INSTITUTO POLITECNICO NACIONAL

ESIME

UNIDAD PROFESIONAL “ADOLFO LOPEZ MATEOS”

ZACATENCO

SISTEMA DE ADMINISTRACIÓN DE PERSONAL ADMINISTRATIVO Y ESTUDIANTIL DENTRO DE UNA INSTITUCION ACADÉMICA DE MANERA REMOTA

REPORTE TÉCNICO

PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMUNICACIONES Y ELECTRÓNICA

PRESENTA:

CÉSAR RODRÍGUEZ CALDERÓN

ASESORES:

ING. ARTURO ALONSO HIT ESPINOSA

ING. GUILLERMO MANUEL ESCÁRCEGA CARRERA

CIUDAD DE MEXICO, ENERO 2021

Autorización de uso de obra

Instituto Politécnico Nacional

Presente

Bajo Protesta de decir la verdad el que suscribe **César Rodríguez Calderón**, manifiesto ser el autor y titular de los derechos morales y patrimoniales de la obra titulada “**Sistema de administración de personal administrativo y estudiantil dentro de una institución de manera remota**”, en adelante **“La Tesis”** y de la cual se adjunta copia, por lo que por medio del presente y con fundamento en el artículo 27 fracción II, inciso b) de la Ley Federal del Derecho de Autor otorga a el **Instituto Politécnico Nacional**, en adelante **El IPN,** autorización no exclusiva para comunicar y exhibir públicamente total o parcialmente en medios digitales o impreso para consulta “**La Tesis**” por un periodo de **1 año** contando a partir de la fecha de la presente autorización dicho periodo se renovara automáticamente en caso de no dar aviso expreso a “**El IPN**” de su terminación.

En virtud de lo anterior “**El IPN**” deberá reconocer en todo momento nuestra calidad de autor de “**La Tesis**”:

Adicionalmente y en mi calidad de autor y titular de los derechos morales y patrimoniales de “**La Tesis**”, manifestó que la misma es original y que la presente autorización no contraviene ninguna otorgada por el suscrito respecto de “**La Tesis**”, por lo que deslindamos de toda responsabilidad a “**El IPN**” en caso de que el contenido de “**La Tesis**” o la autorización concedida afecte o viole derechos autorales, industriales, secretos industriales, convenios o contratos de confidencialidad o en general cualquier derecho de propiedad intelectual de terceros y asumimos las consecuencias legales y económicas de cualquier demanda o reclamación que puedan derivarse del caso.

México, Ciudad de México., a de Enero de 2021.

Atentamente

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

César Rodríguez Calderón

ÍNDICE

OBJETIVO GENERAL

JUSTIFICACIÓN

**CAPÍTULO 1 – Marco Conceptual**

Introducción

Middleware como herramienta para la comunicación entre servicios web

Servicios REST

Definición de una Interfaz de Aplicación Programable (API)

**¿qué es**[API-Led Connectivity](https://www.mulesoft.com/lp/whitepaper/api/api-led-connectivity)**?**

Descripción de la plataforma Anypoint Studio

Conclusión

Bases de datos

Bases de datos operacionales

Bases de datos analíticas

Bases de datos relacionales

Componentes de una base de datos

Tipos de relaciones en una base de datos

Infraestructura de portal web

HTML

NodeJs

Desarrollo del Servidor con express

**CAPÍTULO 2 – Análisis**

Desventajas de los sistemas actuales de servicios remotos

Propuesta de solución

Implementación de un servicio por medio de una API

Capas lógicas de la conectividad API-Led

Conclusión

**CAPÍTULO 3 – Diseño**

Arquitectura

Operatividad para cada agente

Propuesta de portal web

Herramientas de software

JavaScript, NodeJs, SQL, HTML, CSS, XML, Postman

Plataforma de desarrollo de API propuesta

Recursos REST propuestos y funcionalidad

Anypoint Studio (IDE de desarrollo de API)

Microsoft SQL Management

Estructuras de datos

Introducción

Diccionario de datos

Diseño de la base de datos

**Seguridad**

Autenticación

Autorización

**CAPÍTULO 4 – Implementación**

Construcción de API

Especificación RAML

Desarrollo de flujos

Enlace de base de datos

Desarrollo de Base de datos

Desarrollo de portal web

CONCLUSIONES

GLOSARIO

BIBLIOGRAFIA

**OBJETIVO**

Desarrollar un sistema de control escolar en línea para cualquier tipo de institución académica pública o privada para ayudarlos al alta de profesores y/o alumnos, actualización de contactos, asignación de profesores a los diferentes cursos, obtener lista de cursos y su disponibilidad, dar capacidad a un alumno de inscribirse a un curso, promoviendo la administración de tal institución de una manera remota.

**JUSTIFICACION**

Cada vez es más importante para las organizaciones operar en un entorno de plataforma virtual, esto les permite aportar mayor eficiencia en sus servicios. En esta época muchas instituciones están emigrando la gestión de sus procesos en este sentido no solo por brindar mayor eficiencia en sus operaciones si no por cuestiones aún más trascendentes como la pandemia COVID-19 en la que actualmente vivimos y en dónde se busca disminuir al máximo el tiempo que una persona está en el espacio público, mientras se encuentran mejores soluciones.

Por otra parte este desarrollo puede ser utilizado como base práctica y metodológica por cualquier organización que busque la virtualización de sus procesos administrativos ya que podría ocupar el desarrollo práctico y estructural para adaptarlo a sus necesidades más específicas.

**INTRODUCCIÓN**

--------- Introducción o resumen

En este trabajo se ofrece la descripción de cada uno de los elementos que formarán la arquitectura, los cuales son el desarrollo de un portal web usando HTML y NodeJs, un API Gateway en donde se implementara el esquema OAuth 2.0 con un grant type de código de autorización para reducir el riesgo de vulnerabilidades en el proceso de autenticación ya que se consumirá una API en dónde se llegan a exponer datos confidenciales y permite a los usuarios realizar acciones de información sensible, el uso del bus de datos MuleSoft para definir los recursos y métodos necesarios, también ayudará a garantizar que la información sea correcta, antes de llevar cabo cualquier acción u operación en la base de datos, dentro de este servicio se encuentra la validación de estructura de mensajes y/o cuerpos por medio de especificaiones RAML (Restful API Modeling Language), finalmente este bus de datos estará enlazado a una base de datos MySQL para el almacenamiento del sistema.

**Capítulo 1: Marco Teórico**

**MIDDLEWARE**

Un middleware es un punto donde se centraliza todas las integraciones evitando que un sistema se conecte directamente con otro sin pasar por este.

Una plataforma de integración como servicio iPaaS (integration Platform-as-a-service), este middleware se encontrará alojado en la nube sin la necesidad de la gestión asociada de hardware y costos de infraestructura y además, puede ser accesible desde cualquier lugar.

**SERVICIOS REST**

La transferencia de estado representacional **REST** (Respresentational State Transfer) es una interfaz para conectar varios sistemas basados en el protocolo **HTTP** ( Hyper Text Transfer Protocol) y nos sirve para obtener y generar datos en formatos muy específicos, como en notación de objetos de JavaScript **JSON** (Java Script Object Notation).

Las principales ventajas de esta interfaz por las cuales se esta empleando en los últimos años son las siguientes:

* Crea una petición HTTP que contiene toda la información necesaria, es decir, un REQUEST a un servidor tiene toda la información necesaria y solo espera una RESPONSE, ósea una respuesta en concreto.
* Se apoya sobre un protocolo que es el que se utiliza para las páginas web, que es HTTP, es un protocolo que existe hace muchos años y que ya está consolidado, no se tiene que inventar ni realizar cosas nuevas.
* Se apoya en los métodos básicos de HTTP, como son:

**Post**: Para crear recursos nuevos.

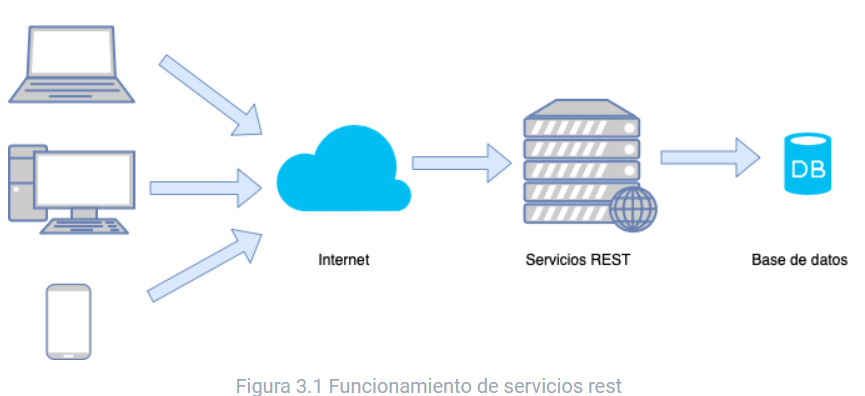
**Get**: Para obtener un lichado o un recurso en concreto.

**Put**: Para modificar.

**Patch**: Para modificar un recurso que no es un recurso de un dato, por ejemplo.

**Delete**: Para borrar un recurso, un dato por ejemplo de nuestra base de datos. (https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/)

* Nos permite separar el cliente del servidor. Esto quiere decir que el servidor se puede desarrollar en Node y Express, y la API REST con MuleSoft por ejemplo, no tienen por qué estar todos dentro de un mismo.



Los servicios REST te permiten acceder y/o modificar la información mediante los métodos HTTP, por lo cual puedes acceder a ellos mediante URL's. Por lo general regresan la información en formato JSON, aunque también pueden regresar archivos XML o csv. Debido a lo sencillo de desarrollar y consumir actualmente muy utilizados. (<https://abi.gitbook.io/net-core/3.-servicios-rest/3.1-servicios-rest>)

Los servicios más comunes son los siguientes:

|  |  |  |
| --- | --- | --- |
| Verbo | Servicio REST | Descripción |
| GET | http://localhost:5000/api/Clientes | Obtiene todos los clientes |
| GET | http://localhost:5000/api/Cliente/1 | Obtiene los datos del cliente 1 |
| POST | http://localhost:5000/api/Cliente | Agrega un nuevo cliente |
| PUT | http://localhost:5000/api/Cliente/1 | Modifica todos los datos cliente 1 |
| PATCH | http://localhost:5000/api/Cliente/1 | Modifica algunos datos del cliente |
| DELETE | http://localhost:5000/api/Cliente/1 | Borra el cliente 1 |

**CÓDIGOS DE ESTADO**

Los servicios REST manejan Códigos de Estatus para indicar si la acción se realizó correctamente empiezan con 200 de lo contrario es un error. Si empiezan con 400 por lo general quien esta consumiendo el servicio esta enviando mal la información y si empieza con 500 el error es de parte del servidor o de quien crea el servicio. https://abi.gitbook.io/net-core/3.-servicios-rest/3.1-servicios-rest

|  |  |  |
| --- | --- | --- |
| **Acción** | **Código de estatus** | **Descripción** |
| GET | 200 | OK |
| POST | 201 | Creado |
| PUT | 204 | Sin Contenido |
| DELETE | 204 | Sin Contenido |

Los servicios REST manejan códigos de estatus, para indicar si la acción se realizó correctamente empiezan con 200 de lo contrario es un error. Si empiezan con 400 por lo general quien está consumiendo el servicio está enviando mal la información y si empieza con 500 el error es de parte del servidor o de quien crea el servicio.

