicios gestionados por un proveedor experimentado, ayudan a las organizaciones a centrarse en sus objetivos principales, acceder a las habilidades y la experiencia que necesitan, optimizar los gastos operativos, permitir una escalabilidad fluida y evitar costosos tiempos de inactividad, todo ello a la vez que ofrecen servicios de TI fiables, garantizan el cumplimiento normativo y la seguridad 24/7 y permiten maximizar el retorno de la inversión en TI.

Los proveedores de servicios en la nube, o CSPs (Cloud Service Providers), son plataformas que ofrecen servicios de computación a través de Internet, permitiendo a las empresas acceder a infraestructura, plataformas y software sin necesidad de gestionarlos ellos mismos, los servicios se ofrecen bajo diferentes modelos, como IaaS (Infraestructura como Servicio), PaaS (Plataforma como Servicio) y SaaS (Software como Servicio)

Algunos proveedores de Servicios en la Nube:

Amazon Web Services (AWS): Ofrece una amplia gama de servicios en la nube, incluyendo computación, almacenamiento, bases de datos, análisis de datos, IA/ML, y más.

Microsoft Azure: Ofrece una plataforma en la nube versátil con servicios similares a AWS, incluyendo computación, almacenamiento, bases de datos, análisis de datos, y servicios de Azure DevOps.

Google Cloud Platform (GCP): Conocida por su capacidad en IA/ML y análisis de datos, GCP ofrece servicios en la nube como computación, almacenamiento, bases de datos, y análisis.

IBM Cloud: Ofrece una plataforma en la nube híbrida y multinube, con servicios en computación, almacenamiento, bases de datos, análisis de datos, y servicios de desarrollo.

Oracle Cloud: Se enfoca en servicios de base de datos y análisis, junto con servicios de computación y almacenamiento.

Alibaba Cloud: Un importante proveedor en el mercado asiático, ofrece una amplia gama de servicios en la nube, incluyendo computación, almacenamiento, bases de datos, y servicios de AI.

DigitalOcean: Se centra en soluciones de computación en la nube para desarrolladores y empresas de rápido crecimiento.

VMware: Ofrece soluciones de nube híbrida y gestión de infraestructura de nube.

Rackspace: Un proveedor de servicios de nube administrados, conocido por su experiencia en la gestión de infraestructura en la nube.

Huawei Cloud: Ofrece soluciones en la nube para empresas y desarrolladores, con un enfoque en la conectividad y la colaboración.

Por lo anterior se estará haciendo referencia a AWS dado que es de los principales proveedores de web services, y pensando en que la mayoría de los proveedores de servicios proporcionan experiencia, seguridad y tecnologías de información necesarios.

AWS proporciona una variedad de herramientas y servicios para asegurar datos, tanto en tránsito como en reposo, además de controlar estrictamente quién accede a ellos y cómo.

Cifrado de Datos

Cuando se transfieren a través de redes, los datos están en riesgo de exposición a terceros. Por lo tanto, los datos deben cifrarse, preferiblemente usando cifrado de extremo a extremo.

1. El **cifrado en tránsito** (Transport Layer Encryption) garantiza que los datos que viajan entre un cliente y un servidor, o entre servicios de backend, estén protegidos mediante protocolos seguros como HTTPS (TLS/SSL).

Cuando se hace una petición a una API (ejemplo, GET `https://api.banco.com/cuenta`), esa información viaja por internet. Para evitar que alguien la vea o la modifique, se usa:

TLS (Transport Layer Security): protocolo que cifra las comunicaciones entre cliente y servidor.

HTTPS: HTTP sobre TLS. Todas las APIs serias lo usan hoy.

El cliente y el servidor negocian una clave temporal, todos los datos que se envían se cifran con esa clave, así nadie en medio puede leer lo que se transmite, aunque vea los paquetes.

En AWS, esto se implementa automáticamente con API Gateway + AWS Certificate Manager.

- Amazon API Gateway soporta HTTPS por defecto.
- Elastic Load Balancer (ELB) puede manejar la terminación TLS para distribuir tráfico seguro.
- AWS Certificate Manager (ACM) permite gestionar certificados SSL/TLS fácilmente.
- Puedes configurar CloudFront como CDN que también aplica HTTPS a las peticiones.

Buenas prácticas

- Desactivar TLS 1.0 y 1.1.
- Utilizar certificados emitidos por ACM con renovación automática.
- Usar políticas de seguridad TLS estrictas (por ejemplo, TLS 1.2 o superior y cifrados fuertes como AES-256-GCM).

2. El **cifrado en reposo** (At-Rest Encryption) protege los datos almacenados en bases de datos, archivos u otros recursos para que no puedan ser leídos por terceros, incluso si obtienen acceso físico al almacenamiento. Para proteger este tipo de datos, se necesita cifrar todos los datos almacenados, controlar los permisos de acceso, crear regularmente copias de seguridad y almacenarlas en lugares físicos seguros, implementar firewalls, parches de seguridad y actualizaciones, etc.

Una vez que los datos llegan a su destino (como una base de datos, almacenamiento S3 o un log), deben almacenarse de forma segura:

- Se usan algoritmos como AES-256 para cifrarlos.
- Las claves de cifrado son gestionadas por servicios como AWS KMS (Key Management Service).
- El acceso a las claves está estrictamente controlado por políticas de seguridad.

Por ejemplo en AWS puedes activar cifrado en DynamoDB, S3, RDS y otros servicios con solo unos clics o una línea de código. El sistema se encarga del cifrado y descifrado automáticamente.

- Amazon S3:
 - ✓ SSE-S3: cifrado con claves gestionadas por AWS.
 - ✓ SSE-KMS: cifrado con claves gestionadas por AWS KMS (más control y auditoría).
 - ✓ SSE-C: el cliente proporciona y gestiona la clave.
- Amazon RDS y DynamoDB:
 - ✓ Cifrado con claves gestionadas por AWS KMS, habilitable al momento de creación.
- AWS Secrets Manager:
 - ✓ Almacenamiento seguro de credenciales API, claves de base de datos y más. Cifrado automático con KMS.

Buenas prácticas

- Usar SSE-KMS para control detallado del acceso a claves.
- Activar la rotación automática de claves en AWS KMS.
- Aplicar políticas estrictas de IAM sobre quién puede usar las claves.

3. El **control de acceso y gestión de identidades**, es imperativo controlar quién accede a los recursos y qué acciones pueden realizar es esencial para proteger la integridad y confidencialidad de los datos.

No basta con cifrar. Hay que evitar que usuarios o sistemas no autorizados accedan a la información:

Autenticación (¿quién eres?): tokens JWT, OAuth, Cognito, API Keys.

Autorización (¿qué puedes hacer?): roles, políticas, scopes.

Auditoría (¿qué hiciste?): logs de CloudTrail, métricas de CloudWatch, detección de amenazas con GuardDuty.

a) AWS IAM:

- ✓ Define usuarios, roles y políticas que determinan el acceso a recursos.
- ✓ Implementa el principio de menor privilegio.

b) Amazon Cognito:

- ✓ Administra usuarios finales, logins federados (Google, Facebook, SAML) y emite tokens JWT.

c) API Gateway Autorizadores:

- ✓ Puede utilizar Cognito, IAM o funciones Lambda personalizadas para controlar el acceso a endpoints.

Buenas prácticas

- Configurar MFA para accesos administrativos.
- Usar roles temporales con duración limitada.
- Aplicar políticas condicionales (por IP, horario, grupo de usuarios).

4. El **monitoreo y auditoría** permiten detectar accesos no autorizados, intentos de ataque y errores de configuración que puedan poner en riesgo los datos.

- a) AWS CloudTrail: Registra todas las llamadas a la API de los servicios AWS. Fundamental para auditoría forense.
- b) Amazon CloudWatch: Provee métricas, logs y alarmas. Útil para detectar patrones inusuales de tráfico o errores.
- c) Amazon GuardDuty: Servicio de detección de amenazas basado en machine learning que analiza los logs y tráfico.

Buenas prácticas

- Habilitar CloudTrail en todas las regiones.
- Configurar alarmas para actividades sospechosas en CloudWatch.
- Integrar con AWS Config para detectar desviaciones en configuraciones de seguridad.

Si se requiere exponer un microservicio a través de API Gateway se perseguirían los siguientes objetivos:

- Solo usuarios autenticados mediante Cognito pueden acceder.
- Las peticiones deben ser cifradas con TLS 1.2.
- Las respuestas incluyen datos sensibles almacenados en una tabla DynamoDB cifrada con KMS.
- Las credenciales de acceso al backend están almacenadas en AWS Secrets Manager.
- Todos estos componentes pueden auditarse con CloudTrail, y su seguridad se refuerza con métricas y alertas en CloudWatch.

Si una empresa que tiene una app móvil que se conecta con una API para consultar el saldo de un cliente, debería contemplar que:

- a) Los datos sensibles como el nombre del usuario, su número de cuenta y el saldo deben cifrarse durante la transmisión mediante HTTPS.
- b) Los tokens de autenticación se deben almacenar de manera segura y estar protegidos contra accesos no autorizados.
- c) Se debe registrar cada transacción en un sistema de monitoreo que permita detectar intentos de acceso sospechosos.

Las acciones que quedan registradas en CloudTrail para auditoría:

Cifrado en tránsito - La app se comunica con la API usando HTTPS (TLS)

Autenticación - La API valida al usuario con un token JWT emitido por Cognito.

Cifrado en reposo - La API accede a una tabla DynamoDB donde está el saldo, cifrada con KMS.

Autorización - El acceso a esa tabla solo lo tiene una función Lambda con un rol IAM limitado.

El cifrado juega un papel crucial en la protección de datos. las buenas prácticas que ayudan a gestionar el acceso a los datos y prevenir violaciones:

- Categoriza los datos según su estado, sensibilidad, perfil de riesgo y requisitos regulatorios. Eso ayudará a elegir los métodos de protección más efectivos, asegurando la seguridad de todos los tipos de datos.
- Implementa herramientas básicas de protección. Los firewalls, métodos de autenticación fuertes y la gestión de identidades y accesos (IAM) son relativamente fáciles de implementar y son de suma importancia para la protección de datos.
- Usa control automatizado. Algunas herramientas bloquean automáticamente archivos y usuarios maliciosos y cifran datos para reducir el riesgo de perjudicarlos.
- Proporciona cifrado completo de disco. Se hace para proteger los datos cuando el dispositivo se pierde o es robado. Con el cifrado completo del disco, los hackers no podrán acceder a los datos almacenados en el portátil o el smartphone.
- Implementa soluciones de prevención de pérdida de datos (DLP). Estas soluciones monitorean los datos para detectar posibles fugas en todos los estados de datos y prevenir violaciones.
- Protege el almacenamiento en la nube. Utiliza intermediarios de seguridad de los corredores de acceso a la nube (CASB) que ayudan a mantener la información en la nube segura.
- Utiliza prácticas de gestión de dispositivos móviles (MDM) para proteger los datos corporativos. El software MDM ayuda a controlar todos los dispositivos de la empresa, incluidos no solo las computadoras sino también los dispositivos móviles.

