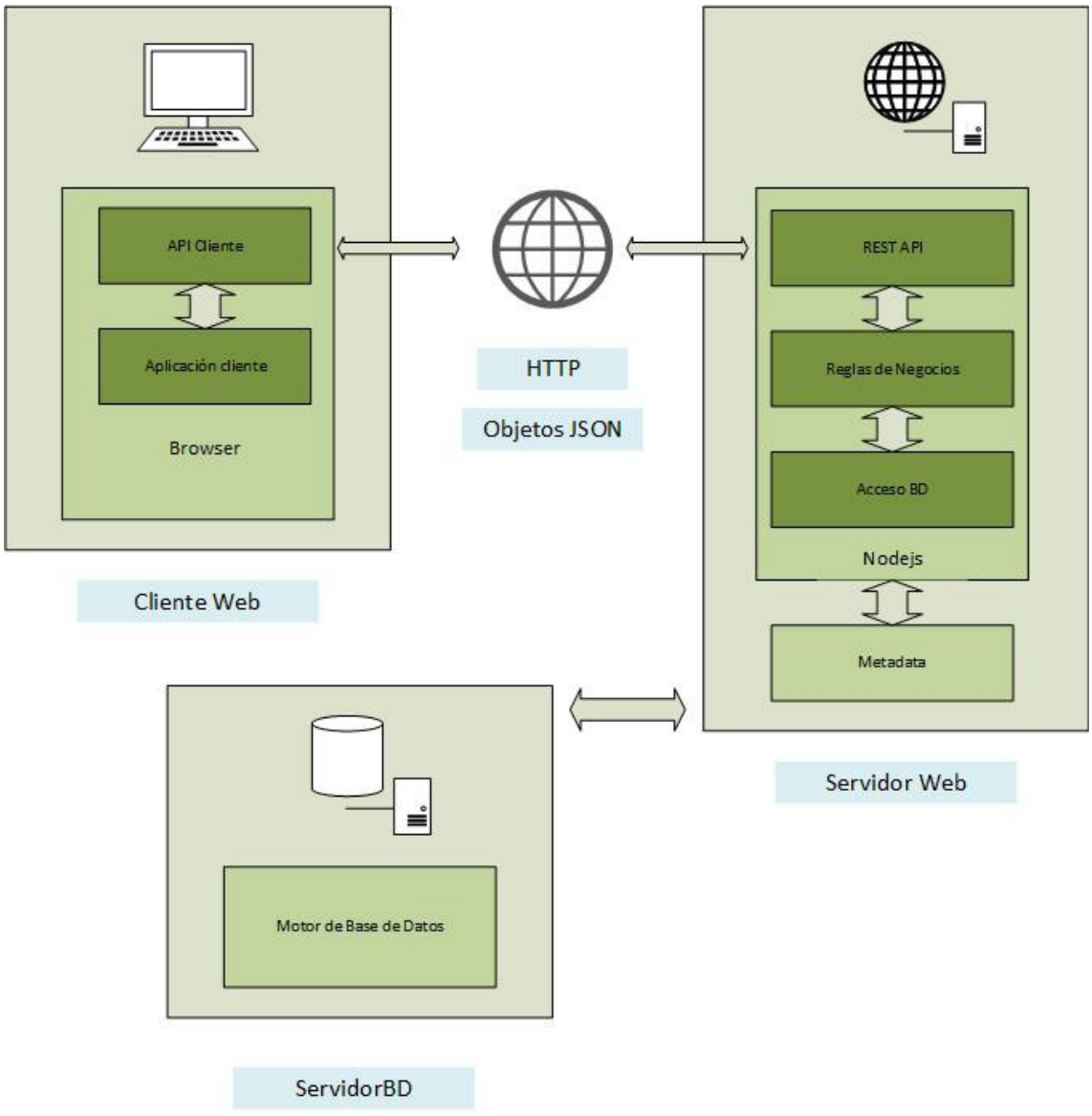
26/08/2018

Nodejs REST API

Arquitectura

Este documento es una presentación de la arquitectura de la REST API.