**JSON**

Es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto de lenguaje de programación JavaScript. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

Resumiendo lo anterior un servicio REST se compone principalmente del localizador de recursos uniforme **URL** (Uniform Resource Locator) un verbo HTTP, en algunos casos también se considera un cuerpo para la respuesta o petición. En la figura (ñ) se ejemplifica lo anterior:



**CONECTIVIDAD API-LED**

Es un principio de integración, de hecho, totalmente contrario a la integración punto a punto (point-to-point) y que aboga por una integración basada en interfaz de programación de aplicaciones API(Application Programing Interface) reusables proporcionando un bajo acoplamiento con la consiguiente mejora en la escalabilidad en nuestra application network, es decir, nuestra red de microservicios.

**API**

Una API nos permite generar comunicación y lógica entre servicios, sin necesidad de saber cómo están implementados cada uno de ellos.

Una API se considera como un contrato, con documentación que representa un acuerdo entre las partes: si una de las partes envía una solicitud remota con cierta estructura en particular, esa misma estructura determinará cómo responderá el software de la otra parte.

También nos permite habilitar el acceso a sus recursos y al mismo tiempo a mantener la seguridad y el control.

Este proyecto está desarrollado con una API para una sección de los múltiples procesos que llegue a tener una institución académica, el alcance se revisará más adelante.

El API permitirá que un cliente web acceda a información sobre cursos, profesores, y/o alumnos.

Existen tres enfoques respecto a las políticas de una API:

**Privado**

La APIs solo se puede consumir internamente, así que las organizaciones tienen un mayor control sobre ellas. Esto le da a las organizaciones mayor control sobre sus APIs.

**Partners**

Las API se comparten con otras organizaciones específicas, lo cual puede ofrecer flujos de ingresos adicionales e implementaciones bilaterales, sin comprometer la calidad. Cómo habilitar el acceso y a quiénes dependerá del desarrollador de la API.

**Público**

En este caso todos tienen acceso a las API, así que otras organizaciones pueden desarrollar API que interactúen con la de usted y así convertirse en una fuente de innovaciones. Esto permite que terceros desarrollen aplicaciones que interactúan con su API, y puede ser un recurso para innovar.

El alcance en políticas de este proyecto está previsto para ser privado de una organización académica con sus desarrolladores para exponer los recursos en una aplicación o portal web, o también tener alguna organización gubernamental cómo Partner para que por ejemplo se integre con alguno de sus servicios y que el CURP (Clave Única de Registro de Población) que provee un estudiante sea validado en las bases de datos del Registro Nacional de Población antes de darlo de alta en el sistema de la institución.

Estas características permiten un uso dinámico de esta API y que consumiendo la información y lógica que accesa se pueda integrar con nuevas APIs que implementan otras funcionalidades de la institución de una manera más eficiente.

**API Designer**

Es una herramienta que permite el diseño, la documentación, el testing de especificación de interface, así como su compartición por medio de una interfaz web. La nomenclatura que se utiliza en MuleSoft para la definición de una API es el lenguaje de modelado de APIs REST **RAML** (REST API Modeling Language) que permitirá la definición de métodos, recursos y parámetros, así como otros componentes HTTP. También permite el modelado de los cuerpos de entrada y salida de una API para generar un contrato y validación de información antes de que esta entre al servicio.

### 3. Anypoint Studio

Es el lugar donde implementaremos la lógica de las API que previamente hemos diseñado, para ello, MuleSoft nos provee de Anypoint Studio, un ambiente de desarrollo integrado **IDE (Integrated Development Enviroment)** más que similar a eclipse en el que por medio de una interfaz gráfica iremos arrastrando los componentes necesarios para implementar la lógica necesaria.

### 4. Anypoint Management Center

Es la herramienta utilizada para el control y monitorización de todas aquellas aplicaciones que hemos implementado en Anypoint Studio. Dentro del conjunto de servicios que ofrece esta herramienta, podemos destacar los dos más importantes.

**Runtime Manager:** para gestionar el despliegue de la aplicación y nos podemos decantar por dos metodologías de despliegue.

* **Despliegue On Premise:** La aplicación sería desplegada en los servidores del cliente.
* **Despliegue en Cloudhub:** La aplicación se desplegará en los workers de la nube de MuleSoft aprovisionada por AWS.

Y, ¿qué es un worker en MuleSoft? **Un Cloudhub Worker es el computador virtual donde se ejecuta la aplicación** que hemos desplegado y en un mismo worker únicamente puede desplegarse una aplicación mule.

**API Manager:** es una herramienta que permite realizar toda la gestión de las APIs publicadas en el exchange, entre las diferentes funcionalidades que ofrece, destacamos:

* Configuración de un proxy para nuestra API
* Configuración de SLAs
* Configuración de políticas de autenticación y seguridad.
* Configuración de alertas basadas en SLAs o políticas definidas
* Revisión de informes de actividad de uso de la API

### Conclusión

MuleSoft es la herramienta perfecta para la orquestación de las comunicaciones entre múltiples sistemas, permite disponer de un **punto común donde gestionar todas las integraciones** de nuestro ecosistema de aplicaciones sin necesidad de tener conocimientos técnicos avanzados, abstrayéndose en cierta medida, de la tecnología utilizada.

**BASES DE DATOS**

Una base de datos es una colección organizada de información usada para modelar algún tipo de proceso organizacional. No importa que se use papel o un programa computacional para recolectar datos. Se considera una base de datos siempre que se haga colección y guardado de información organizadamente para propósitos específicos.

**BASES DE DATOS OPERACIONALES**

Las bases de datos operacionales son la base de muchas organizaciones e instituciones en la actualidad. Los datos guardados son dinámicos, lo que significa que están en constante cambio y además actualizados en cada momento. Organizaciones como tiendas, empresas de manufactura, hospitales o clínicas usan bases de datos operacionales porque sus datos están en constante cambio.

**BASES DE DATOS ANALÍTICAS**

En contraste, una base de datos analítica guarda información histórica. Es muy valiosa para determinar tendencias, visualización de estadísticas a través de largos periodos de tiempo, o para hacer proyecciones de negocio. Los datos que se guardan son estáticos, lo que significa que esta información prácticamente nunca se modifica. Laboratorios químicos, empresas geológicas, y firmas de análisis de marketing son ejemplos de organizaciones que usan bases de datos analíticas.

Nótese que la información encontrada en una base de datos analítica es usualmente recogida de una base de datos operacional. Por ejemplo, el historial mensual de ventas debe ser resumido y guardado en una base de datos analítica.

**BASES DE DATOS RELACIONALES**

Un sistema de manejo de bases de datos relacionales (RDBMS) es una aplicación software que se usa para crear, mantener, modificar, y manipular una base de datos relacional. También proveen herramientas que se ocupan para crear aplicaciones que interactúan con los datos guardados en la base de datos.

De acuerdo al modelo relacional, los datos en una base de datos relacional son almacenados con relaciones, que se perciben por el usuario como tablas. Cada relación está compuesta de duplas (registros o renglones) y atributos (campos o columnas).

**COMPONENTES BASICOS DE UNA BASE DE DATOS**

**TABLAS**

Las tablas son la estructura principal en una base de datos. Cada tabla siempre representa un tema u objeto en específico. Cada tabla contiene al menos una columna (conocida como *llave primaria*) que identifica cada uno de sus renglones.

El tema que se le da a una tabla puede ser desde un objeto hasta un evento. Cuando el tema es un objeto, la tabla representa algo tangible, como una persona, un lugar, o una cosa. Independientemente del tipo cada objeto tiene características que pueden ser almacenadas como datos. Productos, máquinas, edificios, estudiantes o herramientas son ejemplos de objetos que pueden ser representados en una tabla.

Cuando el tema de una tabla es un evento, la tabla representa algo que ocurre en un determinado punto en el tiempo y tienes características que se desean guardar. Estas características pueden ser guardadas y después ser procesadas de la misma manera que una tabla que representa un objeto en específico. Ejemplos de eventos que se pueden representar en una base de datos son: pruebas de laboratorio, encuestas geológicas, distribución de fondos, visitas de pacientes en un hospital etc.

**COLUMNAS**

Las columnas son la estructura de datos más pequeña en una base de datos, y representa una característica del tema de la tabla a la cual pertenece. Las columnas son las estructuras que guardan datos.

Cada columna en una base de datos correctamente diseñada contiene un y único valor, y su nombre identifica el tipo de valor que contiene. Esto hace que la inserción de datos sea muy intuitiva. Si se pueden observar columnas con nombres como: Nombre, Apellido, Ciudad, Estado se encuentra exactamente qué tipo de valor va en cada columna. También será fácil encontrar datos por Estado o filtrar a las personas que se apelliden Rodríguez.

**RENGLONES**

Los renglones representan una única instancia del tema de una tabla. Están compuestos por un conjunto de columnas. Por la manera en que la base de datos sea definida, cada renglón es identificado por un único valor en la columna de *llave primaria*.

**LLAVES**

Las llaves son columnas especiales que juegan un rol importante en una tabla. Aunque una tabla puede contener muchos tipos de llaves, solo se hablará de las dos más importantes:

* Llave primaria
* Llave foránea

Las llaves primarias consisten en una columna que identifican cada renglón de manera única en cada tabla. La llave primaria es la más importante por dos razones:

* Su valor identifica un renglón dentro de toda la base de datos
* Las llaves primarias refuerzan la integridad a nivel de tablas y ayudan a establecer relaciones con otras tablas. Cada tabla en la base de datos debe tener una llave primaria.

**LLAVE PRIMARIA**

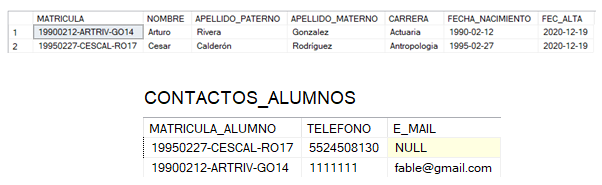
**Alumnos**

La matrícula de un alumno es un buen ejemplo de llave primaria por que identifica a cada alumno dentro de la tabla Alumnos y asegura la integridad evitando que se dupliquen los renglones o en este caso estudiantes.



**LLAVE FORANEA**

Cuando se determina que un par de tablas tienen una relación, típicamente se establece esta relación tomando una copia de la llave primaria de la primer tabla e insertándola en la segunda tabla, en dónde se convierte en la llave foránea. El termino foránea se deriva del hecho de que la segunda tabla ya tiene una llave primaria *per se*, y que la llave primaria que se introduce de la primer tabla es foránea en la segunda tabla.