El ámbito de la protección de datos está evolucionando constantemente para resistir las amenazas, y las tendencias actuales en cifrado de datos que se espera que crezcan en el futuro:

Cifrado de Extremo a Extremo

Asegura que los datos estén cifrados mientras se transfieren entre el remitente y el destinatario, impidiendo que los hackers accedan a los datos mientras están en tránsito. Incluso si los datos son interceptados durante la transmisión, permanecen ilegibles para las partes no autorizadas.

Encryption-as-a-Service (EaaS)

La implementación de soluciones basadas en la nube para el cifrado de datos se está volviendo común. Estas soluciones proporcionan protección de datos para todos los estados de datos y son convenientes para empresas y organizaciones gubernamentales.

Bring Your Own Encryption (BYOE)

Con los BYOE, las personas pueden gestionar sus claves de cifrado de manera independiente, asegurando un control total sobre la privacidad de sus datos. Para descifrar datos, los clientes necesitan usar su clave para restaurar el mensaje original.

Cifrado a Nivel de Campos

Este método cifra ciertos campos de una página en el lado del cliente, como detalles de pago, contraseñas e información de salud. Así, los proveedores de servidores no tienen acceso a estos datos.

Cifrado de Enlace Secuencial

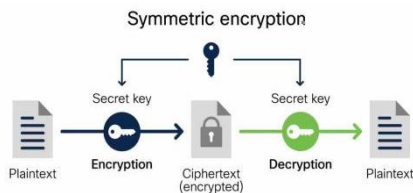
Este método cifra los datos en cada enlace de comunicación o salto en una red, proporcionando seguridad mejorada al prevenir el acceso no autorizado en cualquier punto de la transmisión de datos.

Las operaciones de cifrado pueden ser complicadas, algunas empresas dan servicio de cifrado por ejemplo para apps de pago para garantizar que las operaciones criptográficas se implementen correctamente en el código fuente, sin embargo se puede lograr un cifrado adecuado con el módulo de cifrado integrado de Node.js, para realizar correctamente las operaciones de cifrado/descifrado (simétrico) y proteger los datos de sus aplicaciones.

Cuando se habla de "cifrado", se suele hacer referencia al **cifrado simétrico**, útil para cifrar texto en una cadena aleatoria de caracteres. Un caso común donde esto es relevante es el cifrado de datos de usuario en un servidor para que se almacenen "**cifrados en reposo**" en una base de datos.

El **cifrado simétrico** consiste en tomar el texto que se desea cifrar (llamado texto plano) y usar una clave secreta con un algoritmo de cifrado para generar el texto cifrado (llamado texto cifrado). La operación es reversible, por lo que el descifrado se produce cuando podemos usar la misma clave secreta con el texto plano.

La implementación del cifrado simétrico, involucra:



<<Gráfica - Cifrado simétrico>>

Formatos de codificación: los datos se pueden codificar/decodificar de muchas maneras, estas diferentes representaciones a menudo confunden a desarrolladores cuando convierten de un formato a otro.

Algoritmos y configuración: Hay muchos algoritmos de cifrado a considerar como aes-256-gcm, aes-256-cbc, cada uno con sus propios requisitos.

Aleatoriedad: Las claves utilizadas en los procedimientos de cifrado deben generarse aleatoriamente para garantizar una alta entropía.

Entropía es la aleatoriedad que recopila una aplicación para su uso en criptografía u otros usos que requieren datos aleatorios.

Complejidad: El cifrado en Node.js implica algunos pasos que no siempre son intuitivos para los desarrolladores, y algunos conceptos pueden resultar confusos.

Forma simple de cifrado en Node.js

Para entender el cifrado, se usará cifrado con Node.js, mediante el algoritmo **aes-256-gcm**.

Para realizar el cifrado, se necesitan dos elementos:

- **plaintext:** El texto que desea cifrar.
- **key:** Una clave de cifrado de 256 bits.

Y se obtiene el resultados del cifrado:

- **ciphertext:** El texto cifrado.
- **iv:** Un vector de inicialización de 96 bits para proporcionar el estado inicial para el cifrado y permitir que el mismo key se reutilice con uno diferente para futuras operaciones de cifrado.
- **tag:** Un fragmento de datos generado durante el proceso de cifrado para ayudar a verificar que el texto cifrado no fue alterado posteriormente durante el proceso de descifrado.

```
const crypto = require ( 'crypto' );

const encryptSymmetric =( key, plaintext ) => {
  const iv = crypto.randomBytes ( 12 ) .toString ( 'base64' );
  const cipher = crypto.createCipheriv ( "aes-256-gcm", Buffer.from ( key , 'base64' ) ,
  Buffer.from ( iv , 'base64' ) ) ;
  let ciphertext = cipher.update ( plaintext , 'utf8' , 'base64' ); ciphertext += cipher.final
  ( 'base64' );
  const tag = cipher.getAuthTag ()
  return { ciphertext, tag } ;
  const plaintext = "encryptame";
  const key = crypto.randomBytes ( 32 ) .toString ( 'base64' );
  const { texto cifrado, iv, etiqueta } = encryptSymmetric (clave, texto plano);
```

En el código se definen las **variables** de entrada **plaintext** y **key** necesarias para realizar el cifrado. Además del texto que se desea cifrar, es importante generar **key** aleatoriamente para garantizar la seguridad, usando la función integrada **crypto.randomBytes()**, además convertir y codificar todo en base64formato, ya que es fácil de almacenar.

La parte de la **función de cifrado** genera un nuevo cifrado durante **crypto.createCipheriv** la inicialización con el algoritmo **aes-256-gcm**, el **key** y el **iv**.

La siguiente parte carga el texto que desea cifrar plaintext, realiza el cifrado y recupera una etiqueta de autenticación que puede usar para verificar que ciphertext no se haya manipulado durante el proceso de descifrado.

```
const encryptSymmetric = ( key, plaintext ) => {

  // crea un vector de inicialización aleatorio
  const iv = crypto. randomBytes ( 12 ). toString ( 'base64' );

  // crea un objeto de cifrado
  const cipher = crypto. createCipheriv ( "aes-256-gcm" , key, iv);

  // actualiza el objeto de cifrado con el texto plano a cifrar
  let ciphertext = cipher. update (plaintext, 'utf8' , 'base64' );

  // finaliza el proceso de cifrado
  ciphertext += cipher. final ( "base64" );

  // recupera la etiqueta de autenticación para el cifrado
  const tag = cipher. getAuthTag ();

  return { ciphertext, iv, tag };
}
```

Al cifrar datos de usuario y almacenarlos en reposo, conviene almacenar los datos cifrados **ciphertext** en **iv** la **tag** base de datos y mantener la clave secreta guardada de forma segura en otro lugar, como variable de entorno en el servidor o, en un gestor de claves.

Para recuperar los datos, se consultan en la base de datos y descifrarlos con la clave secreta, se necesitan cuatro elementos:

- **key**: La clave de cifrado de 256 bits para cifrar el texto original.
- **ciphertext**: El texto cifrado que desea descifrar.
- **iv**: El vector de inicialización de 96 bits utilizado durante el cifrado.
- **tag**: La etiqueta generada durante el cifrado.
- **plaintext**: Se obtiene el texto original encriptado.

```
const decryptSymmetric = ( key, ciphertext, iv, tag ) => {

  // crear un objeto descifrador
  const decipher = crypto. createDecipheriv (
    "aes-256-gcm" ,
```

```

    Buffer . from (key, 'base64'),
    Buffer . from (iv, 'base64')
  );

  // establecer la etiqueta de autenticación para el objeto descifrador
  decipher.setAuthTag ( Buffer . from (tag, 'base64') );

  // actualizar el objeto descifrador con el texto cifrado codificado en base64
  let plaintext = decipher.update (ciphertext, 'base64', 'utf8' );

  // finalizar el proceso de descifrado
  plaintext += decipher.final ( 'utf8' );

  return plaintext;
}

const plaintext = decryptSymmetric ( clave, texto cifrado, iv, etiqueta);

```

Aquí, la función de descifrado crea un objeto de descifrado **crypto.createDecipheriv** durante la inicialización con el algoritmo **aes-256-gcm**, **key** creado previamente y **iv** también previamente. La siguiente parte establece la etiqueta de autenticación que se usará para verificar si el objeto **ciphertext** fue manipulado y, finalmente, se realiza el descifrado para obtener el texto original.

Se pueden definir funciones auxiliares para *encryptSymmetric* y *decryptSymmetric* como las mencionadas anteriormente para mas consistencia en todo el código base.

La seguridad de datos depende de las claves utilizadas para cifrarlos, por tanto, es recomendable almacenar de forma segura las claves de cifrado y acceder a ellas en su aplicación como variables de entorno o recuperarlas de un gestor de claves en tiempo de ejecución.

Referencias.

ISO/IEC 27001:2013. (2013). Information technology — Security techniques — Information security management systems — Requirements. International Organization for Standardization.

Stallings, W., & Brown, L. (2018). Computer Security: Principles and Practice (4th ed.). Pearson.

Whitman, M. E., & Mattord, H. J. (2018). Principles of Information Security (6th ed.). Cengage Learning.

Amazon Web Services. (2023). AWS Key Management Service – Developer Guide.
<https://docs.aws.amazon.com/kms/latest/developerguide/>

Amazon Web Services. (2023). Security Best Practices for AWS.
<https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/>

Amazon Web Services. (2023). Encrypting Data at Rest and in Transit.
<https://docs.aws.amazon.com/whitepapers/latest/encryption-at-rest-and-in-transit/encryption-at-rest-and-in-transit.html>

Chapple, M., & Seidl, D. (2021). CISSP (ISC)² Certified Information Systems Security Professional Official Study Guide (9th ed.). Wiley.

Fernández-Medina, E., & Piattini, M. (2017). Seguridad en bases de datos. Madrid: Ra-Ma.

Kaufman, C., Perlman, R., & Speciner, M. (2022). Network Security: Private Communication in a Public World (3rd ed.). Prentice Hall.

Kotenko, I., & Chechulin, A. (2014). Security Metrics and Security Evaluation in Cloud Computing. In *Advances in Intelligent Systems and Computing* (Vol. 264, pp. 53–64). Springer.
https://doi.org/10.1007/978-3-319-08422-0_6

Santos, N., Gummadi, K. P., & Rodrigues, R. (2009). Towards Trusted Cloud Computing. *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud'09)*, USENIX.

OWASP Foundation. (2023). OWASP API Security Top 10.
<https://owasp.org/www-project-api-security/>

RFC Editor. (2018). RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. Internet Engineering Task Force (IETF). <https://datatracker.ietf.org/doc/html/rfc8446>
<https://nysysgroup.com/es/blog/encryption-for-the-three-states-of-data-best-practices/>
Shaw, P. (2017). Web API Security. O'Reilly Media.

Barth, A., Jackson, C., & Boneh, D. (2008). Securing Web APIs: A Best Practice Guide. IEEE Security and Privacy.
<https://www.akamai.com/newsroom/press-release/akamai-finds-api-vulnerabilities-to-be-a-high-stakes-game-for-companies-and-individuals-worldwide>

CAPÍTULO 5

Casos de éxito

Referentes en Integración de servicios

El objetivo de este capítulo es conocer implementaciones, incluyendo la planificación y optimización de servicios así como analizar los factores que contribuyeron al éxito de estas.