La implementación de la REST API responderá a una arquitectura de diseño en tres capas expresada en el siguiente diagrama:



El diseño en tres capas es conceptual, lo que significa que en el modelo real podrá implementarse en dos o una capas.

Servidor Web.

El servidor web se implementa sobre Nodejs con un middleware de router Express.

En Express se definirán los “end points” de la API, que en principio serán siete.

La programación del server Nodejs toma datos del modelo de la Base de Datos estructurados como “Recursos” a los cuales se les aplicarán los cuatro verbos REST.

Estos datos están almacenados como “metadata” (archivos JSON) que proveen la información necesaria para procesar la validación de la información recibida por la API, previo a su impacto en la Base de Datos, evitando tanto “round trips” a la base como interpretación de errores SQL.

La API genera en forma dinámica tanto las consultas a la base como las actualizaciones a la misma.

La API maneja las transacciones contra la base y provee control de concurrencia a nivel fila de las tablas de la base (recursos).

La programación del Nodejs es asincrónica con manejo de “promises” y secuenciado con “async/await”.

Las reglas de negocios se deberán programar de acuerdo a este paradigma, devolviendo “promises”.

Estas reglas (o su invocación) deberá insertarse en el código provisto para que tenga la capacidad de ser ejecutado en las distintas etapas del “ciclo de procesamiento”.

El servidor Web expondrá una API en formato JSON para los verbos HTTP que soporten “body” (POST, PUT, DELETE) y un intérprete del “query string” para el verbo GET.

Cliente Web.

El cliente web (fuera del alcance de este proyecto) deberá contemplar el procesamiento asincrónico del consumo de la API interpretando los códigos de retorno y el formato JSON del “response body”.

En el próximo envío de detallarán los “endpoints” y su funcionalidad, el formato de las interfaces y la estructura de la Metadata.

15/09/2018

Nodejs REST API

Metadata (Adenda)

Se agrega el atributo “auto” con valores posibles “Y/N” al arreglo “columns”.

Este atributo es necesario cunado el rol = “P” (Primary key) e indica si la PK se genera en forma automática o debe informarse.

03/09/2018

Nodejs REST API

Metadata

La metadata le brinda la información necesaria a la API para realizar la validación individual y cruzada de los parámetros recibidos desde el cliente.

La API efectúa la validación individual de los parámetros respecto a tipo de dato, longitud y cantidad de decimales, y si el parámetro es requerido o no.

Efectúa también la consistencia de "uniqness" de las columnas, validación de existencia de valores de "foreign key" en insert y update, y chequeo de "primary key" como "foreign key" de otras tablas en delete contemplando, si así se indica, el "cascade delete".

Se denomina recurso a la unidad de afectación de la API. Normalmente un recurso tiene una relación biunívoca con una tabla de la base de datos. La excepción es el caso de una vista (view).

La metadata se compone de un grupo de archivos JSON (dentro de la carpeta "metadata" a partir del "root" de la aplicación). Existirá un archivo JSON por cada recurso. El servidor lee la metadata cada vez que se instancia. Al servidor se le debe indicar explícitamente que recursos debe leer (que archivos de metadata).

Estructura de un archivo de metadata:

{

resource: <string> nombre del recurso,

table: <string> nombre de la tabla en la base de datos,

verbs: <char array> [G,P,U,D] lista de los código de verbos permitidos sobre el recuso,

columns: <object array> array de objetos columna:

[

{

name: <string> nombre de la columna en la tabla,

rol: <string> [P, F, D, V] código de rol de la columna,

cascade: <string> [Y,N] si el rol es “primary key” controla la acción a realizar cuando se elimina un registro : si es “N” dará un error si tiene registros de otras tablas que lo referencian, si es “Y” eliminará, en la mediad de lo posible los registros de las tablas que lo referencian,

type: <string> [S,I,N,T,D,M,F] código de tipo de dato,

length: <integer> longitud de la columna aplicable a tipo de dato string como longitud máxima y para columnas decimales el largo máximo total del número incluyendo punto decimal y decimales (para el resto de los tipos de dato debe ser “null”),

decimals: <integer> cantidad máxima de decimales para el tipo de datos decimal (para otro tipo de datos debe ser “null”),

required: <string> [Y,N] si la columna es mandatoria (“not null”),

unique: <string> [Y,N] si el valor debe ser único en la tabla,

table: <string> para el caso de rol “foreign key” (caso contrario “null”) indica el nombre de la tabla cuya “primary key” es “foreign key”