**RELACIONES**

Cuando los renglones en una tabla pueden ser asociados de alguna manera con renglones de otra tabla, se dices que estas tablas mantienen una relación. Existen tres tipos de relaciones: uno a uno (one-to-one), uno a muchos (one-to-many), y muchos a muchos (many-to-many).

**ONE-TO-ONE**

Un par de tablas tienen una relación one-to-one cuando un renglón de la primer tabla está relacionado a un único renglón en la segunda, y cuando el renglón en la segunda tabla está relacionado a un único renglón en la primer tabla. La relación se establece tomando la llave primaria de la primera tabla e insertándola en la segunda tabla, en dónde como se explicó previamente se convierte en una llave foránea.

**ONE-TO-MANY**

Cuando un par de tablas tienen una relación one-to-many, un simple renglón en la primera tabla está relacionado con muchos renglones en la segunda tabla, pero un simple renglón en la segunda tabla estará relacionado únicamente a un renglón de la primera tabla. Esta relación se logra tomando la llave primaria de la primera tabla e insertándola en la segunda convirtiéndose en llave foránea, pero agregando más de un renglón en la segunda tabla que estarán relacionados a únicamente un renglón de la primera tabla.

**FRONT-END**

En este proyecto se hace una representación interactiva de los recursos que nuestra API está exponiendo de una manera entendible para un usuario final (alumno, profesor, personal administrativo) con esta finalidad se emplea un desarrollo de vista front-end con HTML y un entorno en tiempo de ejecución multiplataforma NodeJs para crear el portal web.

**HERRAMIENTAS DE DISEÑO WEB**

**HTML**

El lenguaje de marcado de híper texto **HTML** (Hypertext Markup Language) hace referencia a un lenguaje de marcado para la elaboración de páginas web, permite definir estructuras básicas para la visualización de contenido en una página web, como texto, imágenes, videos entre otros.

Para añadir un elemento externo a la página (imagen, vídeo, [script](https://es.wikipedia.org/wiki/Script), entre otros.), este no se incrusta directamente en el código de la página, sino que se hace una referencia a la ubicación de dicho elemento mediante texto. De este modo, la página web contiene solamente texto mientras que recae en el navegador web (interpretador del código) la tarea de unir todos los elementos y visualizar la página final.

**NODEJS**

NodeJs es un entorno de ejecución de código abierto y a su vez herramienta para desarrollar back-end que se ejecuta por medio de código JavaScript, otro lenguaje de programación que después de Python es el más usado en la actualidad.

Tanto JavaScript como **NodeJs** se ejecutan en el motor de tiempo de ejecución JavaScript V8 (V8 es el nombre del motor de JavaScript que alimenta Google Chrome. Es lo que toma el código JavaScript y lo ejecuta mientras navega con Chrome). Este motor toma el código JavaScript y lo convierte en un **código de máquina**más rápido. El código de máquina es un código de nivel más bajo que la computadora puede ejecutar sin necesidad de interpretarlo primero, ignorando la compilación y por lo tanto aumentando su velocidad.

https://openwebinars.net/blog/que-es-nodejs/

**Para qué nos sirve NodeJs?**

**NodeJs** utiliza un**modelo de entrada y salida sin bloqueo**controlado por eventos que lo hace ligero y eficiente (con entrada nos referimos a solicitudes y con salida a respuestas). Puede referirse a cualquier operación, desde leer o escribir archivos de cualquier tipo hasta hacer una solicitud HTTP.

La idea principal de Node.js es usar el modelo de entrada y salida sin bloqueo y controlado por eventos para seguir siendo liviano y eficiente frente a las aplicaciones en tiempo real de uso de datos que se ejecutan en los dispositivos. Es una plataforma que no dominará el mundo del desarrollo web pero si satisface las necesidades de una gran mayoría de programadores.

La finalidad de NodeJs no tiene su objetivo en operaciones intensivas del procesador, de hecho, usarlo para programación de más peso eliminará casi todas sus ventajas. Donde NodeJs realmente brilla es en la **creación de aplicaciones de red rápidas,** ya que es capaz de manejar una gran cantidad de conexiones simultáneas con un alto nivel de rendimiento, lo que equivale a una **alta escalabilidad.**

https://openwebinars.net/blog/que-es-nodejs/

(La escalabilidad es la propiedad deseable de un sistema, una red o un proceso, que indica su habilidad para reaccionar y adaptarse sin perder calidad, o bien manejar el crecimiento continuo de trabajo de manera fluida, o bien para estar preparado para hacerse más grande sin perder calidad en los servicios ofrecidos.)( https://www.tokioschool.com/noticias/detalles-escalabilidad-en-redes/)

**Express**

Es un framework de nodeJs que nos permite crear aplicaciones web del lado del servidor, cuando se trabaja en una aplicación web se usa html, css y JavaScript que son enviados por el servidor web y también se pueden recibir datos de usuarios, imágenes, videos etc. Al final del lado del servidor se controla la lógica que se va a mostrar al usuario del lado del front-end y para lograr todo esto se necesita el lenguaje de programación como PHP o Ruby, pero a partir del año 2010 se empezó a popularizar el concepto de JavaScript para normalizar el uso de un solo lenguaje de programación del lado del servidor y express es una herramienta que nos puede ayudar a construir una aplicación real utilizando JavaScript puro.

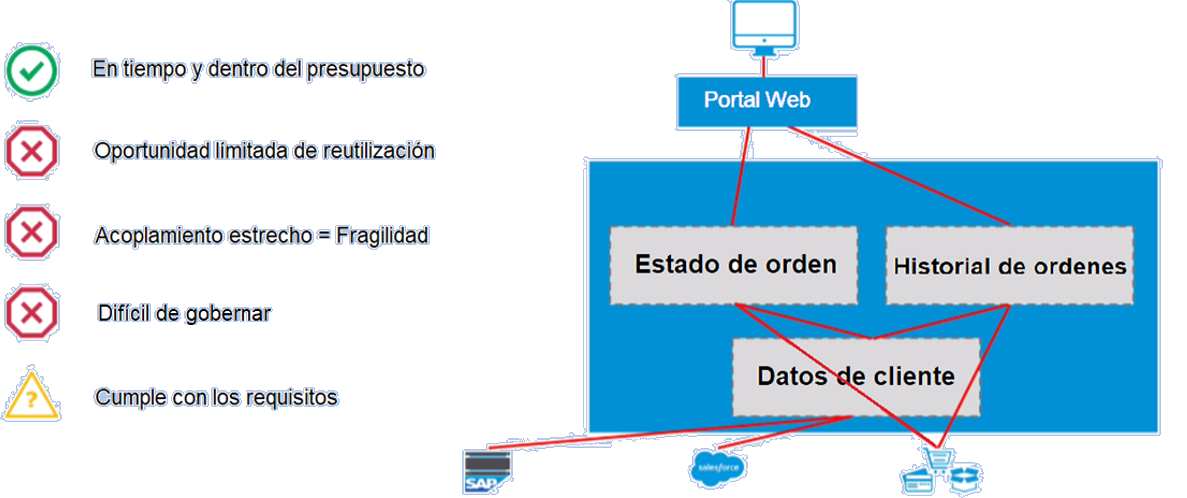
A su vez express es un módulo de terceros, esto quiere decir que en lugar de escribir código puro desde cero en el back-end utilizando JavaScript, también puedes utilizar código JavaScript de otras personas para poder crear el servidor. Este tipo de contribuciones es algo bastante común en la industria del software ya que muchas aplicaciones se complementan de módulos de terceros. Esta característica convierte a express en una herramienta bastante productiva.

Express brinda todas las herramientas para crear el servidor web, esto quiere decir que se complementa con otros módulos de back-end, en el caso de este proyecto se complementa con la comunicación a los recursos de la API que se explicarán más adelante, pero también hay módulos enfocados a conectarse a una base de datos directamente, para subir imágenes u otros para subir videos, etc.

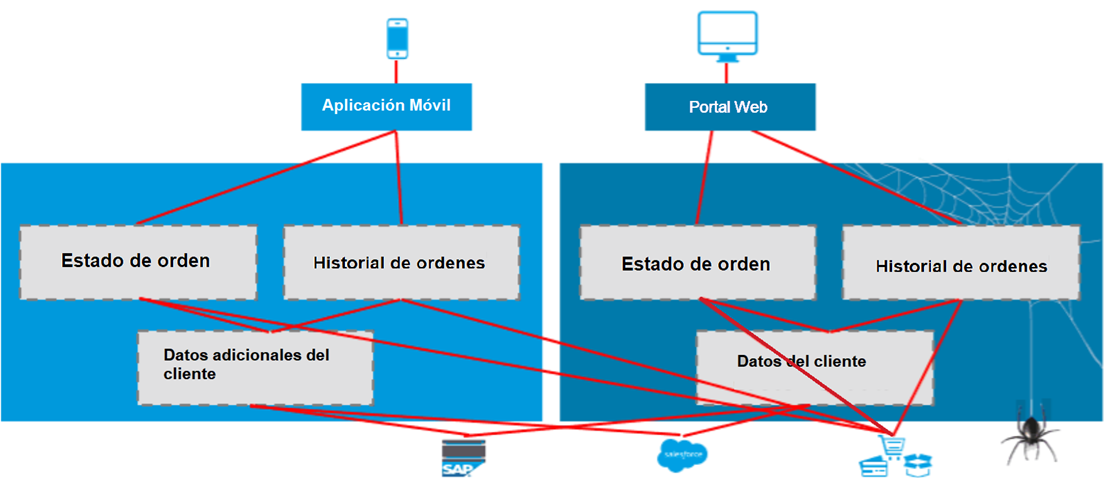
**CAPÍTULO 2 – Análisis**

**Desventajas de los Sistemas Actuales de Servicios Remotos**

Desde hace algunos años los sistemas de servicios y portales web siguen una arquitectura descrita en la figura (e), en dónde notamos que el Portal web esta enganchado a una lógica y funcionalidad directamente, explicado de otra manera es un portal con acoplamiento estrecho a su back-end. Esta arquitectura tiene más desventajas que ventajas enfocadas hacia el futuro, como única ventaja se tiene que es un desarrollo de implementación rápido. Sin embargo, se tienen oportunidades de reutilización de lógica reducida debido al acoplamiento estrecho y esto lo hace también difícil de gobernar. Un cambio en la base del sistema significaría un cambio completo en la lógica superior.



Ahora imaginemos que este portal desea añadir una aplicación móvil para acceder a los servicios desde un teléfono celular, la arquitectura que se tendría que seguir se muestra en la figura (d).