La integración de sistemas y la interoperabilidad de aplicaciones son esenciales para ofrecer servicios ágiles y eficientes, lo que ha llevado a un creciente interés en arquitecturas como Service-Oriented Architecture (SOA - APIs - Microservicios).

Estos enfoques permiten a las empresas desacoplar sus sistemas y facilitar la comunicación entre aplicaciones de manera más flexible escalable y eficiente.



En la era digital actual, las organizaciones enfrentan la necesidad de adaptarse rápidamente a un entorno tecnológico en constante evolución.

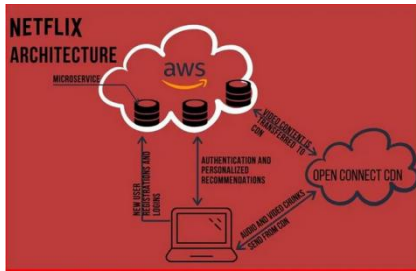
Cuando la empresa no aprovecha la nube, puede enfrentar desventajas significativas como menor accesibilidad y colaboración, rigidez para el trabajo remoto, pérdida de oportunidades de innovación y crecimiento, mayor costos de TI y mantenimiento de infraestructura, e incapacidad para escalar y adaptarse rápidamente a las necesidades del mercado, lo que la deja rezagada frente a sus competidores que sí se benefician de la flexibilidad, eficiencia y ahorro de la nube.

La nube nivela el campo de juego, permitiendo que empresas de menor tamaño accedan a tecnologías avanzadas. Las empresas que la evitan pierden esta ventaja competitiva.

Cuando se necesita; integrar sistemas heredados, buscar reutilizar funcionalidades y reducir redundancia, así como acelerar desarrollo y adaptabilidad de aplicaciones, persiguiendo objetivos de agilidad empresarial, automatización de procesos y una mayor diferenciación competitiva, es el momento de pasar a una arquitectura orientada a servicios (SOA) ya que es ideal para modernizar sistemas, romper silos departamentales y optimizar la gestión de recursos compartidos,

Este capítulo presenta casos de empresas (Netflix, Uber, Cisco) que han implementado con éxito arquitecturas SOA, proporcionando una visión de estrategias, desafíos y resultados.

A través de estos ejemplos, se podrá entender cómo las organizaciones pueden transformar su infraestructura tecnológica, mejorar la experiencia del usuario y optimizar sus operaciones, en áreas donde la adopción de arquitectura SOA ha tenido un impacto significativo en la mejora de la eficiencia operativa y en la creación de nuevas oportunidades de negocio.



Netflix como referente de migración a la nube

La arquitectura de Netflix actualmente es un referente en arquitectura de servicios por su capacidad para gestionar millones de usuarios simultáneos, funciona como un servicio de streaming por

suscripción que ofrece una amplia biblioteca de películas y series de televisión, tanto de producción propia como de contenido con licencia, ofreciendo contenido con una latencia muy baja y recuperación automáticamente de fallos lo que prueba que su arquitectura como sistema distribuido es de primera clase, y por esa razón ha establecido el punto de referencia para aplicaciones a gran escala.

La experiencia de streaming fluida de Netflix no es solo por la gran biblioteca de contenido, también es por la arquitectura de diseño de sistema, aprovechando la computación en la nube, los microservicios, el aprendizaje automático, la ingeniería del caos (Introduce fallos controlados en un sistema para comprender su impacto) y la computación en el borde (edge computing).

Netflix migró de una arquitectura monolítica a una basada en servicios, inicialmente como SOA y posteriormente consolidada como microservicios, que les permitió escalar, innovar y mantener confiabilidad global, el tránsito de Netflix a la nube **inició en agosto de 2008**, cuando se corrompió la base de datos, ocasionando la falta de atención a los suscriptores durante tres o cuatro días y evidenciando que el monolito y una base de datos única eran un punto único de falla y limita la escalabilidad del sistema.

Aunque Netflix solo tuvo una falla en la base de datos, también tuvo que reparar todos los demás servicios por lo que adoptó una estrategia de migración a la nube y migró su infraestructura de TI a la nube.

Netflix eligió Amazon Web Services (AWS) como proveedor de nube ya que le ofrecían la escala más alta y la gama más amplia de servicios y prestaciones.

En la arquitectura monolítica, todos los elementos se apilan en un contenedor compacto y sus servicios dependen entre sí, la falla de un servicio afectaba al otro, por ejemplo cuando un sitio web experimentaba un problema en su módulo de atención al cliente, el equipo debía solucionar los problemas no solo en este, sino también en otros módulos, entonces cualquier cambio afectaba a todo el sistema, dificultando el despliegue ágil de nuevas funcionalidades, frenando la innovación y

provocando interrupciones frecuentes, y no se podía gestionar un alto nivel de tráfico dificultando el mantenimiento de sitios web complejos, por lo que para actualizar un solo servicio, es necesario actualizar e implementar toda la aplicación.

Con lo anterior netflix tenía una pérdida de tiempo e incapacidad del servicio web para solucionar los problemas también resultó en pérdidas económicas, además la arquitectura tampoco era neutral en cuanto al lenguaje.

El pasar lo monolítico a cientos de servicios generó desafíos para que se descubrieran y comunicaran entre sí de forma confiable, lo cual podría resultar en errores en cascada, por lo que se tenía que pasar de los puntos de error únicos de escala vertical (por ejemplo, las bases de datos relacionales del centro de datos) a unos sistemas en la nube distribuidos, altamente fiables y escalables horizontalmente.

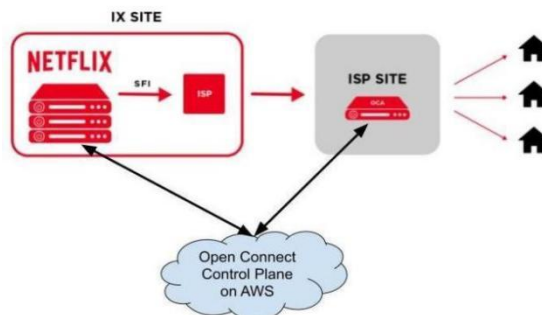
Netflix cambio la aplicación en cientos de microservicios, con responsabilidades delimitadas (perfiles de usuario, recomendaciones, reproducción, pagos...) y los desplegó en AWS, mejorando la escalabilidad, resiliencia y velocidad de innovación y para optimizar la experiencia en distintos dispositivos, Netflix creó un “mini-backend” por tipo de cliente (TV, Web, móviles) que agregaba datos de múltiples microservicios y presentaba solo lo necesario para esa interfaz.

Netflix desarrolló **Open Connect**, la red de entrega de contenido **CDN**(Content Delivery Network), desplegando dispositivos físicos en redes de ISP.

La arquitectura de Netflix funciona principalmente en, Cliente, Back-end (AWS) y Open Connect ó Netflix CDN.

Características de CDN:

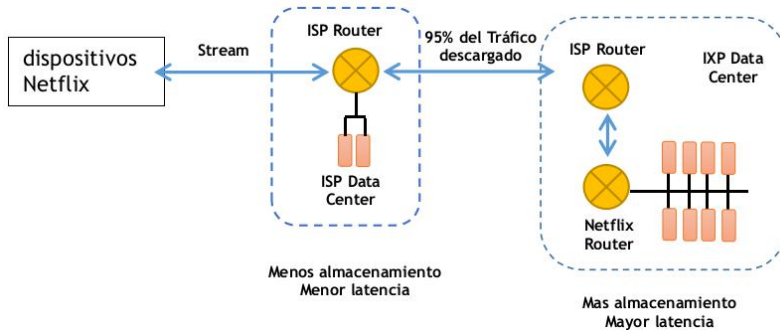
- Replicación de Contenido
- Eficiencia del Almacenamiento en Caché
- Principalmente desde la memoria
- Diversidad de Servidores



<<Gráfica - Open Connect>>

Así cuando se pulsa el botón de reproducción, el vídeo se transmite desde el servidor más cercano, lo que reduce significativamente el tiempo de respuesta (latencia y streaming)

CDN tiene servidores ubicados en centros de datos de ISP (Internet Service Provider) o centros de datos de IXP (Internet Exchange Point). ofrece más contenido a través de la ruta de red más corta.



<<Gráfica - CND (Content Delivery Network)>>

La cantidad de espacio de disco disponible por servidor es limitada y el catálogo de Netflix es muy grande, y no se puede contener en cada clúster de servidores ubicados conjuntamente, muchos clusters ubicados cerca de los usuarios finales (clusters de ISP) no tienen suficiente capacidad de disco para alojar todo el catálogo de Netflix. Por lo tanto, se almacena en caché sólo el contenido más popular en estos clusters.

En ubicaciones con un gran volumen de tráfico, se usa infraestructura por niveles:

- Servidores de alto rendimiento (hasta 100 Gbps) para contenido muy popular.
- Servidores de alta capacidad (más de 200 TB) para la parte final del catálogo.

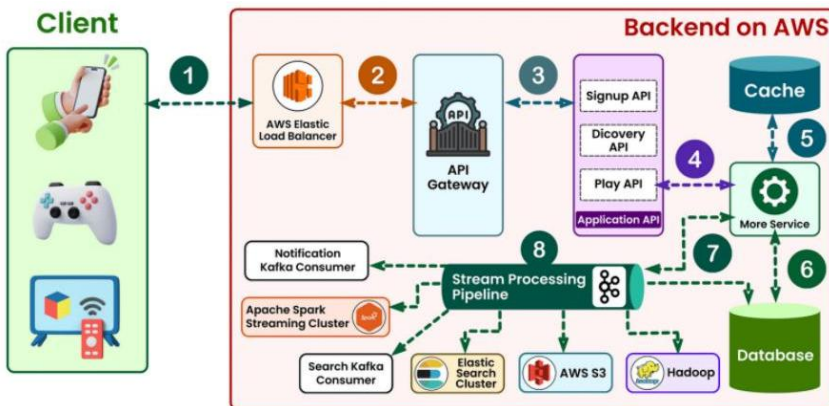
Un archivo muy popular, puede saturar los recursos de dicho servidor, si se implementa en un solo servidor, mientras que otros servidores podrían quedar subutilizados. Este efecto no es tan pronunciado en el entorno de implementación de netflix gracias a dos optimizaciones cruciales:

- Al enrutar el tráfico según la proximidad de la red, la demanda regional, incluso del contenido más popular, se comparte y difunde a través de nuestra red.
- Los archivos populares se almacenan en memoria en lugar de recuperarse constantemente del disco. Esta optimización de la memoria elimina la posibilidad de que la E/S del disco sea la causa de un cuello de botella en la capacidad del servidor.

Un equilibrio uniforme entre servidores, permite maximizar la cantidad de tráfico que atiende todo el clúster, Si falla un servidor, el tráfico dirigido a él debe distribuirse desde otros servidores del mismo clúster o desde ubicaciones más distantes de la red. Permanecer dentro del mismo clúster, minimizando así la distancia de red, es mucho más preferible, especialmente cuando se trata de contenido muy popular.