}

]

}

Tablas de códigos:

Verbos:

Código	Valor
P	POST
G	GET
U	PUT

Roles:

P	Primary Key
F	ForeignKey
D	Dato
V	Version control (tipo integer)

Tipos de Dato:

S	String
I	Integer
N	Decimal
F	Float (o Double)
T	Datetime
D	Date
M	Time
B	Boolean

Al instanciarse, el servidor carga la metadata y valida la existencia de los recursos necesarios para efectuar el control de integridad referencial.

12/09/2018

Nodejs REST API

Instalación y set up

Instalación de Node:

Desde la página de Node (https://nodejs.org/en/download/) descargar la última versión disponible para la plataforma adecuada (para Windows es 8.12) e instalarla.

Creación del proyecto:

- Crear una carpeta con el nombre deseado del proyecto y hacerla corriente.
- Crear un proyecto con npm : " >npm init " completando los prompts a medida que se solicitan. Ninguno de los prompts es relevante para el funcionamiento del proyecto.
- Instalar las dependencias del proyecto (paquetes de node):
    - "npm install mysql --save" (driver de mysql)
    - "npm install express --save" (express router)
    - "npm install path --save" (path resolver)
    - "npm install url --save" (url resolver)
    - "npm install body-parser --save" (body parser)
    - "npm install moment --save" (date & time functions)
- Crear una subcarpeta "metadata"
- Crear una subcarpeta "src"

La estructura de carpetas deberá quedar (donde "root" es el nombre asignado a la carpeta del proyecto):

root

..

node_modules (carpeta)

metadata (carpeta)

src (carpeta)

package.json

Instalación del programa:

El programa consiste de un script principal (e-tangram.js) que deberá copiarse a la carpeta “root”y una serie de módulos javascript que deberán copiarse a la subcarpeta “src”.

La subcarpeta “metadata” contendrá los archivos de metadata descriptores de los recursos.

Instalación de pruebas:

Descomprimir el archivo “metadata.zip” y copiar los .json a la subcarpeta “metadata”.

Deberá crearse una base de datos (schema) con el nombre “testapi”.

Descomprimir el archivo “dbscripts.zip” y ejecutar los scripts desde el MySQL Workbench. El archivo .zip contiene los scripts necesarios para la creación de las tablas de ejemplo que se corresponden con los archivos .json descriptores de la metadata.

El modelo de prueba contiene cinco tablas:

- categorías
- productos
- clientes
- remitos
- remitos_items

las que deberán crearse en este orden para respetar la integridad referencial.

En este modelo se podrán probar los conceptos de unique, required (not null), primary key, foreign key, cascade delete y version (concurrencia), como también los distintos tipos de dato.

Todas las tablas tienen como PK una ID numérica (integer) autoincrementada.

Configuración:

Configuración del script “e-tangram.js”, (el server).

Editar e-tangram.js y modificar los parámetros de conexión: host, database, user y password de acuerdo a las necesidades:

```
var dbConfig = {
    host:"localhost",
    database: "testapi",
    user:"sa",
    password: "mysqladmin",
    multipleStatements: true,
    supportBigNumbers: true,
    dateStrings: true,
    connectionLimit:30
}
```

En producción estas variables deberán guardarse en el “environment”.

El “port” por defecto donde “escucha” Node es 1337 y puede cambiarse a voluntad.

El programa e-tangram.js está ampliamente comentado.

Cliente:

Para las pruebas se sugiere el uso de Postman (https://www.getpostman.com/).

Nota:

4427 6845

12/09/2018

Nodejs REST API

Descripción de la API

End Points:

La API expone 6 end points implementando los cuatro verbos standard HTTP de REST: GET (2), POST (1), PUT (1) y DELETE (2).