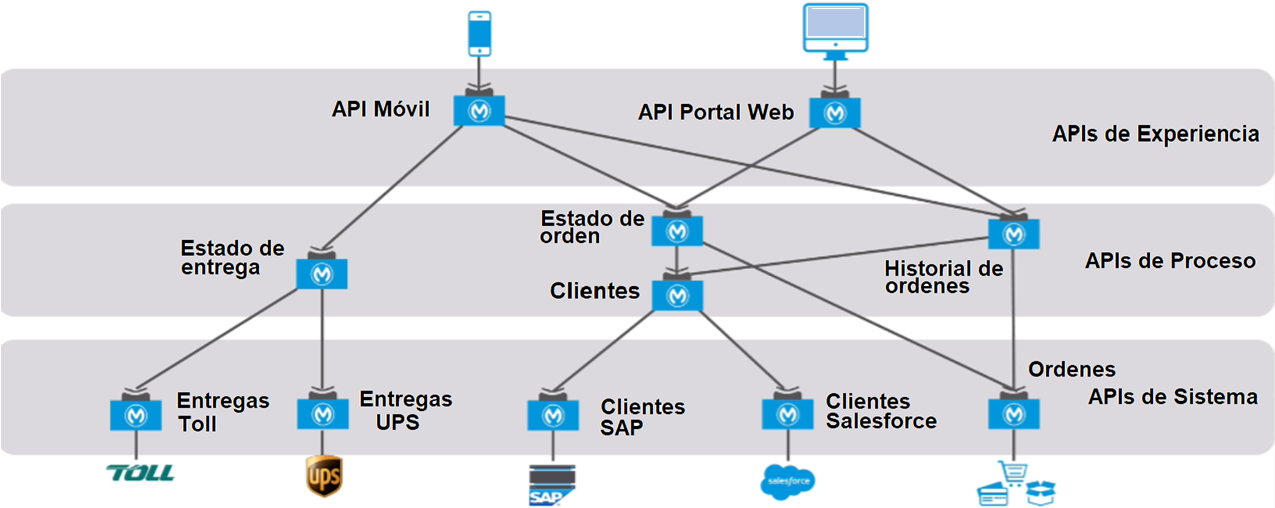


Esto implica desarrollar un sistema totalmente nuevo al que se engancharía la aplicación móvil, esto representa costos de desarrollo nuevamente. A su vez la aplicación Móvil no puede hacer uso de las funcionalidades que el desarrollo del Portal Web implementa, siguiendo este modo de trabajo encontramos que es ineficiente y sin dinamismo.

**Propuesta de solución**

**Implementación de un servicio por medio de una Interfaz de Aplicación Programable**

En los últimos años el auge de las APIs es debido a que las organizaciones están buscando un mayor dinamismo y versatilidad en sus sistemas tanto internos como a los expuestos públicamente, que se puedan comunicar y generar lógica con otros sistemas, y que también sus implementaciones puedan ser de utilidad a otras organizaciones por medio de políticas de relación establecidas. Esta nueva tecnología se basa en el API-Led Connectivity (véase figura (w)).



**Capas lógicas de la conectividad API-Led**

Sistema:

Básicamente las APIs de Sistema exponen información sensible de una organización. No deben ser expuestas para su uso público.

Proceso:

El objetivo principal de las APIs de Proceso es encapsular lógica de negocio o procesos independientemente de los sistemas fuente (APIs de sistema) desde donde la información se origina. Lógicas de negocio o de proceso pueden ser compartidas con toda la organización.

Experiencia:

En este punto ya se tiene la información de una organización expuesta de una manera privada en las APIs de sistema, y también la orquestación de esta información para generar lógica en las APIs de proceso. Las APIs de experiencia son aquellas que se exponen de una manera púclica para su consumo. Son el producto final de la organización. Además en este nivel es en donde se pueden aplicar políticas de consumo.

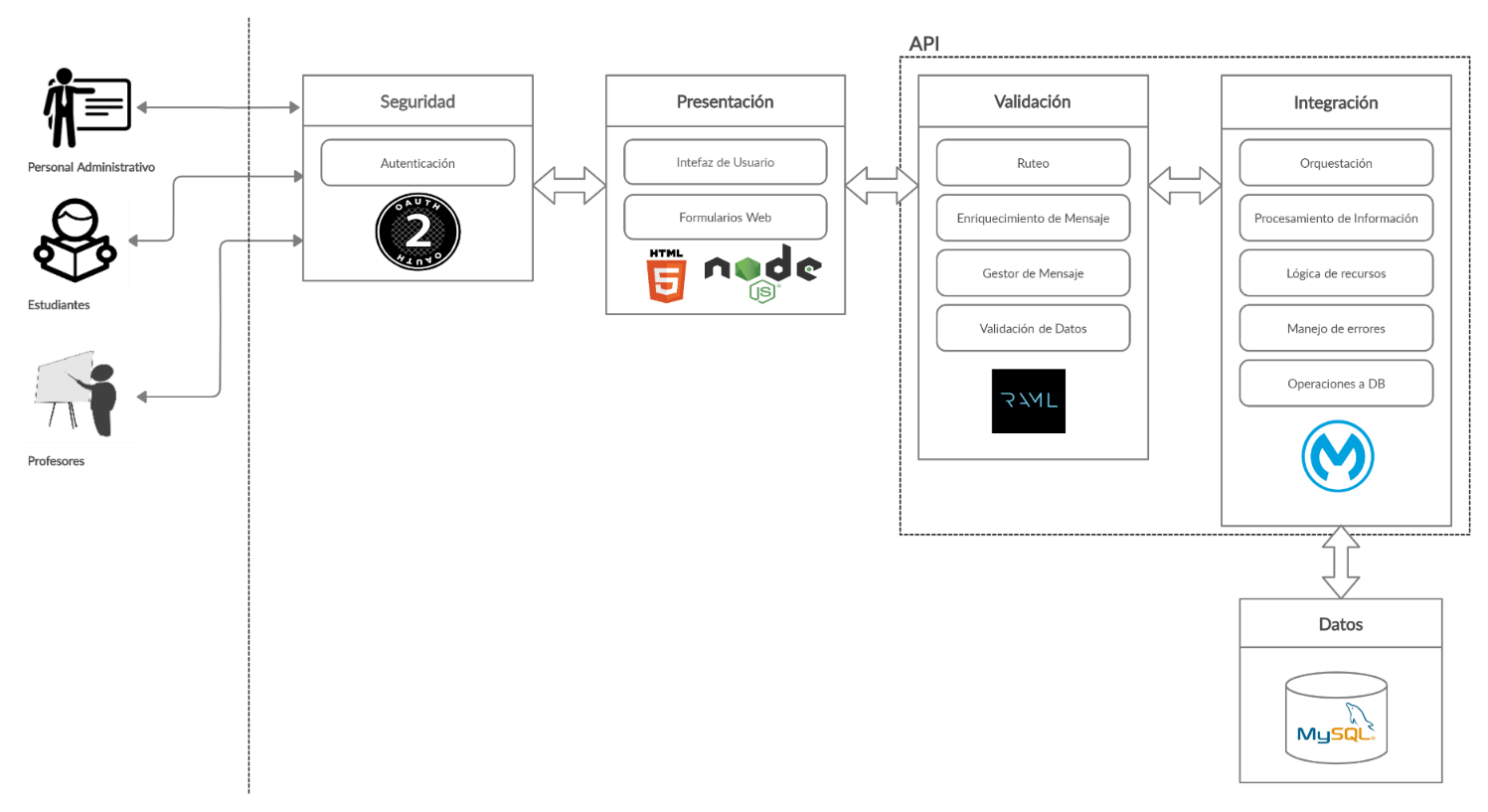
**Conclusión**

El desarrollo de este proyecto emplea en su back-end una API desarrollada con MuleSoft con el objetivo de aprovechar las bondades que nos brinda este tipo de arquitectura hacia el futuro, ya que siempre se podrá reutilizar el trabajo presente para poder enriquecer sistemas de gestión escolar, además de que es altamente adaptable a las necesidades especificas en cuanto a las políticas de información que a la institución le interese mantener.

**CAPÍTULO 3 – Diseño**

**ARQUITECTURA**

El diagrama de bloques de alto nivel de este proyecto se puede estudiar con ayuda de la figura (x). El flujo de ejecución del sistema es de izquierda a derecha, cuando la API ejecuta alguna operación en la base de datos responde el contenido o el estatus correspondiente a la capa de presentación para ser descrita a los usuarios finales, dando un resultado exitoso o también la propagación de un error debidamente manejado y descrito para ser entendido por el usuario.

****

**OPERATIVIDAD PARA CADA AGENTE**

|  |  |
| --- | --- |
| **Agentes** | **Funciones** |
| Personal Administrativo | * Alta de un profesor * Alta de un alumno * Actualización de un contacto * Actualización de una dirección * Asignar un profesor a un curso |
| Profesores | * Obtener lista de cursos disponibles * Consultar información sobre un curso disponible * Inscribir alumno a un curso * Remover alumnos inscritos a un curso * Actualizar el programa de un curso |
| Estudiantes | * Obtener lista de cursos disponibles * Consultar información sobre un curso disponible * Inscribirse a un curso |

**PROPUESTA DE PORTAL WEB**

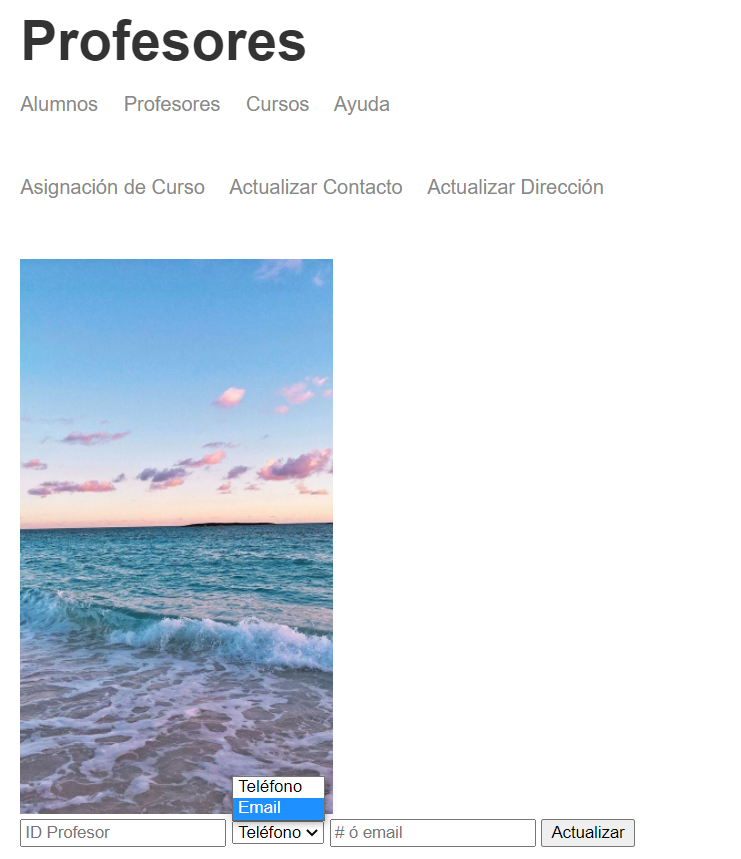
En esta sección se describe el portal web propuesto, este portal web hace las peticiones a la API por medio de un servidor que se construye con Express de NodeJs. El diseño de este portal web es sencillo de utilizar y también puede ser personalizado en un futuro por la institución académica que lo implemente.