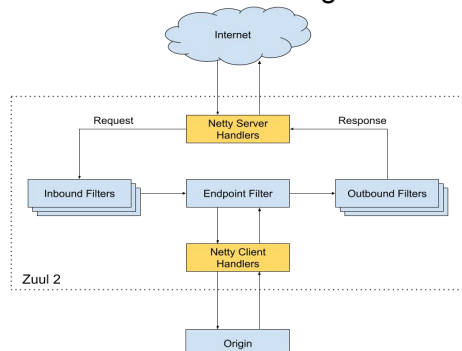
Cada título está codificado en múltiples formatos. Por ejemplo, algunos perfiles pueden usarse en dispositivos iOS y otros en ciertos tipos de Smart TV, también se tienen perfiles de video, de audio y con subtítulos, y cada perfil de audio y video se codifica con diferentes niveles de calidad, cuanto mayor sea el número de bits utilizados para codificar un segundo de contenido (bps), mayor será la calidad.

Diseño del backend de Netflix



<<Gráfica - Backend Netflix>>

1. El proceso comienza con AWS Elastic Load Balancer (ELB), que emplea una arquitectura de dos niveles para el balanceo de carga en diferentes zonas. La programación DNS Round-robin garantiza una distribución uniforme de solicitudes y una serie de instancias de balanceo de carga en el segundo nivel equilibra aún más la carga mediante el balanceo round-robin dentro de la misma zona.



2. ELB pasa el control a la API Gateway, donde Netflix utiliza ZUUL, en las instancias EC2 de AWS, ZUUL es el guardián del enrutamiento

dinámico, monitorización y seguridad, y actúa como un punto único de entrada, manejando tareas como el balanceo de carga, la seguridad, la monitorización y la resiliencia antes de redirigir las solicitudes al servicio adecuado.

Las capacidades para enrutar se basan en parámetros de consulta, URL, paths, asegurando eficiencia en manejo de solicitudes.

3. Un desafío para Netflix es transmitir video de alta calidad a los usuarios sin buffering ni retrasos, por lo que creó su propia Red de Entrega de Contenido (CDN), para no depender de terceros, Netflix instala servidores de caché en los centros de datos de los ISP, acercando el contenido a los usuarios, con eso se tiene:

- Menor latencia: El contenido se transmite desde servidores locales de ISP en lugar de centros de datos remotos en la nube.
- Reducción del uso del ancho de banda del ISP: Al almacenar en caché el contenido popular más cerca de los usuarios, Netflix reduce la congestión en las redes troncales de internet.
- Calidad de transmisión optimizada: Garantiza la entrega de contenido 4K y HDR con un buffering mínimo.

4. Netflix emplea una amplia y robusta pila tecnológica que abarca frontend, backend, bases de datos, streaming y servicios de CDN:

Tecnologías frontend:

- React.js y Node.js: la interfaz de usuario de Netflix se creó con React.js para la renderización dinámica, y Node.js admite renderización del lado del servidor.
- Redux y RxJS: para gestión de estados y el manejo de flujos de datos asíncronos.
- GraphQL y Falcor: mecanismos de obtención de datos para optimizar las respuestas de la API.

Tecnologías de backend:

- Java y Spring Boot: varios microservicios se crean con Java y Spring Boot.
- Python y Go: para diversos servicios de backend, en herramientas de aprendizaje automático y observabilidad.
- gRPC y API REST: comunicación de alto rendimiento entre microservicios.

Bases de datos y almacenamiento:

- DynamoDB y Cassandra: bases de datos NoSQL para preferencias de usuario, historial de visualizaciones y almacenamiento de metadatos.
- MySQL: se utiliza para datos transaccionales como facturación y pagos.
- S3 y EBS (Elastic Block Store): para almacenar registros, metadatos y activos.

Arquitectura basada en eventos:

- Apache Kafka y AWS Kinesis: Gestionan la transmisión de eventos, el análisis en tiempo real y el procesamiento de registros.

Infraestructura de transmisión:

- FFmpeg: Se utiliza para la codificación de video y la conversión de formatos.
- VMAF (Video Multi-Method Assessment Fusion): herramienta de evaluación de calidad de Netflix, impulsada por IA, optimiza la calidad de la transmisión.
- Protocolos DASH y HLS: Protocolos de transmisión con tasa de bits adaptativa para ajustar dinámicamente la calidad del video.

Entrega de contenido: Open Connect :

- Implementa servidores de caché dedicados en las ubicaciones de los ISP.
- Reduce la congestión de la red y mejora la calidad de la transmisión de video.
- Utiliza enrutamiento BGP para optimizar la transferencia de datos a los usuarios finales.

Seguridad y Gestión de Derechos Digitales (DRM):

- DRM Widevine, PlayReady y FairPlay: Para proteger el contenido digital de la piratería.
- Autenticación basada en tokens: Garantiza la seguridad de las llamadas API entre microservicios.
- Detección de fraude con IA: aprendizaje automático para prevenir el robo de credenciales y el abuso de cuentas compartidas.

5. Netflix garantiza una alta disponibilidad mediante la Ingeniería del Caos, una disciplina en la que se introducen fallos deliberadamente para probar la resiliencia del sistema. La herramienta **Chaos Monkey** cierra servicios aleatoriamente para verificar los mecanismos de recuperación automática.

Otras herramientas de Simian Army incluyen:

- Latency Monkey: Introduce retrasos artificiales para simular ralentizaciones de la red.
- Conformity Monkey: Detecta instancias no estándar o mal configuradas y las elimina.
- Chaos Gorilla: Simula el fallo de regiones completas de AWS para probar la resiliencia de todo el sistema.

6. El cerebro detrás de las recomendaciones de Netflix (IA)

El motor de recomendaciones de Netflix se basa en algoritmos de aprendizaje automático y aprendizaje profundo que analizan:

- Historial de visualizaciones: Lo que los usuarios han visto anteriormente.
- Interacciones de los usuarios: Comportamiento de navegación, pausas, saltos y repeticiones de visionado.

- Metadatos de contenido: Género, actores, directores, estilos cinematográficos e incluso composición de escenas.
- Filtrado colaborativo: Encuentra usuarios similares y sugiere contenido según sus preferencias.
- Algoritmos contextuales: Una forma de aprendizaje por refuerzo que ajusta las recomendaciones en tiempo real según los comentarios de los usuarios.

Netflix utiliza pruebas A/B a escala, lo que garantiza que cada cambio en la interfaz de usuario, ajuste de recomendaciones o actualización.

7. Seguimiento de millones de eventos por segundo viendo contenido simultáneamente, Netflix debe monitorear el rendimiento del sistema en tiempo real. Las herramientas clave de monitoreo incluyen:

- Atlas: la plataforma de telemetría en tiempo real, para monitorear el estado.
- Eureka: herramienta de descubrimiento de servicios para enrutar el tráfico entre microservicios.
- Hystrix: biblioteca de disyuntores para prevenir fallos en cascada.
- Spinnaker: herramienta de implementación automatizada para implementar actualizaciones de software sin problemas.
- Zipkin: herramienta de rastreo distribuido para analizar el flujo de solicitudes entre microservicios.

Esta pila de observabilidad permite a Netflix detectar anomalías de forma proactiva, lo que reduce el riesgo de degradación del rendimiento.

8. Para mantener la seguridad y privacidad Netflix implementa:

- Cifrado de extremo a extremo: Protege los datos de los usuarios y el contenido en streaming del acceso no autorizado.
- Autenticación multifactor (MFA): Previene la apropiación indebida de cuentas.
- Control de acceso y políticas basadas en roles: Restringe el acceso de los empleados a servicios sensibles.
- DRM (Gestión de derechos digitales): Previene la distribución de contenido no autorizado mediante marcas de agua y cifrado.
- Detección de bots y prevención de fraude: Identifica y bloquea los ataques de robo de credenciales y el abuso de compartir cuentas.

Las actividades del usuario y los datos históricos van por el canal de procesamiento de streaming. Transformándose en una experiencia de visualización personalizada, estos datos son la base de las recomendaciones personalizadas de Netflix, garantizando una experiencia única y atractiva para cada usuario.

Netflix aprovecha la eficacia de herramientas de procesamiento de big data como AWS, Hadoop y Cassandra. Estas herramientas analizan a fondo la vasta base de datos de los usuarios y extraen información

valiosa que contribuye a mejorar la experiencia de streaming en general y para mejorar productividad y reducir errores, migraron a **gRPC**, automatizando clientes mediante generación de código y reduciendo la mano de obra manual en definiciones de APIs.

Netflix sigue evolucionando.

Para el 2015 ya se había migrado a la nube a la mayoría de los sistemas, incluidos todos los servicios de cara al cliente. Aunque se seguía pensando en una ruta a la nube segura y duradera para la facturación y para todos los aspectos de la gestión de datos de clientes y de empleados. A principios de enero de 2016, después de siete años de esfuerzo, se terminó la migración a la nube. y se apagaron los últimos bits del único centro de datos que seguía usando el servicio de streaming.

La tecnología de streaming de Netflix ha avanzado muchísimo en los últimos años y ya no sufre las limitaciones que enfrentaba anteriormente, además de perfeccionar el uso de microservicios, Netflix publicó en código abierto varias herramientas que utilizaba para desarrollarlos. El Centro de Software de Código Abierto (OSS) de Netflix ahora es utilizado por otras empresas para crear sus sistemas de microservicios.

Habría sido complicado sostener un crecimiento tan vertiginoso con los centros de datos propietarios. La elasticidad de la nube ha permitido añadir miles de servidores virtuales y petabytes de espacio de almacenamiento en unos minutos, y por eso ha sido posible esta expansión.

El transformar a Netflix en una empresa nativa de la nube exigió mucho tiempo y esfuerzo, pero los colocó en una posición mucho mejor para seguir creciendo y convertirnos en una cadena de TV global, una opción sencilla habría sido sacar todos los sistemas, sin cambiarlos, del centro de datos y soltarlos en AWS, pero con la desventaja de que al hacer eso, se llevarían todos los problemas y limitaciones del centro de datos.

El 6 de enero de 2016, Netflix expandió su servicio a más de 130 países nuevos, convirtiéndose en una cadena verdaderamente global de TV por Internet, el poder utilizar varias regiones en la nube de AWS, repartidas por el mundo, permite cambiar de sitio dinámicamente y ampliar la capacidad de nuestra infraestructura global.

La nube también permitió aumentar radicalmente la disponibilidad del servicio, en los centros de datos se sufría de apagones, sobre todo los primeros años de la migración, pero se aumentó considerablemente la disponibilidad, es difícil evitar los fallos en los sistemas distribuidos a gran escala, ni siquiera si están basados en la nube, no obstante, la nube permite crear servicios de alta fiabilidad a partir de componentes fundamentalmente no fiables pero redundantes .

Al incorporar principios de la redundancia y degradación correcta a la arquitectura, y manteniendo la disciplina de barrenos de producción regulares con Simian Army, se puede sobrevivir a las fallas en la infraestructura de la nube y dentro de nuestros propios sistemas sin que la experiencia del suscriptor se vea afectada.