Los endpoints se acceden desde el cliente con la siguientes URLs:

http://<dominio.ext>:<port>/api/<recurso>/<id>|<query>

donde:

<dominio.ext>: nombre del dominio donde reside el server Node.

<port>: puerto donde escucha Node (si es 80, se omite).

<recurso>: nombre del recurso afectado por el verbo HTTP. Debe corresponderse con el nombre del recurso en la metadata correspondiente.

<id>: identificación de la instancia del recurso afectado.

<query>: query string de selección de instancias del recurso.

Nota: <id> y <query> son mutuamente excluyentes.

HTTP GET:

Un endpoint para acceso con “id” y otro para acceso con “query”.

Ejemplos:

http://miserver:1337/api/cliente/3

recupera la instancia del recurso con clienteId = 3

http://miserver:1337/api/item/?_remitoItemCantidad ge 12 &remitoItemCantidad le 24

recupera las instancias con cantidad entre 12 y 24

HTTP POST:

Un endpoint donde se “postea” el contenido del “body”:

El “body” será un objeto JSON (stringified) que contendrá las columnas y los valores a insertar.

Ejemplo:

http://miserver:1337/api/producto

body:

{

“productoCategoriaId”: 2,

“productoNombre”: “Envase plástico”,

“productoDescripcion”: ”Descripción de envase plástico”

}

Crea una nueva instancia del recurso producto con los valores indicados de las columnas.

HTTP PUT:

Un endpoint para selección por “id” de la instancia del recurso a modificar.

Ejemplo:

http://miserver:1337/api/cliente/3

body:

{

“clienteNombre”: “Mi Nuevo nombre”,

“clienteVersion”: 325

}

Actualiza el nombre del cliente en la instancia clienteId = 3 si la versión de la instancia del recurso es 325. Caso contrario no la actualiza.

HTTP DELETE:

Un endpoint para selección por “id” (una instancia) y otro para selección con “query” (múltiples instancias).

Ejemplos:

http://miserver:1337/api/cliente/3

elimina la instancia del recurso cliente con clienteId = 3

http://miserver:1337/api/item/?_remitoItemCantidad ge 12 &remitoItemCantidad le 24

elimina las instancias del recurso remito con cantidad entre 12 y 24

Retorno:

El código de retorno HTTP de la API es siempre 200 (OK).

El body del HTTP Response es un objeto JSON (stringified) con dos elementos:

returnset, que contiene información de resultado de la operación, y dataset, que contiene los datos recuperados.

returnset es un arreglo con una sola ocurrencia de un objeto y tiene contenido en las respuestas de las cuatro operaciones.

dataset es un arreglo con tantas ocurrencias como “registros” se hayan recuperado y tiene contenido solo en la respuesta de la operación GET.

```
{
“returnset”: [{
      “RCode”:<integer> (código de retorno de la operación),
      “RTxt”:<string> (texto de retorno de la operación),
      “RId”: <integer> (id de la nueva instancia, solo POST),
      “RSQLErrNo”:<integer> (retorno de MySQL),
      “RSQLErrtxt”:<string> (texto de retorno de MySQL)
      }],
“dataset”: [{}]
}
```

Nota:

La descripción y sintaxis del query y la tabla de códigos de retorno en la próxima entrega.

4427 6845

12/09/2018

Nodejs REST API

Descripción de la API (query string)

El “query string” se utiliza en los verbos GET y DELETE como alternativa de selección de las filas a ser afectadas por cada sentencia.

En ambos verbos también existe la selección por Id del recurso, como se indicó en la parte 4 de este documento.

El “query string” consiste en una o más ocurrencias de pares nombre/valor separados por el signo “&”. El “nombre” se separa del “valor” con el signo “=”.

Estos pares nombre/valor son analizados por la API para fijar los siguientes parámetros:

- Cláusula where.
- Cláusula order by.
- Meta instrucciones include/exclude.
- Limit.
- Offset.