Se tienen dos cabeceras principales, en la primera encontraremos enlaces a Alumnos, Profesores, Cursos y Ayuda. Cada una de ellas nos llevará a una nueva pantalla y dependiendo el tema (Alumnos, Profesores etc.) habrá otra cabecera personalizada con las funcionalidades que se asocian a la cabecera superior.

En la figura (q) se observa que un alumno puede consultar sus datos por medio de su matrícula simplemente escribiéndola en el formulario y presionando el botón de consulta.

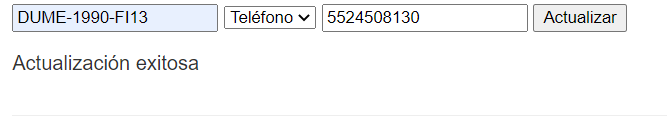


En la figura (w) estamos situados en la sección de *Profesores* y en la subsección Actualizar Contacto. Cuando el profesor tiene la necesidad de actualizar su contacto debe proveer su ID seleccionar el tipo de contacto y escribir el nuevo valor posteriormente presionar el botón Actualizar para ejecutar la operación.

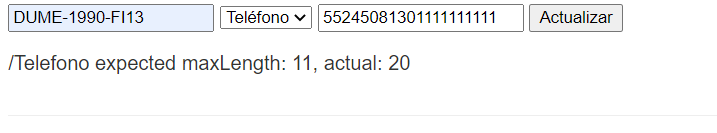


Cuando se presiona el botón Actualizar la vista despliega ‘Actualización exitosa’ dependiendo si la operación es válida o en caso contrario describe el error.

* 1. Operación exitosa



* 1. Operación fallida, en Teléfono solo se permiten 11 caracteres



**Plataforma de desarrollo de API propuesta**

**RECURSOS REST PROPUESTOS Y FUNCIONALIDAD**

[/cursos](https://anypoint.mulesoft.com/designcenter/designer/#/cursos)

**GET:** Este método permite obtener la lista de cursos

[/cursos/{clave}](https://anypoint.mulesoft.com/designcenter/designer/#/cursos/{clave})

**GET:** Permite obtener información de un curso en específico con la clave del curso

/cursos/{clave}/alumno

**PUT:** Permite a un alumno inscribirse a un curso con la clave del curso

**DELETE:** Permite dar de baja a un alumno de un curso con la clave del curso

[/profesor](https://anypoint.mulesoft.com/designcenter/designer/#/profesor)

**POST:** Este recurso permite dar de alta un profesor

/profesor/{numeroEmpleado}

**GET:** Permite obtener la información de un profesor con su numero de empleado

/profesor/{numeroEmpleado}/dirección

**PUT:** Permite actualizar la dirección de un profesor con su número de empleado

[/profesor/{numeroEmpleado}/contacto](https://anypoint.mulesoft.com/designcenter/designer/#/profesor/{numeroEmpleado}/contacto)

**PUT:** Permite actualizar el contacto de un profesor con su número de empleado

[/profesor/{numeroEmpleado}/asignar/{clave}](https://anypoint.mulesoft.com/designcenter/designer/#/profesor/{numeroEmpleado}/asignar/{clave})

**PUT:** Permite asignar un profesor a un curso con la clave del curso y con su número de empleado

[/alumno](https://anypoint.mulesoft.com/designcenter/designer/#/alumno)

**POST:** Este recurso permite dar de alta un alumno

[/alumno](https://anypoint.mulesoft.com/designcenter/designer/#/alumno)/{matricula}

**GET:** Este recurso permite obtener los datos de un alumno

[/alumno/{matricula}/contacto](https://anypoint.mulesoft.com/designcenter/designer/#/alumno/{matricula}/contacto)

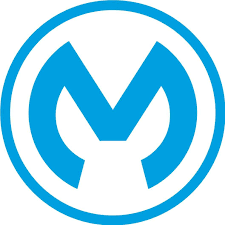
**PUT:** Permite actualizar el contacto de un alumno por medio de su matricula

/alumno/{matricula}/direccion

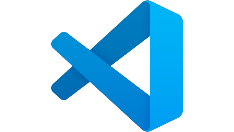
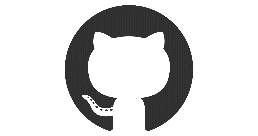
**PUT:** Permite actualizar la dirección de un alumno con su matricula

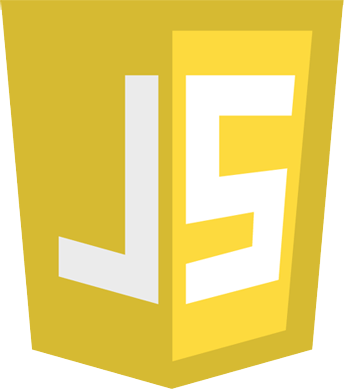
**HERRAMIENTAS DE SOFTWARE Y TECNOLOGIAS**

********

** **

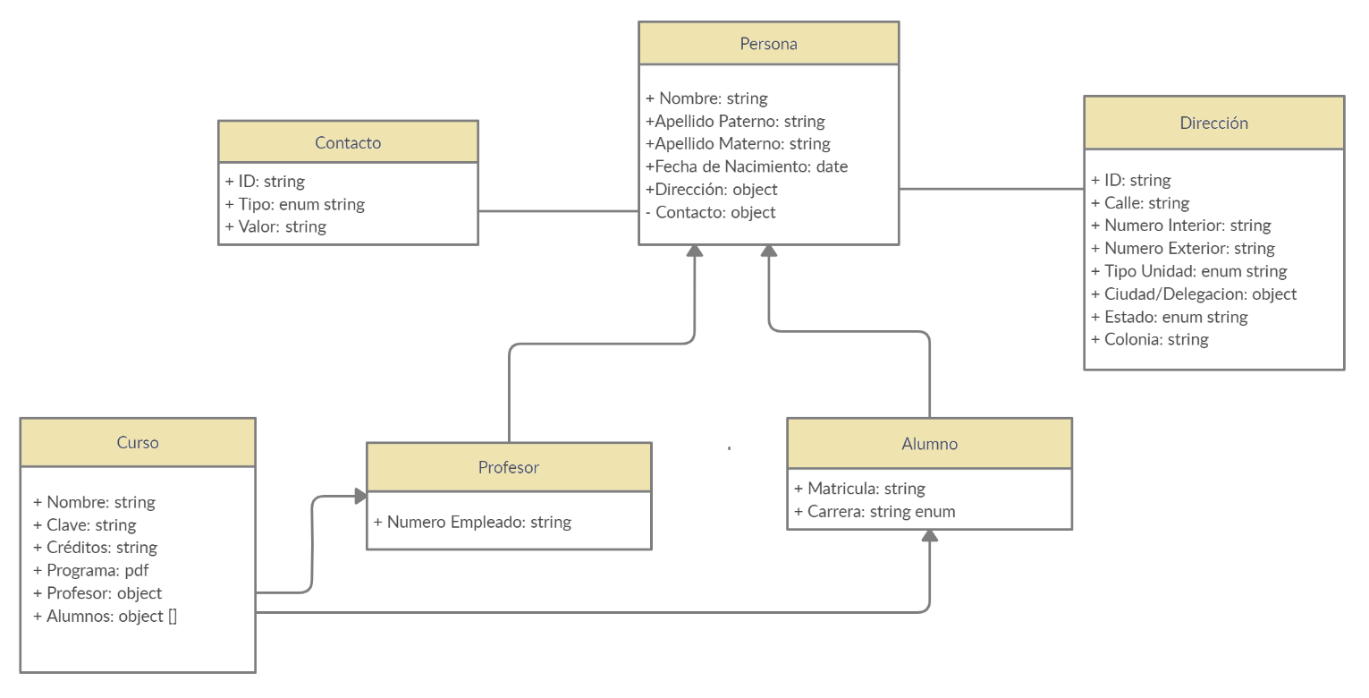






**ESTRUCTURAS DE DATOS**

En la figua (y) se puede hacer un análisis de las estructuras de datos que utiliza este proyecto. Esta es la información de los usuarios que abarca el proyecto, en donde se pueden hacer generalizaciones o especialidades de datos u objetos, por ejemplo un profesor o un alumno se pueden generalizar como una persona, e inversamente un profesor es una especialidad del grupo que engloba a las personas y que tiene una propiedad exclusiva que lo identifica (Numero Empleado), a su vez el objeto persona esta relacionado con otros objetos de diferente naturaleza como una dirección y un contacto, esto permite a nivel de desarrollo la reutilización de estructuras de datos que están relacionadas y al mismo tiempo son independientes, en base a esta estructura se desarrolló la estructura de la base de datos.



**DICCIONARIO DE DATOS**

El objetivo en esta sección es documentar y describir un diccionario de datos en el que se basan las validaciones de entrada/salida de la especificación RAML para cada propiedad de cada objeto. Marcar reglas de sintaxis, contenido y de nomenclaturas ayuda a tener un mejor control de la información que entra/sale de la API REST y evitar por ejemplo guardar información basura. El diccionario de datos se basa en las estructuras de datos descritas en el capítulo anterior.