La disminución de costos no fue el motivo principal de la decisión de transitar a la nube. Pero los costos en la nube por lo menos en streaming son mucho menores que los del centro de datos, gracias a la elasticidad de la nube, que nos permite optimizar constantemente la mezcla de tipos de instancias y crecer y menguar el espacio que ocupamos casi al instante sin tener que mantener búferes de gran capacidad, además de aprovechar la economía de escala que solo pueden existir en un gran ecosistema de la nube.



Uber un modelo de nube híbrida

En 2008 Travis asistía junto a Garrett a LeWeb, evento tecnológico en París y escuchan la

idea de **reducir gastos en transporte** compartiendo vehículo con otras personas. Más tarde, ambos intentan conseguir un taxi de vuelta sin éxito, y es cuando empiezan a platicar de una idea **crear un medio de transporte que se solicite simplemente pulsando un botón**, de esta idea **nace Uber en 2009**.

En 2009 Travis Kalanick funda UberCab, con ayuda de Garrett Camp como cofundador, la empresa se centró en ofrecer vehículos premium en un solo clic, en algunas áreas metropolitanas de San Francisco.

En 2011, inicia el éxito de **UberCab**, y por problemas con sindicato de taxistas de San Francisco cambia a Uber convirtiéndose en un servicio disponible en toda la ciudad, por la facilidad de pedir un auto pulsando un botón en el teléfono, ese mismo año, Uber llega a Nueva York y se internacionaliza con el servicio en París.

En 2012, inicia **UberX**, un servicio de transporte a bajo costo que permite a las personas conducir bajo el nombre de Uber con su propio vehículo. En 2013, Uber inaugura en India y algunas ciudades de África.

En 2014, Uber inicia operaciones en China y lanzan **UberRUSH** en Manhattan, un servicio de entrega a través de ciclistas a distintos lugares de la ciudad, a mitad del año, nace **UberPool**, un servicio que permite a los usuarios compartir el viaje con otros que vayan al mismo destino o que utilizan la misma ruta.

En 2015 lanzan **UberCargo** en Hong Kong, el primer servicio de entrega instantánea, y tres meses más tarde, lanza **UberEats**, en Los Ángeles, Chicago, New York y Barcelona.

Durante el evento virtual Go/Get 2022 (evento virtual de GoToWebinar), Uber anunció algunas novedades para contrarrestar los efectos de COVID-19, destacando pedidos por voz por la integración del Asistente de Google en, UberEats, UberTravel, UberCharter y UberOne.

La principal misión de Uber es ser fiables en transporte, en todas partes y para todo el mundo, para esto se crean y trabajan los datos complejos, luego se agrupan y se ordenan la plataforma, que permite a los conductores poder conseguir que las empresas y los pilotos se desplacen.

El desafío de Ingeniería de Uber es disponibilidad y escalabilidad, ya que el negocio está basado en el mundo físico en tiempo real, y las personas confían en su tecnología para poder hacer dinero, ir a donde tienen que ir.

La interfaz de usuario de Uber debe ser simple, diseñando sistemas complejos detrás de él para mantenerse, manejando interacciones difíciles, sirviendo grandes cantidades de tráfico de datos (Principio KISS).

Principio KISS (Keep It Simple, Stupid!) establece que la mayoría de sistemas funcionan mejor si se mantienen simples que si se hacen complejos, la simplicidad debe ser mantenida como un objetivo clave del diseño, y evitar toda complejidad accidental.

La compleja ingeniería de Uber se inició a partir de una arquitectura monolítica simple, y se pensó en cómo romper este esquema de trabajo para formar una arquitectura que soporte grandes cantidades de datos y tráfico y además que provea de seguridad, usando los principios sociales y los de ingeniería de software para hacer que la plataforma tecnológica vaya creciendo de la mano de su modelo de negocios.

Uber tuvo su código base y los repositorios diseñados para trabajar en la ciudad de San Francisco, CA, y permitía resolver los problemas de negocio de manera sencilla, lo que incluía conectar pasajeros con transportistas, sistema de pagos y facturación. Para cumplir con esto se requería tener la lógica de negocios en un solo lugar, pero en la medida en que comenzaron a expandirse, esto ya no era tan razonable.

Al crecer a varias ciudades, añadir productos nuevos, y con muchos equipos de ingeniería, el monolito se volvió problemático, y los modelos de negocio comenzaron a crecer en conjunto con la adición de nuevas características, los componentes se comenzaron a tornar **altamente acoplados**, y esto logró que la encapsulación y la separación de intereses se tornara un trabajo difícil y Uber enfrentó problemas como:

- **Despliegue monolítico**, cada cambio, aunque fuera pequeño, requería desplegar todo el sistema, retrasando entregas y más riesgo.
- **Fuerte acoplamiento**, componentes con lógica fuertemente entrelazada, dificultando la separación de responsabilidades y pruebas aisladas por no modularización.
- **Escalabilidad limitada**, al escalar tráfico en una parte del sistema, se escalaba toda la aplicación.
- **Conocimiento tribal**, sólo ingenieros con mucha experiencia sabían cómo funcionan muchas partes del monolito, haciendo que los nuevos desarrolladores tuvieran curva de aprendizaje alta.
- **Problemas de seguridad, consistencia y comunicación**, al pasar a servicios distribuidos, los desafíos de latencia, fallas, seguridad de API, contratos entre servicios se hicieron críticos.

Uber adopta un enfoque de SOA más específico (**microservicios**), de dominios bien separados, cada servicio con responsabilidad clara, incluso su propia base de datos, su propio proceso de despliegue, y se toman decisiones clave:

- **Descomposición del monolito**, pasar de un código monolítico a múltiples repositorios / servicios independientes. Cada servicio dedicado a un dominio (ej. trip matching, pagos, usuarios, perfil, etc.).
- **Comunicación entre servicios**, usar RPC y REST, pero estandarizar contratos, introducir INTERFACE DEFINITION LANGUAGE (IDL), concretamente **Apache Thrift**, para permitir comunicación entre servicios escritos en diferentes lenguajes, con contratos fuertes.
- **Resiliencia y tolerancia a fallas**, crear librerías para manejar latencia, timeouts, retries, evitar que la falla de un servicio propague fallas a otros, así como asegurar que los servicios evolucionen sin romper a los consumidores.
- **Organización de equipos y despliegue continuo (CI/CD)**, separar equipos por dominios o servicios, cada equipo responsable de su servicio y poder desplegar servicios independientes, con menos riesgo de afectar otros servicios.
- **Descubrimiento de servicios, documentación y monitoreo**, encontrar cuál servicio hace qué, cómo consumirlo, entender su API, así como publicar clientes de servicio, documentación, bibliotecas/vínculos que faciliten a los desarrolladores usar los servicios.

Por lo tanto la tarea primordial del equipo de ingeniería de Uber era encontrar una forma en que rápidamente se pudieran integrar piezas de código al modelo de negocio, y se decidió seguir el ejemplo de empresas como Amazon, Twitter, Netflix, SoundCloud y romper el esquema monolítico base para formar una arquitectura orientada a servicios (SOA).

Uber decidió abordar SOA a partir del uso del patrón de microservicios, rompió la arquitectura monolítica original en muchas partes a escala con el crecimiento, en cientos de microservicios que dependen el uno del otro, dibujando un diagrama de cómo funciona Uber actualmente es complicado, y todo cambia rápidamente.

La arquitectura de microservicios pone cada funcionalidad atómica, en un servicio separado, y la forma que tiene de crecer es distribuyendo estos servicios a través de los servidores,

replicando de acuerdo a las necesidades del negocio, cada servicio puede ser escrito en su propio lenguaje y puede o no tener su propia base de datos.



Gráfica - Microservicios

Los problemas se clasifican analizando servicios, seguridad y resiliencia, con más de 500 servicios, encontrar el servicio adecuado puede ser difícil, ya identificado la forma en que se utilizará no es tan obvia, porque que cada microservicio tiene estructurado su propio camino.

Es evidente la necesidad de una forma estándar de comunicación que ofrezca la seguridad de tipos, la validación y la tolerancia a fallas. Los servicios que ofrecen REST o RPC, se caracterizan por ofrecer contratos débiles, y en el caso de Uber, los contratos varían mucho entre servicios, la adición de un esquema JSON a una API REST mejora la seguridad y proceso de desarrollo contra el servicio (API monolítica a API monolítica distribuida), pero no es trivial de escribir o mantener, esto no ofrece garantía respecto a la latencia ó tolerancia a fallas, no hay forma estándar para manejar los tiempos de espera del lado del cliente y las interrupciones, tampoco se garantiza que una interrupción de un servicio no cause interrupciones en cascada, por lo que la capacidad de recuperación global del sistema se vería afectada negativamente por estas debilidades.

En esta etapa de crecimiento de Uber se continúan evaluando tecnologías y herramientas para adaptarse a las metas, siendo ideal el uso de una interfaz IDL, que proporciona una gran cantidad de herramientas pre-construidas permitiendo que un programa u objeto escrito en un idioma se comunique con otro programa escrito en un idioma desconocido.

Los IDL (Lenguaje de definición de interfaz) generalmente se usan para describir los tipos de datos e interfaces de manera independiente del lenguaje.

Evaluaron herramientas existentes y **Apache Thrift** (usada por Facebook y Twitter) cumplió con las necesidades. Thrift es un conjunto de bibliotecas y herramientas para la construcción de servicios de lenguajes cruzados altamente escalables

Para lograrlo, se definen en un archivo los tipos de datos y las interfaces de servicios siendo flexible del lenguaje de programación, posteriormente se genera el código para abstraer el transporte y la codificación de mensajes RPC entre los servicios escritos en todos los lenguajes que usamos (Python, Node, Go, etc.)

```

1  struct Animal {
2      1: i32 id
3      2: string name
4      3: string sound
5  }
6
7  exception NotFoundException {
8      1: i32 what
9      2: string why
10 }
11
12 service Zoo {
13     /**
14      * Returns the sound the given animal makes.
15      */
16     string makeSound(1: string animalName) throws (
17         1: NotFoundException noAnimalFound
18     )
19 }

```

Gráfica - Thrift Example: SOA at Uber

Thrift garantiza la seguridad de la unión de los servicios al utilizar contratos estrictos. El contrato describe cómo interactuar con ese servicio incluyendo como llamar a los procedimientos de ese servicio, lo que proporcionan entradas y salida de lo que se espera al momento de ejecutarlo.

Un contrato significa menos tiempo en encontrar la manera de comunicarse con un servicio y hacer frente a la serialización, con la evolución de microservicios no hay preocupación acerca de las interfaces que cambian repentinamente, aumentando la capacidad de desplegar servicios de forma independiente de los consumidores.

Uber se inspiró en la tolerancia a fallas y las bibliotecas de latencia en otras empresas que se enfrentan a problemas similares, tales como Netflix y la biblioteca Hystrix y Twitter con la biblioteca Finagle , para abordar el problema de la resistencia, estas bibliotecas que aseguran los clientes son capaces de hacer frente a situaciones de falla con éxito.

Hystrix: biblioteca de latencia y tolerancia a fallas diseñada para aislar puntos de acceso a sistemas remotos, servicios y bibliotecas de terceros, dejar de fallas en cascada y permitir la resiliencia en sistemas distribuidos complejos donde la falla es inevitable.

Finagle: sistema RPC extensible para el JVM, utilizado para construir servidores de alta concurrencia, implementa API de cliente y servidor uniforme para varios protocolos, está diseñado para un alto rendimiento y concurrencia, la mayor parte del código es protocolo independiente y flexible, que simplifica la implementación de nuevos protocolos.

Uber buscó mejorar la escalabilidad organizacional y proveer mayor resistencia y tolerancia a fallas usando la arquitectura de microservicios, entendiéndolo que hasta este punto la solución no es perfecta y requiere de cambios, desafortunadamente existen pocas herramientas alternativas para Python y Node que son parte del stack de desarrollo de Uber y se requiere de buena inversión de tiempo para construirlas.

La pila tecnológica de Uber (Interacción de apps)

La plataforma de Uber, es lo que le da poder a la organización de la ingeniería de Uber, equipos de plataforma crean y mantienen las cosas que permiten a los ingenieros para posteriormente construir otros programas, funciones, y las aplicaciones que se utilizan.

Nube híbrida.

En cuanto a Infraestructura, se ejecuta en un modelo de nube híbrida, usando una mezcla de proveedores de la nube y múltiples centros de datos activos. Se asignan ciudades para el centro de datos más cercano geográficamente, pero cada ciudad se copia en un centro de datos

diferente en otra ubicación. Esto significa que los centros de datos son los viajes en todos los tiempos de funcionamiento, no se tiene noción de un centro de datos como copia de seguridad. Para la provisión de esta infraestructura, se utiliza una mezcla de herramientas internas y Terraform (herramienta de infraestructura como código) que permite construir, cambiar y ver la infraestructura de versión de manera segura y eficiente. Esto incluye componentes de bajo nivel como instancias de cómputo, almacenamiento y redes; y componentes de alto nivel como entradas DNS y características SaaS.

Las nubes híbridas combinan modelos de nubes públicas y privadas, así un usuario es propietario de unas partes y comparte otras, aunque de una manera controlada. Las nubes híbridas ofrecen un escalado a demanda, pero añaden la complejidad de determinar cómo distribuir las aplicaciones a través de estos ambientes diferentes.

La nube híbrida está reservada a aplicaciones simples, que no requieren de ninguna sincronización o necesiten bases de datos complejas, se unen mediante la tecnología y permiten enviar datos o aplicaciones entre ellas, un ejemplo son los sistemas de correo electrónico empresarial.

Almacenamiento.

El almacenamiento cambió con el crecimiento, y fue necesario aumentar el tiempo de almacenamiento en disco y la disminución de respuesta del sistema disponibles.

Inicialmente se usó un modelo sin esquema para el almacenamiento de datos a largo plazo, con MySQL, Riak y Apache Cassandra (base de datos distribuida de código abierto NoSQL) se reúnen como plataforma de alta disponibilidad, y las demandas de baja latencia.

Con el tiempo, las instancias sin esquema reemplazaron instancias individuales de MySQL y Postgres, y Cassandra reemplazó a Riak debido a la velocidad y el rendimiento.

Para el almacenamiento y análisis de datos complejos distribuidos, se utilizó un almacén Apache Hadoop (procesamiento distribuido de grandes conjuntos de datos en grupos de computadoras).

Más allá de las bases de datos mencionadas, los ingenieros de Seattle se centraron en la construcción de una plataforma de datos en tiempo real.

Se utiliza Redis tanto para el almacenamiento en caché y la puesta en cola, y Twemproxy (proxy desarrollado por Twitter) que actúa como una capa intermedia entre las aplicaciones cliente y múltiples instancias de bases de datos en memoria como Redis y Memcached, así ofrece

escalabilidad de la capa de almacenamiento en caché sin sacrificar la tasa de aciertos de caché a través de su algoritmo de hash coherente.

Redis (REmote DIctionary Server) almacén de datos en memoria de código abierto, rápido y versátil, que funciona como base de datos, caché, broker de mensajes y base de datos vectorial, gracias a su almacenamiento de datos en la memoria RAM en lugar de discos.

Rastreo de eventos.

Los servicios interactúan entre sí y con los dispositivos móviles, y esas interacciones son valiosas para usos internos como la depuración, así como los casos de negocios como precios dinámicos, para el registro, se utilizan varios clústeres Apache Kafka(plataforma de transmisión de eventos distribuida de código abierto), y los datos se archivan en Hadoop y/o en un servicio web de almacenamiento de archivos antes de que caduque Kafka. Estos datos también se ingieren en tiempo real por los diversos servicios e indexados en una pila ELK(plataforma de búsqueda Elasticsearch, Logstash, y Kibana).

- Elasticsearch es un motor de búsqueda y análisis distribuido de código abierto construido para aplicaciones de velocidad, escala y IA.
- Logstash es una tubería de procesamiento de datos del lado del servidor de código abierto que ingiere datos de una multitud de fuentes, los transforma y luego lo envía a su "alojamiento" favorito.
- Kibana es una herramienta de interfaz visual que permite explorar, visualizar y crear un panel de control sobre los datos de registro almacenados en clústeres de Elasticsearch.

Identidad en la nube para la gestión de accesos.

Utilizamos contenedores Docker (gestión de apps en contenedores) sobre Mesos (utiliza los principios del núcleo Linux, a un nivel diferente de abstracción) para hacer funcionar los microservicios con configuraciones consistentes de manera evolutiva, con la ayuda de Aurora (marco de mesos) para servicios de larga duración y trabajos cron.

Enrutamiento y el descubrimiento de servicios.

La arquitectura (SOA) de Uber hace que el descubrimiento de servicios de enrutamiento sea crucial para el éxito de Uber. Los servicios deben ser capaces de comunicarse entre sí en la compleja red, para eso se utiliza una combinación de HAProxy y Hyperbahn.

La colección de software de código abierto desarrollado en Uber (Ringpop, TChannel y Hyperbahn) comparten una misión común para agregar la automatización, la inteligencia, y el rendimiento de una red de servicios.

Los servicios legados utilizan instancias locales HAProxy para encaminar JSON sobre peticiones HTTP a otros servicios, con el servidor front-end web Nginx a los servidores en el extremo posterior, así la solución de problemas es fácil, pero se le dio prioridad a la fiabilidad a largo plazo sobre la capacidad de depuración, por lo que usando protocolos alternativos para HTTP como SPDY, HTTP/2 (facilidad de publicación de actualizaciones a los suscriptores), y TChannel junto con los lenguajes de definición de interfaz como Thrift y Protobuf, ayudan a evolucionar el sistema en velocidad y fiabilidad. Además Ringpop, una capa de hash coherente, trae la cooperación y la auto-sanación (self-healing) a nivel de aplicación e Hyperbahn permite que los servicios puedan encontrarse y comunicarse con los demás de forma sencilla y fiable, así como los servicios se programan de forma dinámica con Mesos.

Desarrollo e implementación.

Phabricator provee una gran cantidad de poder para poder manejar nuestra gran cantidad de operaciones internas, a partir de la revisión de código hasta el proceso de la documentación automatizado, a través del código en OpenGrok. Para proyectos de código abierto de Uber, se desarrolla usando GitHub.

Los ingenieros de Uber se esfuerzan por hacer que el desarrollo simule ambiente de producción lo más cerca posible, por lo que desarrollan en máquinas virtuales que se ejecutan en un proveedor de la nube o en el portátil de un desarrollador, construye su propio sistema de despliegue interno para gestionar las construcciones usando Jenkins para la integración continua. Se combinaron Packer, Vagrant, Boto, y Unison para crear herramientas para la construcción, gestión y desarrollo de las máquinas virtuales, y utiliza Clusto para la gestión de inventarios en el desarrollo. Puppet gestiona la configuración del sistema.

Se construye y mantienen canales de comunicación estables, para el descubrimiento de información, usa uBlame para realizar un seguimiento de qué equipo es propietario de un determinado servicio, y Whoer para buscar nombres, caras, información de contacto, y la estructura organizativa.

Un servicio empresarial de alerta a los ingenieros para mantener los sistemas en funcionamiento, la mayoría de desarrolladores ejecutan OSX y Linux en sus portátiles, los casos de producción se ejecutan con Debian.

Languages.

A bajo nivel, los ingenieros de Uber escriben principalmente en Python, Node.js, Go, y Java, se usa Node.js para el equipo del mercado, y Python para todos los demás, estos alimentan la mayoría de los servicios que se ejecutan en Uber, se usa Go y Java por razones de alto rendimiento,

Proporciona soporte para los lenguajes. Java aprovecha el código abierto y se integra con las tecnologías externas como Hadoop y otras herramientas de análisis, y Go da eficiencia, simplicidad y velocidad de ejecución.

Se cambió el código Python por Tornado con Python, para romper el código base original en microservicios, un modelo de programación asíncrona para mejor rendimiento, pero el soporte nativo de Go, para la concurrencia es ideal para la mayoría de los nuevos servicios de rendimiento crítico, cuando es necesario por eficiencia, velocidad a nivel del sistema, se usa C y C ++.

Caso específico (servicio facturación Uber **Invoice-gen**).

El servicio de facturación **Invoice-gen**, acumulaba deuda técnica, problemas de rendimiento y escalabilidad, y necesidad de modernización.

Problemas del servicio, lenguaje Python, legado difícil de mantener bajo demanda creciente, así como lógica complicada, flujo de generación de facturas atravesaba muchas colas Redis, muchas tablas en base de datos (12 tablas), múltiples pasos intermedios. Esto generaba cuellos de botella, facturas que no se generaban dentro del SLA, pérdidas, retrasos.

Mejora del servicio **Invoice-gen**, se crea **Invoicer** en Go, para lograr mayor desempeño, menor latencia, mejor capacidad de manejo de carga, reducción del número de tablas necesarias de 12 a 2 en algunas operaciones, simplificación del modelo de datos, unificación de modelos de datos, rediseño de APIs, de endpoints; generalización para cubrir casos futuros, y Shadowing / tráfico duplicado, el nuevo servicio comenzaba a recibir el mismo tráfico que el viejo en modo “shadow” para validar resultados, sin afectar producción.

Resultados (impactos a clientes y negocio)

- Mayor confiabilidad, menor retraso en facturas, cumplimiento de SLAs (Acuerdo de Nivel de Servicio).
- Mejor desempeño, en recursos, latencia y carga en base de datos.
- Reducción de complejidad técnica, mantenimiento más sencillo.
- Flexibilidad y soporte en facturación (países, reglas fiscales) con menor esfuerzo.