**Persona**

|  |  |  |  |
| --- | --- | --- | --- |
| **Atributo** | **Tipo** | **Requerido** | **Nota** |
| Nombre | String | Sí | No se aceptan números. Longitud Máxima: 60 |
| Apellido Paterno | String | Sí | No se aceptan números. Longitud Máxima: 30 |
| Apellido Materno | String | Sí | No se aceptan números. Longitud Máxima: 30 |
| Fecha de Nacimiento | String | Sí | Formato: yyyy-MM-dd |
| Dirección | Objeto | Sí |  |
| Contacto | Objeto |  |  |

**Profesor**

|  |  |  |  |
| --- | --- | --- | --- |
| **Atributo** | **Tipo** | **Requerido** | **Nota** |
| Número Empleado | String | Sí | Los números de empleados se forman de la siguiente manera:  XXXX-9999-XX99  Donde:  X representa una letra  9 representa un dígito |

**Alumno**

|  |  |  |  |
| --- | --- | --- | --- |
| **Atributo** | **Tipo** | **Requerido** | **Nota** |
| Matricula | String | Sí | La matrícula de un alumno estacompuesta de la siguiente manera:  99999999-XXXXXX-XX99  Donde:  X representa una letra  9 representa un dígito |
| Carrera | String | Si | La institución ofrece las siguientes carreras:   * Actuaria * Administración * Agropecuaria * Antropologia * Arquitectura * Arquitectura de Paisaje * Arte y Diseno * Artes Visuales |

**Dirección**

|  |  |  |  |
| --- | --- | --- | --- |
| **Atributo** | **Tipo** | **Requerido** | **Nota** |
| Calle | String | Sí | Longitud mínima de 1 y máxima de 80 caracteres, donde solo se permiten letras y números |
| Numero Exterior | String | Si | Longitud mínima de 1 y máxima de 30, solo se permiten dígitos y letras |
| Tipo de Unidad | String | Si | Limitado a los siguientes valores:   * Departamento * Edificio * Casa |
| Numero interior | String | No | Longitud mínima de 1 y máxima de 30, solo se permiten dígitos y letras |
| Ciudad | String | No | Longitud mínima de 1 y máxima de 30, solo se permiten dígitos y letras |
| Estado | String | Si | Limitado a los 32 estados de la república Mexicana |
| Colonia | String | Si | Longitud máxima de 60 caracteres |

**Contacto**

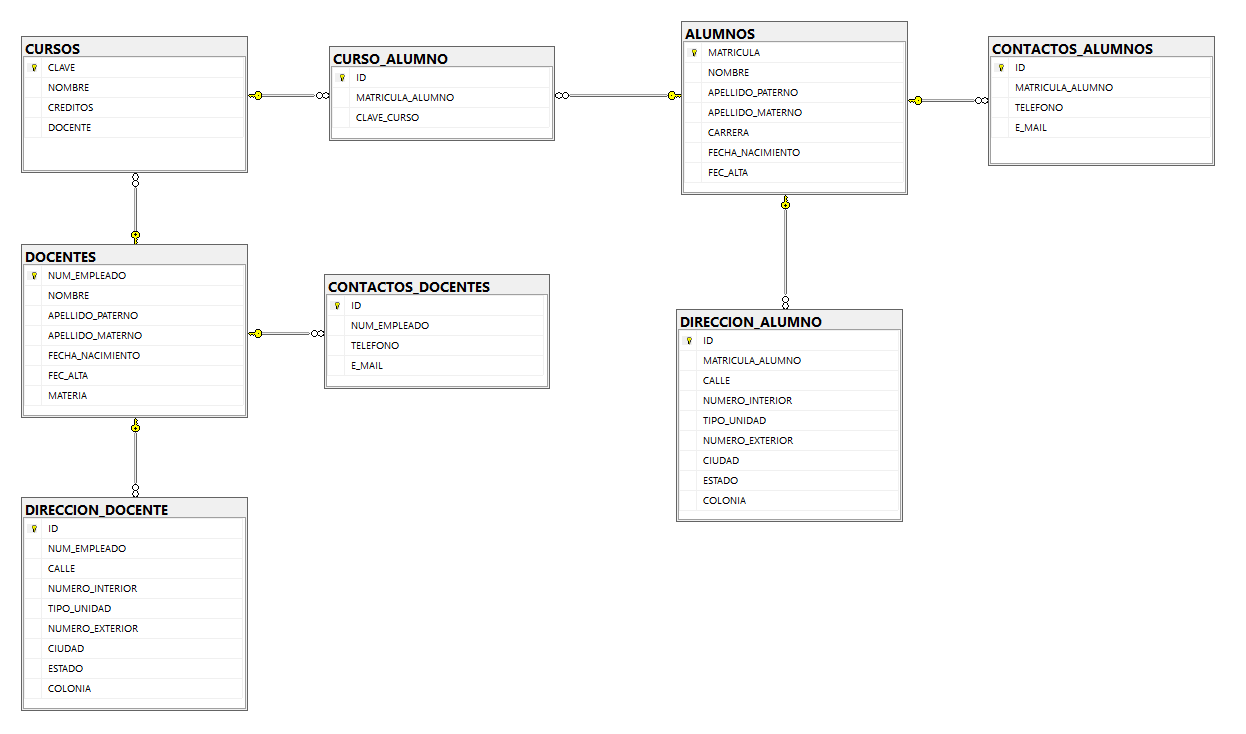
|  |  |  |  |
| --- | --- | --- | --- |
| **Atributo** | **Tipo** | **Requerido** | **Nota** |
| Tipo | Stirng | Sí | Limitado a:   * Email * Teléfono |
| Valor | String | Si | Valor del contacto |

**Curso**

|  |  |  |  |
| --- | --- | --- | --- |
| **Atributo** | **Tipo** | **Requerido** | **Nota** |
| Clave | String | Sí | Las claves de cursos tendrán la siguiente estructura:  XXXX-9999 |
| Nombre | String | Si | Máximo 120 caracteres de longitud, donde solo se permiten letras y números |
| Créditos | String | Si | Numero de créditos del curso |
| Profesor | Objeto | Si | Un curso puede tener solo un profesor asignado |
| Alumnos | Objeto | Si | Un curso puede tener un máximo de 22 alumnos inscritos |

**DISEÑO DE LA BASE DE DATOS**

En la figura (z) se puede observar el diseño de la base de datos de este proyecto, cada una de las tablas tiene relación con otra por medio de llaves foráneas con relaciones de renglón uno a uno y uno a muchos. Como ejemplo de una relación uno a uno se tiene entre la tabla **ALUMNOS** y **CONTACTOS\_ALUMNOS,** en dónde una matrícula en la columna **MATRICULA** de **ALUMNOS** estará relacionada únicamente con una matrícula de la columna **MATRICULA\_ALUMNO** de **CONTACTOS\_ALUMNOS**, la columna **MATRICULA\_ALUMNO** es llave foránea de la columna **MATRICULA** en **ALUMNOS** y permite asegurar que se asocien únicamente contactos a matriculas existentes en la tabla de **ALUMNOS,** en este caso se tienen almacenados el teléfono y correo electrónico del alumno en el sistema. Dos ejemplos de relación de renglones uno a muchos se puede observar en la tabla **CURSO\_ALUMNO** en donde se consigue que un alumno esté inscrito en diferentes cursos gracias a esta relación, esto quiere decir que una matrícula de un alumno en **ALUMNOS** estará presente en repetidas ocasiones en **CURSO\_ALUMNO** asociándola a diferentes claves de cursos.



**CAPÍTULO 4 – Implementación**

**CONSTRUCCION DE LA API**

**Especificación RAML**

El siguiente fragmento de RAML sirve para ejemplificar la definición de dos nodos que contiene la API. Como se estudio anteriormente lo primero que se define es el nombre del recurso (*/alumno*) después el verbo HTTP que este recurso ejecuta (POST). En el caso del verbo HTTP POST se ejecuta con la finalidad de crear una entidad nueva en el servidor en este caso a un alumno, por lo tanto, este recurso espera un cuerpo de entrada para procesarlo y almacenarlo siguiendo la estructura de datos **Alumno** que revisaremos más adelante. Finalmente, el código de respuesta HTTP que se espera en caso de que la ejecución POST sea exitosa es un *201*, también devuelve un objeto con la propiedad **matricula** que se construye en la API en base a información del cuerpo de entrada y sirve para generar un vector de información en la base de datos asociada a este valor.

Dentro del recurso */alumno* tenemos un segundo recurso anidado que permite el filtrado de un alumno por medio de un parámetro en la URI llamado **matricula** y que el servidor ocupa para filtrar el contenido especifico de esa matricula en la base de datos. Cuando el contenido es obtenido de la base de datos el API construye un cuerpo de respuesta tomando esa información y devolverla al usuario cliente, como respuesta se recibe una estructura de datos tipo **Alumno**.

/alumno:

post:

is: [errores-comunes]

description: Este recurso permite dar de alta a un alumno.

body:

type: type.Alumno

example: !include examples/alumno.json

responses:

201:

description: Alta exitosa.

body:

type: object

properties:

matricula: string

example:

{ "matricula": "19950227-SERROD-CA17"}

/{matricula}:

description: Este recurso permite actualizar el contacto de un alumno con su matricula.

uriParameters:

matricula:

description: Identificador de alumno.

pattern: "[0-9]{8}-[a-zA-Z]{6}-[a-zA-Z]{2}[0-9]{2}"

get:

responses:

200:

body:

type: type.Alumno

example: !include examples/alumno.json

El fragmento anterior es la especificación RAML que el API utiliza para construir los siguientes recursos:

**POST** /alumno

**GET** /alumno/{matricula}

El nodo *pattern* contiene una expresión regular para asegurar la sintaxis definida en el diccionario de datos, si la petición no cumple con esta sintaxis en el campo matrícula el servicio propaga un código de error **400 Bad Request** (Petición no valida).

En la figura (q) tenemos la estructura de datos **Alumno** al igual que ella esta API maneja otras estructuras como **Profesor, Curso, Dirección** etc, que se podrán analizar en la especificación RAML completa en la parte de los Anexos.

Aquí se puede observar la reutilización de estructuras de datos en el tipo **Alumno** que es del tipo **Persona** y también tiene la propiedad **carrera** que lo identifica, la propiedad **carrera** tiene un nodo **enum** que permite restringir los valores que **carrera** acepta y gracias a esto se evita guardar información basura en la base de datos, también se personaliza en base a las carreras que ofrece una institución educativa. A la vez **Persona** contiene las propiedades de **dirección** y **contacto** que ocupa otras estructuras de datos y se pueden revisar con detalle en la especificación RAML completa.

types:

Alumno:

type: Persona

properties:

carrera:

enum: ["Actuaria","Administracion","Administracion Agropecuaria","Administracion de Archivos y Gestion Documental","Antropologia","Arquitectura","Arquitectura de Paisaje","Arte y Diseno","Artes Visuales"]

Persona:

properties:

Nombre:

maxLength: 30

apellidoPaterno:

maxLength: 30

apellidoMaterno:

maxLength: 30

fechaNacimiento: date-only

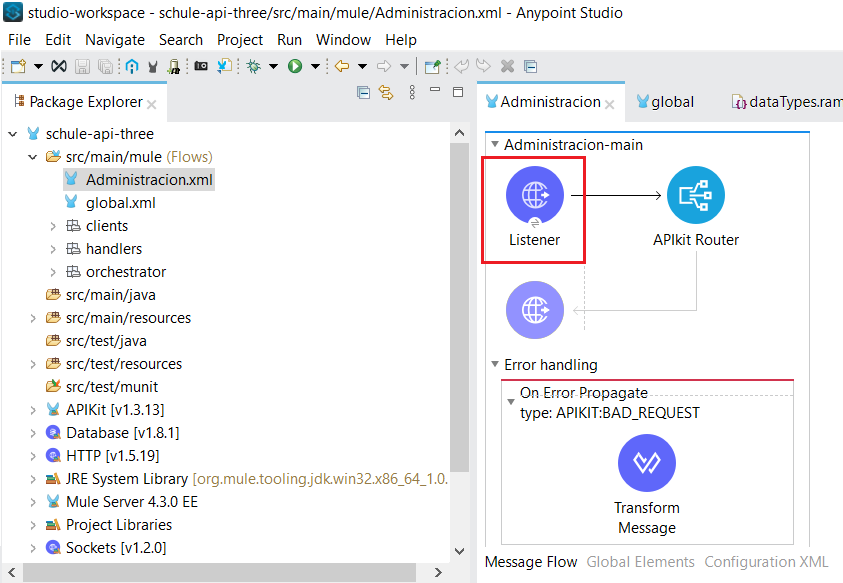
direccion: Direccion

contacto?: Contacto

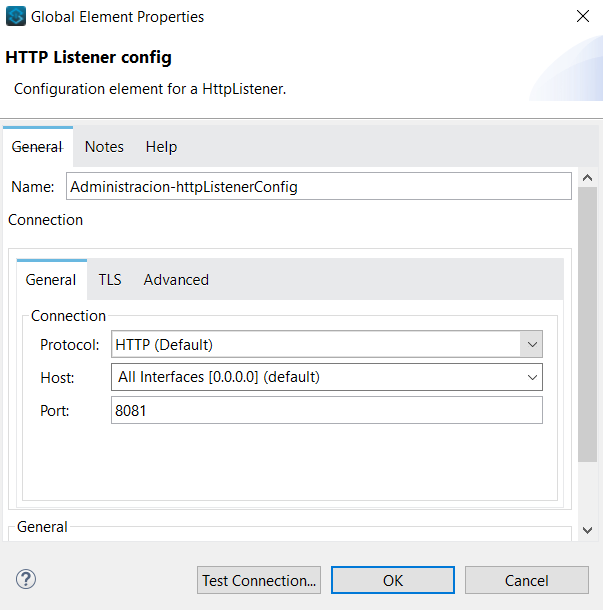
**DESARROLLO DE FLUJOS**

El desarrollo de la API esta implementado con el IDE de la plataforma MuleSoft es decir Anypoint Studio 7.7. Es un IDE bastante práctico con una vista gráfica y otra en XML para hacer el desarrollo de flujos.

En el flujo principal se tiene configurado un HTTP Listener, a través del puerto 8081 y con una dirección a la máquina local. Este elemento crea el nodo que estará esperando peticiones por parte del cliente. En la figura (+) podemos observar gráficamente el flujo principal (Administracion-main) y el nodo HTTP Listener al cual llegarán las peticiones.



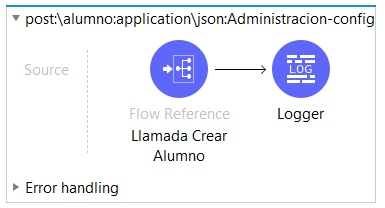
En la figura (i) podemos observar la configuración del nodo HTTP Listener.

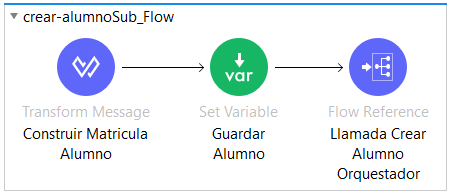


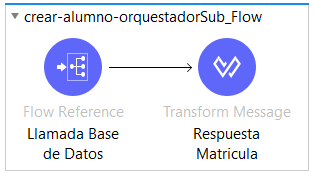
Después del nodo HTTP Listener en la figura (+) podemos observar el elemento APIKit Router que toma la especificación RAML y genera los flujos correspondientes a cada recurso que existe en la especificación. El APIKit Router además de rutear como su nombre lo indica, hace las validaciones de información de entrada también declaradas en la especificación RAML. En caso de que la petición no siga las especificaciones del RAML, el APIKit Router propagará un error **400 Bad Request** regresando una descripción de error en un objeto JSON. Si la petición (Verbo HTTP, Cabeceras, Cuerpos, Datos, Dirección) es correcta el APIKit router encauzará la ejecución hacia el flujo correspondiente.

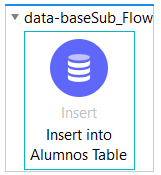
En la figura (+) observamos una estructura de directorios del lado izquierdo, la ruta **src/main/mule** contiene todos los flujos configurados. Adentro de esta ruta existen otros directorios, **handlers**, **clients**, **orchestrator**. Esta estructura facilita la implementación de las funcionalidades, en un flujo **handler** se hacen guardados de variables o transformaciones comúnmente. En un **orquestador** se hacen peticiones a diferentes clientes e implementar lógica o a uno solo y por último un flujo **cliente** solo se encargará de hacer una llamada a un servicio externo, por ejemplo, en este proyecto los flujos **cliente** se conectan a la base de datos.

En la figura (3) tenemos la secuencia de flujos para crear a un alumno, cuando el APIKit Router ha validado la petición, este dirige la petición al flujo *post:\alumno,* posteriormente este flujo hará una llamada al flujo *crear-alumnoSub\_Flow,* adentro de este flujo se genera en base a datos del alumno su matrícula y también se guarda el cuerpo con la información del alumno, finalmene el handler hace una llamada al orquestador. El orquestador hace la llamada al flujo que contiene la operación a la base de datos y construye una respuesta que contiene la matricula y se devolverá al cliente, en este acaso al alumno nuevo. Como último flujo tenemos el cliente de la base de datos, dentro tenemos la operación INSERT que guarda la información del alumno en las tablas correspondientes. La matrícula generada en el handler ayuda a relacionar la información del alumno en las diferentes tablas de la base de datos.





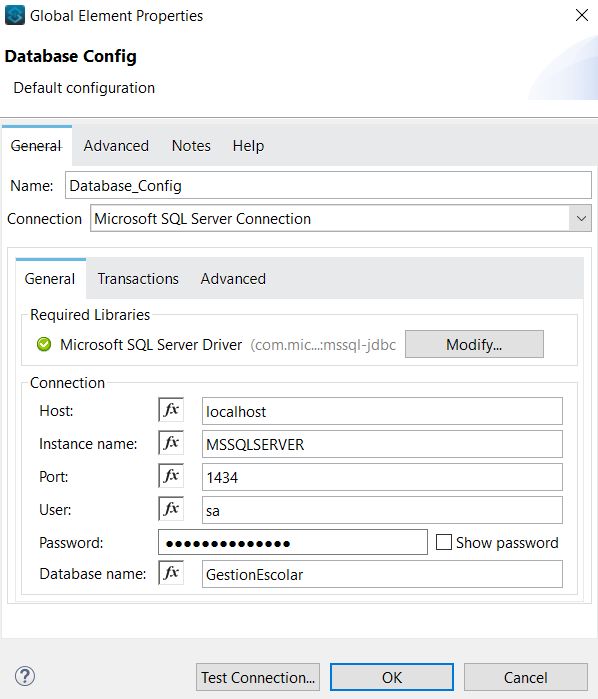




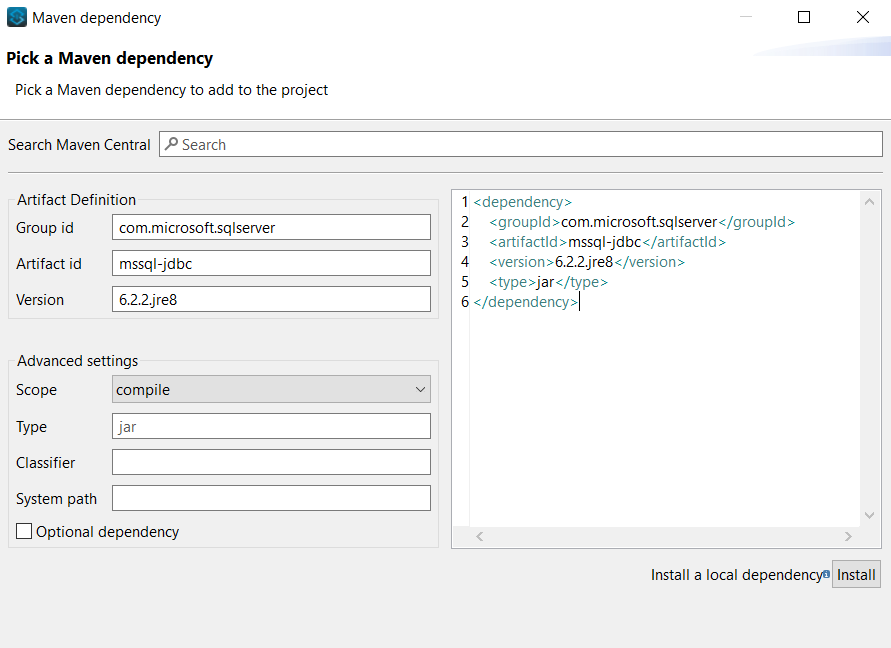
De esta manera se implementan las otras funcionalidades que contiene la API, cambiando en lógica, guardado de variables, decisiones de ruta etc. Se pueden consultar en el artefacto de la API, que se encuentra en el repositorio remoto de manera libre.

**ENLACE CON BASE DE DATOS**

La configuración de la conexión a la base de datos se hace por medio de una configuración global dentro de la API para especificar el **JDBC** y configurar sus características. En la figura (f) se muestra la configuración para la interacción con la base de datos **GestionEscolar**.



La clase del controlador java (**mssql-jdbc**) es una dependencia de maven que se agrega en los Objetos Modelos de Proyecto **POM** (Project Object Model) para ser incorporada al momento de compilar la API:

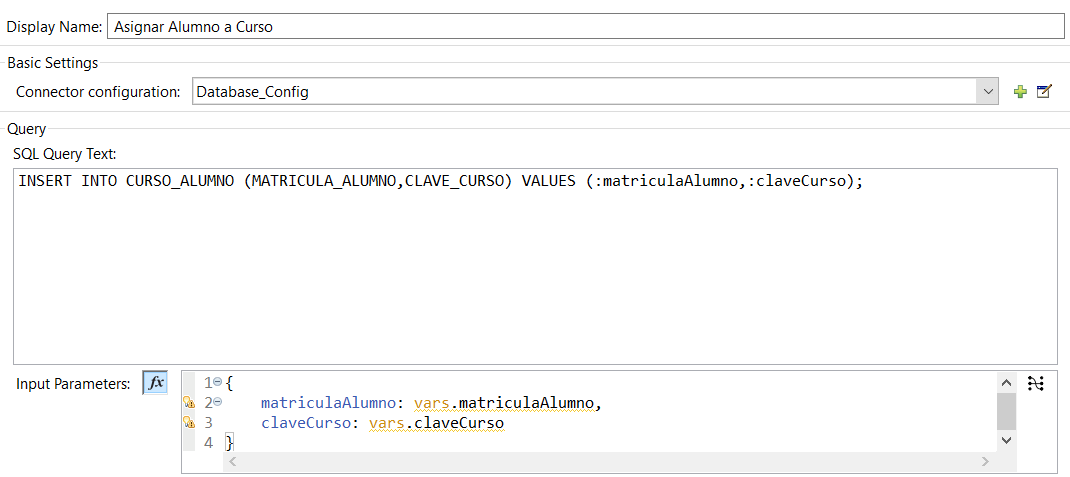


La configuración **DataBase\_Config** definida globalmente se utiliza en cada una de las operaciones que ejecuta la API en la base de datos, variando solamente el lenguaje estructurado de petición **SQL** (Structure Query Language).