Resumen Evolución de Uber (Monolito - Microservicios - Nube)

Inicio del Monolito

- Uber aplicación simple, monolítica, para operar su servicio de viajes en una ciudad (San Francisco), las funcionalidades (matching conductor-pasajero, pagos, tarifas, etc.) estaban integradas en un solo código base / repositorio. *Kambu+1*
- El monolito cumplía inicialmente, pero creció en usuarios, regiones, productos y complejidad, problemas evidentes. Cambios pequeños requerían reinstalar / desplegar todo, coordinación pesada, mucho “conocimiento tribal” para entender efectos colaterales. *Uber+2Kambu+2*

Paso a SOA / Microservicios

- Uber migró hacia la arquitectura orientada a servicios (SOA), adoptando microservicios. Cada servicio dedicado a una parte del dominio: autenticación, matching, pagos, notificaciones, etc. *Uber+3HYS Enterprise+3Uber+3*
- Se utilizaron tecnologías para facilitar esa transición: por ejemplo Apache Thrift como interfaz de definición de servicio (IDL) y para RPC entre servicios — esto ayuda a definir contratos y permitir interoperabilidad entre servicios escritos en distintos lenguajes. *HYS Enterprise+1*
- Desarrollo de herramientas internas para resiliencia, tolerancia a fallas, latencia, monitoreo, tracing, etc. Por ejemplo Jaeger para trazabilidad distribuida, sistemas propios de logging y métricas. *HYS Enterprise+1*

Infraestructura intermedia: On-premises + Contenedores + Orquestación

- Uber tenía (y tenía históricamente) gran parte de su infraestructura en centros de datos propios (“on-premises”). *ITPro Today+1*
- Usaba herramientas de orquestación de contenedores y plataformas propias. Mesos era uno de los componentes clave para orquestar cargas, servicios stateless, etc. También desarrollaron plataformas internas para desplegar sus servicios a escala. *The Stack+2InfoQ+2*

Migración a la nube / cloud & multicloud

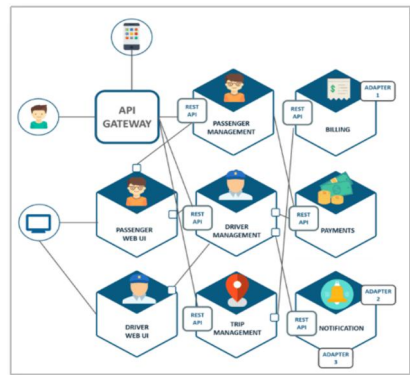
- A partir de alrededor de 2022-2023 Uber inicia una estrategia fuerte de migración hacia proveedores de nube pública, abandonando progresivamente los datacenters on-premise. Su meta es un híbrido / multicloud, diversificando riesgo. *Computer Weekly+4ITPro Today+4Oracle+4*
- En particular, usa Oracle Cloud Infrastructure (OCI) y Google Cloud Platform (GCP) como nubes públicas importantes para cargas críticas, big data, modelos de IA, etc. *Digitalisation World+3Oracle+3ITPro Today+3*

- Ha desplegado una plataforma llamada Up, que es su federación de servicios/stateless, que permite manejar contenedores, despliegues, ubicación geográfica, balance de cargas, etc., desligando a los equipos de producto de muchos detalles infraestructurales. The Stack+2Uber+2

Cambio de Mesos a Kubernetes

- Uno de los hitos recientes es que Uber migró (“stateless container orchestration platform”) desde Apache Mesos hacia Kubernetes. Uber+2The Stack+2
- Las razones incluyen que Mesos dejó de recibir suficientes mejoras/licencias (o comunidad), la interoperabilidad con ecosistemas cloud-nativos, mejores herramientas off-the-shelf, etc. Kubernetes se convirtió en estándar, con mejores ecosistemas para seguridad, observabilidad, compatibilidad con proveedores de nube, etc. Uber+1
- Esta migración es gradual, involucrando decenas de miles de servicios, reemplazando/remodelando pipelines de CI/CD, herramientas de descubrimiento de servicios, registro de contenedores, manejo de artefactos post-fallo, etc. Uber+1

Adoptar este estilo arquitectónico le brindó a Uber innumerables beneficios. En primer lugar, los desarrolladores se dividieron en equipos específicos que debían centrarse en un servicio a la vez. Esto permitió convertirse en expertos en su servicio específico. Cuando surgía un problema, podían solucionarlo más rápido sin afectar las demás áreas de servicio y la velocidad, calidad y facilidad de gestión mejoran de inmediato.



Gráfica - SOA Uber Engineering

A medida que Uber comenzó a crecer, se facilitó el escalamiento, los equipos podían centrarse únicamente en los servicios que necesitaban escalar. Con la **arquitectura**

de microservicios, todo funcionó a la perfección, y se logró una tolerancia a fallas más fiable.

Sin embargo, había un problema, la simple refactorización del monolito en microservicios no fue el final de la trayectoria de Uber. Según Susan Fowler, ingeniera de confiabilidad del sitio de Uber, la red de microservicios necesitaba una estrategia de estandarización clara, o correría el riesgo de descontrolarse .

El primer enfoque de Uber para la estandarización fue crear estándares locales para cada microservicio, funcionó bien al principio para impulsar los microservicios, pero Uber descubrió que los microservicios individuales no siempre podían confiar en la disponibilidad de otros microservicios debido a diferencias en los estándares. Si los desarrolladores cambiaban un microservicio, generalmente tenían que cambiar los demás para evitar interrupciones del servicio, interfiriendo con la escalabilidad, ya que era imposible coordinar nuevos estándares para todos los microservicios después de un cambio.

Uber y el desarrollo de estándares globales

Analizaron los principios que conducían a la disponibilidad, tolerancia a fallas, documentación, rendimiento, fiabilidad, estabilidad y escalabilidad, ya identificados, establecieron estándares cuantificables, así los desarrolladores podían analizar métricas de negocio, como las visitas a páginas web y las búsquedas.

Finalmente, convirtieron las métricas a solicitudes por segundo en un microservicio. Si bien no fue un cambio rápido, fue muy necesario. Uber parecía estar creciendo externamente, pero internamente era muy difícil mantenerlo en un estado de crecimiento sin interrupciones ni deficiencias en el servicio.

Cisco Systems y la implementación de servicios



A lo largo de su historia Cisco, ha experimentado numerosos cambios y transformaciones, especialmente en relación con la adopción de arquitecturas y tecnologías de redes. Aunque no se puede cuantificar con precisión los "cambios sobre SOA" en términos numéricos, se pueden identificar varios hitos importantes en la evolución de Cisco que reflejan su adaptación a diferentes enfoques de arquitectura de red.

Cisco se fundó en 1984 con el objetivo de conectar redes dispares a través de routers. Su enfoque inicial se centró en la conectividad LAN y WAN, utilizando protocolos como TCP/IP. Durante este periodo, Cisco desarrolló y vendió routers multiprotocolo, sentando las bases para la interconexión de redes.

La transición de Cisco **de una arquitectura monolítica a una basada en microservicios** implicó una estrategia gradual, comenzando con la identificación de componentes modulares dentro del monolito y luego transformándolos en servicios independientes, cada uno con su propia base de datos.

Dentro de la estrategia, se utilizó el patrón "Strangler Fig", *se usa en aplicaciones legadas, servicios de alta complejidad, o desarrollos poco mantenibles*, lo que permitió a Cisco migrar funcionalidades específicas a nuevos servicios mientras mantenía el monolito en funcionamiento, reduciendo gradualmente su tamaño hasta que eventualmente fue reemplazado por completo.

*El patrón **Strangler Fig** (ficus estrangulador), está inspirado en el árbol ficus, que germina cerca de otra planta, y la utiliza como protección, conforme va creciendo la estrangula, usando sus nutrientes y luz solar de forma gradual, hasta que el ficus es fuerte para reemplazarlo.*

A principio de los años 2000, Cisco Systems tenía una arquitectura de TI que limitaba su capacidad de crecimiento e innovación, y experimentó un crecimiento exponencial por el auge de las plataformas en la nube, la empresa se diversifica en áreas como conmutación LAN, voz sobre IP (VoIP), y seguridad, adoptando arquitecturas basadas en protocolos de red estándar, experimentó el auge de la burbuja puntocom, alcanzando su punto más alto al convertirse en la empresa más valiosa del mundo, superando brevemente a Microsoft.

Comenzó a integrar tecnologías como MPLS (Conmutación Multiprotocolo de Etiquetas).

- Usaban más de 150 aplicaciones internas, la mayoría desarrolladas por diferentes equipos sin una visión común.
- Los procesos de negocio estaban fragmentados, duplicados y codificados directamente en las aplicaciones.
- Cada sistema funcionaba como una isla, con poca interoperabilidad.
- Las integraciones eran manuales, con flujos rígidos, costosos y difíciles de mantener.
- La empresa necesitaba responder con agilidad a cambios del mercado, pero la arquitectura no lo permitía.

Cisco enfrentó la necesidad de adaptarse a arquitecturas más flexibles y orientadas a servicios, aunque no se puede decir que Cisco haya seguido un camino SOA puro, su evolución tecnológica refleja la adopción de principios clave de SOA, como la modularidad, la reutilización de servicios y la orquestación de procesos de negocio.

Etapas iniciales del proceso de transformación:

Análisis de procesos de negocio, identificando y abstrayendo procesos de negocio comunes.

- Se mapean procesos repetidos entre áreas: órdenes, logística, etc.
- Desacoplar funciones críticas del negocio.

Identificación de servicios potenciales construyendo servicios reutilizables.

- A través de interfaces estándar (WSDL, SOAP), Cisco creó servicios compartidos que pudieran ser combinados y reutilizados (gestión de pedidos, usuarios, facturación, etc.).

Diseño de servicios granulares

- Cada servicio debía representar una unidad atómica de lógica de negocio, desacoplada del sistema que lo originaba

Construcción de un Bus de Servicios Empresarial (ESB), permitiendo orquestar servicios, transformar mensajes y garantizar seguridad e interoperabilidad.

- Se utilizó un ESB para conectar servicios de manera centralizada, simplificando la comunicación entre ellos.
- Integrar sistemas de manera estandarizada.
- Aumentar la agilidad operativa y tecnológica.

Implementación de una estrategia de gobernanza

- Se definieron políticas claras para control de versiones, monitoreo, autenticación y autorización.
- Se crearon reglas para el uso, mantenimiento y evolución de los servicios, incluyendo control de versiones y estándares de documentación.

Cisco no solo enfrentó retos técnicos, sino también organizacionales y culturales, al cambiar de un modelo tradicional a un enfoque orientado a servicios, entre otros:

- **Cultural y organización.** Los equipos de desarrollo resistieron el cambio porque estaban acostumbrados al desarrollo monolítico.
- **Complejidad en modelado.** Traducir procesos complejos a **servicios granulares** fue un reto: ¿qué tan grande o pequeño debe ser un servicio? ¿cómo evitar duplicidades?
- **Falta de experiencia.** No había suficiente talento interno con experiencia en diseño SOA, gobernanza, ni herramientas para testing y monitoreo de servicios distribuidos.
- **Integración con sistemas legados.** Los sistemas existentes no estaban diseñados para integrarse por medio de APIs, lo que requirió adaptar o envolver funcionalidades.
- **Elevada inversión inicial.** La creación del ESB, el rediseño de servicios y la implementación de gobernanza requirió tiempo y recursos antes de ver beneficios.

Las partes más delicadas y críticas de la transición fueron:

- **Los procesos de negocio no estaban pensados en términos de servicios**, ya que tradicionalmente, los procesos están diseñados para ejecutarse de forma lineal y específica para un área, mientras, y había que entender que los servicios deben ser **reutilizables, independientes y atómicos**.
- **Había ambigüedad en los límites del servicio**, ya que al definir el "tamaño correcto" de un servicio es difícil, y si es muy grande, se vuelve un mini-monolito, y si es muy pequeño, puede generar una red de servicios difícil de mantener.
- **Faltaba documentación de procesos**, ya que muchos procesos existían solo en la práctica, sin estar debidamente documentados, obligando a hacer un trabajo de ingeniería inversa.
- **Necesidad de entender el negocio a fondo**, involucrando a los arquitectos y desarrolladores quienes tuvieron que trabajar codo a codo con el negocio, para entender las verdaderas necesidades y traducirlas a servicios funcionales y útiles.
- **Evitar duplicidad y promover la reutilización**, ya que un servicio mal definido puede generar **fragmentación**, lo que va contra la filosofía SOA, por lo que Cisco tuvo que establecer catálogos de servicios, gobernanza estricta y políticas para su publicación.

A pesar de la complejidad, Cisco logró a largo plazo:

- Un ecosistema de servicios reutilizables y escalables.
- Mejora en la **eficiencia operativa**.
- Reducción en los **tiempos de desarrollo e integración**.
- Mayor visibilidad de procesos clave del negocio.

La estrategia basada en Arquitectura Orientada a Servicios (SOA) como paradigma de integración y modernización, fue parte de un cambio mayor hacia la agilidad operativa, y su implementación incluye más estrategias para modernizar su ecosistema digital, como la virtualización y la

computación en la nube las cuales han tenido un impacto significativo en la arquitectura de redes y en la forma en que Cisco ofrece sus soluciones.

Cisco ha invertido en tecnologías de virtualización y ha desarrollado plataformas para la nube, como Cisco ACI (Application Centric Infrastructure), que permite la gestión de aplicaciones y redes de forma más dinámica y eficiente, y está integrando cada vez más la inteligencia artificial y el aprendizaje automático en sus soluciones de red. Esto permite la automatización de tareas, la optimización del rendimiento de la red y la detección proactiva de problemas.

En años recientes, Cisco ha realizado reestructuraciones para enfocarse en áreas de crecimiento como la inteligencia artificial y la seguridad.

La historia de Cisco refleja una continua adaptación a los cambios en el panorama tecnológico, desde la conectividad básica de redes hasta la adopción de SOA, la virtualización y la inteligencia artificial.

Si bien no se puede cuantificar los cambios en términos de "cambios sobre SOA" de forma numérica, la evolución de Cisco demuestra una adaptación a las tendencias arquitectónicas emergentes

Los tiempos van cambiando y a medida que las empresas adoptan el uso de aplicaciones SaaS e infraestructura como servicio (IaaS) en múltiples nubes, el departamento de TI se da cuenta de que la experiencia del usuario con las aplicaciones es deficiente, esto se debe a que las WAN, diseñadas para una era diferente, no están preparadas para la explosión sin precedentes de tráfico WAN que conlleva la adopción de la nube, y este tráfico genera complejidad en la gestión, imprevisibilidad en el rendimiento de las aplicaciones y vulnerabilidad de los datos.



Abrir aún más la empresa a Internet y a la nube expone importantes amenazas y problemas de cumplimiento normativo por lo que proteger los activos críticos de una empresa es extremadamente difícil cuando una fuerza laboral diversa, que incluye empleados, socios, contratistas, proveedores e invitados, accede a las aplicaciones.

La habilitación de banda ancha en la WAN agudiza los requisitos de seguridad, lo que genera desafíos para el departamento de TI a la hora de equilibrar la experiencia del usuario, la seguridad y la complejidad.

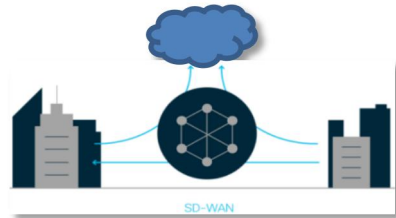
La nueva WAN

La función tradicional de una WAN (red de área amplia) consistía en conectar a los usuarios de la sucursal o campus con las aplicaciones alojadas en servidores del centro de datos. Normalmente, se utilizaban circuitos MPLS dedicados para garantizar la seguridad y una conectividad fiable. Esto no funciona en un mundo centrado en la nube.



Los nuevos modelos de negocio impulsan la necesidad de un nuevo modelo de red, **SD-WAN** (red de área amplia definida por software), aborda los desafíos actuales de TI.

Este enfoque de conectividad de red puede reducir los costos operativos y optimizar el uso de recursos en implementaciones multisitio y los administradores de red pueden usar el ancho de banda de forma más eficiente y garantizar un alto rendimiento para aplicaciones críticas sin sacrificar la seguridad ni la privacidad de los datos.



Con SD-WAN, el departamento de TI puede ofrecer enrutamiento, protección contra amenazas, descarga eficiente de circuitos costosos y simplificación de la gestión de la red WAN.

Cisco SD-WAN (red de área amplia definida por software)

Cisco ofrece su propia solución SD-WAN, llamada Cisco SD-WAN, que surgió de la adquisición de la empresa Viptela en 2017 y se renombró como tal. Cisco SD-WAN aplica los principios de las redes definidas por software (SDN) a la infraestructura de redes de área extensa (WAN) tradicional, para crear una arquitectura de superposición más flexible, eficiente y segura para las empresas.

Cómo funciona Cisco SD-WAN

- **Arquitectura de superposición:** Utiliza principios de redes definidas por software (SDN) para crear una capa virtual sobre la infraestructura de red física existente.
- **Controladores centralizados:** Permite la gestión de la red desde un punto de control centralizado, simplificando la administración.
- **Enrutamiento inteligente:** Emplea el enrutamiento inteligente para optimizar el rendimiento de la red y la conectividad para aplicaciones en la nube.

Componentes y opciones

- Cisco SD-WAN (Viptela): Es la solución empresarial de Cisco que utiliza la tecnología de Viptela y su controlador vManage para ofrecer enrutamiento avanzado.
- Cisco Meraki SD-WAN: Es otra oferta de Cisco, que se centra en una plataforma de SD-WAN y seguridad todo en uno, fácil de gestionar a través de su panel centralizado.

SD-WAN es un concepto de redes, pero Cisco ha desarrollado una marca propia para su implementación, entre los beneficios empresariales se incluyen los siguientes:

- Mejor experiencia de aplicación
- Más seguridad
- Conectividad en la nube optimizada
- Gestión simplificada

Referencias

Yury Izrailevsky, Stevan Vlaovic y Ruslan Meshenberg

https://about.netflix.com/es_es/news/completing-the-netflix-cloud-migration

Netflix's architecture migration from monolithic to microservices and resiliency tools (Eureka, Hystrix, Spinnaker) y mejoras en escala y despliegue (2008–2016)
MediumResearchGatehackernoon.

Implementación de patrón BFF, uso de Zuul y evolución técnica hacia gRPC
Reddit+1TechAheadCNCF.

Observabilidad (Atlas, tracing), benchmarking (NDBench), Data Gateway, prácticas DevOps y chaos engineering (Chaos Monkey).
arXiv+1MediumReddit.

Comprensión de la arquitectura de microservicios en Netflix.

https://www.sayonetech-com.translate.goog/blog/microservices-netflix/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc&_x_tr_hist=true

Overcoming Challenges in Microservice Architectures.

https://www.researchgate.net/publication/386219405_Overcoming_Challenges_in_Microservice_Architectures

transitioning to microservices architecture.

https://www.meegle.com/en_us/topics/software-lifecycle/transitioning-to-microservices-architecture?utm_source=chatgpt.com

Evolution of Java Usage at Netflix.

<https://blog.bytebytego.com/p/evolution-of-java-usage-at-netflix>

Let's Decode Netflix System Design and Backend Architecture.

https://www.techaheadcorp.com/blog/netflix-system-design/?utm_source=chatgpt.com

Erl, T. (2005). Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall.

Fowler, M. (2018). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.

Zimmermann, A., & Stojanovic, J. (2013). Web APIs: A comprehensive overview. Springer.

Arsanjani, A., Holley, K., & Allam, A. (2008). THE CISCO SERVICE-ORIENTED ARCHITECTURE (SOA) JOURNEY: A CASE STUDY. IBM Systems Journal, 47(3), 513–529.
<https://doi.org/10.1147/sj.473.0513>

Patrón Strangler Fig para cargas modernas

<https://aws.amazon.com/es/blogs/aws-spanish/patron-strangler-fig-para-cargas-modernas/>

Uber Engineering. Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow:
Uber Blog.

Uber Engineering. The Perils of Migrating a Large-Scale Service at Uber (Invoice-gen → Invoicer):
Uber Blog.

HYS Enterprise. Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience.

Microservices:

<http://martinfowler.com/articles/microservices.html>

Thrift Example: SOA at Uber Engineering:

<https://gist.github.com/myhrvold/1edb63caf68ffe2c2484#file-animaexample-thrift>

High scalability:
<http://highscalability.com/>

The Uber Engineering Tech Stack, Part I: The Foundation:
<https://www.uber.com/en-MX/blog/tech-stack-part-one-foundation/>

How microservices patterns made Uber's architecture perform better:
https://www.theserverside-com.translate.goog/feature/How-microservices-patterns-helped-Uber-systems-perform-better?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

The Stack cloude uber:
https://www.thestack.technology/uber-migrates-to-new-multi-cloud-platform/?utm_source=chatgpt.com

Uber Ditches On-Prem and Hooks Future to GCP and Oracle Cloud:
https://www.itprotoday.com/oracle-cloud-infrastructure/uber-ditches-on-prem-and-hooks-future-to-gcp-and-oracle-cloud?utm_source=chatgpt.com

¿Qué es SD-WAN?

https://www.cisco-com.translate.goog/c/en_ca/solutions/enterprise-networks/sd-wan/what-is-sd-wan.html?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=imgs

Acerca del Autor

Juan Ibañez es profesor universitario. Nació en la ciudad de Durango, México en 1965, obtuvo la licenciatura en Informática y la maestría en enseñanza de las ciencias.

Como profesional gran parte del tiempo lo ha invertido en la enseñanza, y en administración pública (Tecnológico Nacional de México), además de haber incursionado en tecnología multimedia, internet, software (arquitectura de soluciones) y hardware (gatgets).

Al plasmar en este libro parte de los conocimientos que aporta en docencia, pretende interesar a los lectores en las tecnologías actuales, independientemente las que en este libro se comenten.

En lo personal disfruta música relajante, degustar un buen mezcal con los amigos, así como los viajes en familia.

Contacto: e-mail pacois20@gmail.com

Agradecimientos

*Dedico este libro a quienes han estado a mi lado siempre,
mi familia (Olga, Yida, Jessi y Alison),
y a los amigos infalibles.*

*En especial, con profundo agradecimiento a mi
madre Ma Dina Salas Rivas.*