En este proyecto se implementan solamente las siguientes operaciones **SQL**:

* **SELECT**
* **INSERT**
* **UPDATE**
* **DELETE**

Un ejemplo de operación **INSERT** a la base de datos utilizado código **SQL** empleado en este proyecto se muestra en la figura (i). Esta operación pertenece al recurso que asocia un alumno a una materia por medio de la matrícula que obtiene al registrarse y la clave del curso respectivamente, en donde los datos que se insertan son obtenidos de la petición que hace el usuario cliente y que en este paso ya están validados.



Despliegue

**DESARROLLO DE BASE DE DATOS**

En esta sección se describen un par de sintaxis que se emplean para generar una de las secciones de la base de datos y que también se utiliza para generar todas las secciones y tablas restantes.

Para crear las tablas se usa el lenguaje de definición de datos **DDL** (Data Definition Language) que se encarga también de la modificación de la estructura de los objetos de la base de datos.

El siguiente fragmento **DDL** define la tabla llamada **ALUMNOS** con las columnas **MATRICULA**, **NOMBRE**, **APELLIDO\_PATERNO**, **APELLIDO\_MATERNO**, teniendo como llave primaria a **MATRICULA** que identifica cada renglón y a cada alumno. Cada columna tiene una longitud definida de tipo carácter y no pueden tener un valor nulo.

CREATE TABLE ALUMNOS(

MATRICULA VARCHAR(20) PRIMARY KEY NOT NULL,

NOMBRE VARCHAR(30) NOT NULL,

APELLIDO\_PATERNO VARCHAR(30) NOT NULL,

APLLIDO\_MATERNO VARCHAR(30) NOT NULL

);

El siguiente fragmento **DDL** define la tabla llamada **CONTACTOS\_ALUMNOS**, y se observa cómo generar una llave foránea. La llave foránea esta en la columna **MATRICULA\_ALUMNO** de **CONTACTOS\_ALUMNOS** que está asociada con la columna de llave primaria **MATRICULA** de **ALUMNOS**.

CREATE TABLE CONTACTOS\_ALUMNOS(

ID INT PRIMARY KEY IDENTITY(1,1) NOT NULL,

MATRICULA\_ALUMNO VARCHAR(20) NOT NULL,

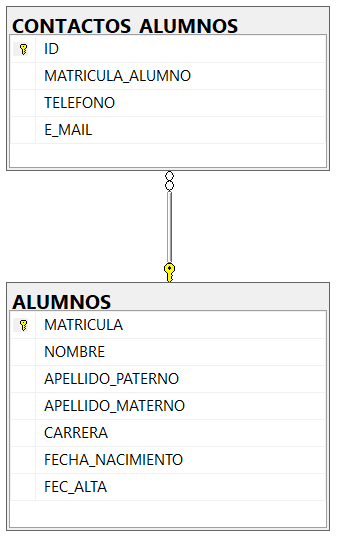
TELEFONO VARCHAR(11) NULL,

E\_MAIL VARCHAR(30) NULL,

CONSTRAINT fk\_contacto\_alumno FOREIGN KEY (MATRICULA\_ALUMNO) REFERENCES ALUMNOS(MATRICULA)

);

Los fragmentos anteriores generan la estructura de datos mostrada en la figura (p)



**DESARROLLO DE PORTAL WEB**

Como se estudio anteriormente, la herramienta o framework con el que se desarrolla el front-end de este proyecto es NodeJs y el modulo express para programar el servidor.

En la figura (4) se tiene la configuración mínima para construir una vista en la ruta /profesores. El módulo express primero se requiere, se inicializa y finalmente configuramos una vista con un verbo GET (por defecto un navegador web hace operaciónes GET al momento de visitar los sitios web), esta configuración la encontramos en la línea que empieza con app.get(). Como parámetros se escribe el nombre de la ruta y una función con dos parámetros (req,res) que se traducen como *petición* y *respuesta,* adentro de esta función ocupamos una propiedad de res (render) para renderizar un archivo HTML llamado ‘profesores’ y el navegador web lo represente.

const express = require('express')

const app = express()

app.get('/profesores', (req,res) => {

    res.render('profesores', {

        title: 'Profesores',

        name: 'César Rodriguez'

    })

})

Después de crear las rutas y renderizar las vistas HTML necesarias se configura el puerto en donde el servidor estará esperando peticiones con otra propiedad de express llamada ‘listen’.

app.listen(3000, () => {

    console.log('Servidor en puerto 3000.')

})

REQUEST

El módulo request es una manera simple de hacer llamadas HTTP con código JavaScript. Este módulo se utiliza para crear las llamadas a los recursos que expone el API, parametrizarlas y configurarlas. Cuando se usa este modulo el portal web se convierte en el cliente de la API. En la figura (2) se muestra un ejemplo del uso de request para llamar al recurso */cursos* que expone la API. La funcionalidad se engloba en la función obtenerCursos y se exporta para poder usarlo fuera de este archivo.

const request = require('request')

const obtenerCursos = (curso,callback) => {

    console.log(curso)

    var url = 'http://localhost:8081/api/cursos'// + curso

    if( curso != '""') {

        var url = `http://localhost:8081/api/cursos/${curso}`

    }

    console.log(url)

    request({ url: url, json: true}, (error, {body}) => {

        console.log(body)

        callback(body)

    }

    )

}

module.exports = obtenerCursos

En el archivo en donde se configuran las vistas se hace la petición de los módulos que hacen la petición a la API con la siguiente línea:

const obtenerCursos = require('./utils/obtenerCursos.js')

Después se crea un recurso que hace la petición a la API con los parámetros necesarios por medio de la función obtenerCursos:

app.get('/api/cursos', (req,res) => {

    if(req.query.codigoCurso == '')

    return res.send ({

        error: 'Debes proveer un filtro de busqueda'

    })

//    console.log(req)

    obtenerCursos(req.query.codigoCurso, (cursos) => {

        res.send(cursos)

    })

})

Finalmente se hacen las peticiones a los recursos dinámicos del servidor por medio de otro modulo llamado *fetch*, se puede observar que desinscripcionAlumno es asignado a un botón de las vistas HTML con ayuda de #Desinscribir, de igual manera los elementos clave, matricula y message-1, que en este caso son formas HTML. Cuando el usuario presiona el botón relacionado al elemento Desinscribir entra en ejecución la función ***addEventListener*** que hace finalmente la petición al recurso del servidor con la información que se recibe de la vista HTML.

const desinscripcionAlumno = document.querySelector('#Desinscribir')

const clave = document.querySelector('#clave')

const matricula = document.querySelector('#matricula')

const mensajeUno = document.querySelector('#message-1')

desinscripcionAlumno.addEventListener ( 'submit', (e) => {

    mensajeUno.textContent = 'Cargando...'

    e.preventDefault()

    const valores = {clave: clave.value, matricula: matricula.value}

    console.log(valores)

    fetch('http://localhost:3000/api/alumno/baja?claveCurso=' + valores.clave + '&' + 'matricula=' + valores.matricula ,{

        method: 'DELETE',

        body: JSON.stringify({ "matricula": valores.matricula }),

        headers: { 'Content-Type': 'application/json' }}

        ).then( (response) => {

                console.log(response),

                mensajeUno.textContent = "Baja exitosa"

})

})

VISTAS HTML

En el siguiente fragmento de código se observa el HTML de la vista para desinscribir a un alumno de un curso. En la línea con la etiqueta script se direcciona al código visto anteriormente en donde se usa el modulo *fetch*. Este fragmento HTML es también el que renderiza el módulo express.

<!DOCTYPE html>

<html lang="en">

<head>

    <title>Alumnos</title>

    <link rel="icon" href="img/alumno.jpg">

    <link rel="stylesheet" href="/css/style.css">

</head>

<body>

    <div class="main-content">

        {{>header}}

        {{>alumnosHeader}}

    <img class="portrait" src="/img/palmeras.jpg">

        <form id="Desinscribir">

            <input id='matricula' name='matricula' placeholder="Matricula">

            <input id='clave' name='clave' placeholder="Código de Curso">

            <button>Dar de baja</button>

        </form>

        <p id="message-1"></p>

    </div>

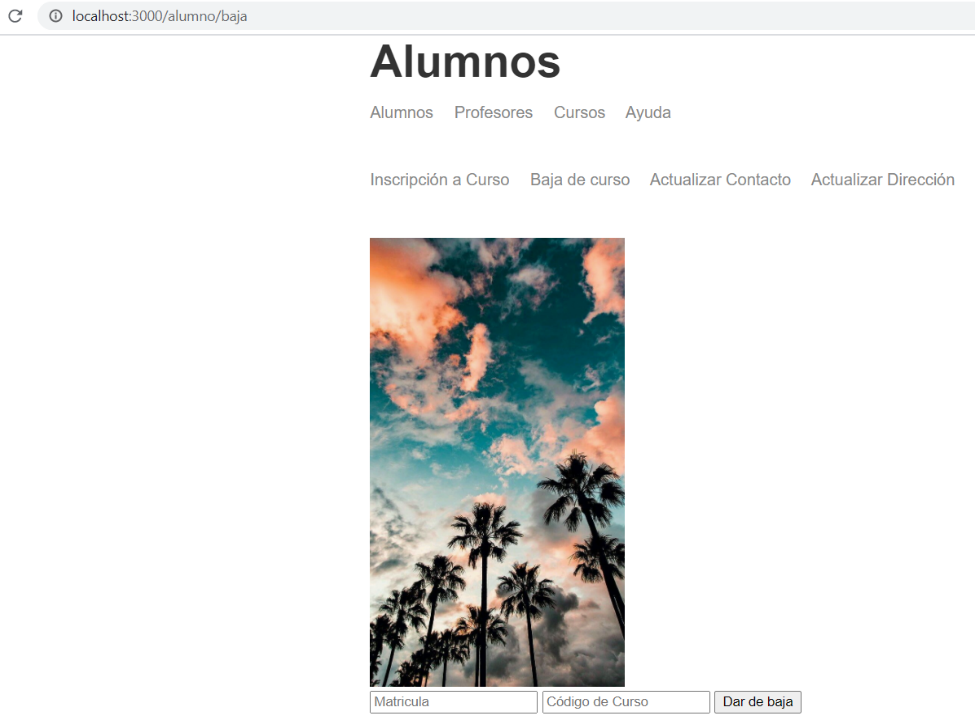
    {{>footer}}

    <script src="/js/desinscribirAlumno.js"></script>

</body>

</html>

VISTA GRAFICA



[1] <https://definicion.de/acceso-remoto/>

[2] <https://blog.gft.com/es/2020/04/28/cuales-son-los-aspectos-clave-de-la-digitalizacion-en-tiempos-de-pandemia-y-post-pandemia/>