Cláusula where:

Consiste en una cantidad ilimitada de pares nombre/valor:

- El nombre indica el nombre de la columna del recurso referenciado, de acuerdo a su “name” en la metadata.
- El valor consiste de dos elementos:
    - Operador de comparación/inclusión.
    - Valor, necesario para la comparación/inclusión. El valor null no es permitido. Para checkear por null utilizar los valores especiales “isnull” e “isnotnull”

Operadores:

Los operadores son los siguientes:

- “eq”, igualdad.
- “not”, desigualdad.
- “lt”, menor que.
- “le”, menor o igual que.
- “gt”, mayor que.
- “ge”, mayor o igual que.
- “lk”, like, ídem SQL.
- “in”, in a set, ídem SQL.

Valores:

El valor de comparación debe indicarse entre corchetes “[“ “]” y deberá ser del mismo tipo que el definido en la metadata para la columna en cuestión.

Ejemplos:

http://miserver:1337/api/item/? remitoItemCantidad = ge [12] &remitoItemCantidad = le [24]

http://miserver:1337/api/cliente/? clienteNombre = lk [ba%]

http://miserver:1337/api/cliente/? clienteId = in [1,3,4]

http://miserver:1337/api/producto/? productoDescripcion = eq [isnotnull]

selecciona aquellos productos cuya descripción no es “null”.

Cáusula orderby (sólo GET):

Puede especificarse o no. Si se especifica debe ser sólo una vez.

- El nombre es la palabra clave “_orderby”
- El valor es una serie de elementos columna/orden separados por coma.
    - columna es el nombre de la columna en la metadata.
    - orden, ordenamiento ascendente “A” o descendente “D”, default “A”

La columna y el orden deben estar separados por “espacio”.

Ejemplos:

http://miserver:1337/api/cliente/? _clienteNombre = lk [ba%] & _orderby = clienteNombre A, clienteId D

ascendente por nombre del cliente y ,por igualdad, descendente por Id del cliente

http://miserver:1337/api/remito/? _orderby = remitoFecha

ascendente por fecha de remito (default Asc).

Meta include/exclude (sólo GET):

Pueden especificarse o no. Son excluyentes.

Son utilizados para limitar las columnas seleccionadas.

Si no se informan se recuperan todas las columnas del recurso (tabla).

Utilizar “_include” para indicar que columnas se seleccionan.

Utilizar “_exclude” para indicar que columnas no se seleccionan, a partir de todas las columnas del recurso (tabla).

- El nombre es la palabra clave(“_include” / ”_exclude”).
- El valor es una lista de los nombres de las columnas (de acuerdo a la metadata) separados por coma, que se incluyen / excluyen en la selección.

Ejemplos:

http://miserver:1337/api/cliente/? _exclude = clienteVersion, clienteId

excluye clienteVersion y cliente Id de la selección.

http://miserver:1337/api/cliente/? _include = clienteNombre

selecciona solamente la columna clienteNombre

Meta limit (GET):

Envía el valor de la cláusula limit a la sentencia SQL.

- El nombre es la palabra clave “_limit”.

- El valor es un número entero que indica la cantidad de filas a recuperar.

Ejemplo:

http://miserver:1337/api/remito/? remitoFecha = ge [2018-09-01] & _limit = 20

selecciona las 20 primeras líneas de remitos que cumplen con la condición de fecha mayor que el 31/08/2018.

Meta offset (GET):

Envía el valor de la cláusula offset a la sentencia SQL.

- El nombre es la palabra clave "_offset".
- El valor es un número entero que indica a partir de que fila se va a recuperar.

Ejemplo:

http://miserver:1337/api/remito/? remitoFecha = ge [2018-09-01] & _limit = 20 & _offset = 20

selecciona 20 primeras líneas de remitos que cumplen con la condición de fecha mayor que el 31/08/2018, a partir del registro 21.

Se utilizan para paginados.

17/09/2018

Nodejs REST API

Descripción de la API (códigos de retorno)

Como se indicó en la parte 4, la API devuelve un response body con la siguiente estructura:

{

“returnset”: [{

“RCode”:<integer> (código de retorno de la operación),

“RTxt”:<string> (texto de retorno de la operación),

“RId”: <integer> (id de la nueva instancia, solo POST),

“RSQLErrNo”:<integer> (retorno de MySQL),

“RSQLErrtxt”:<string> (texto de retorno de MySQL)

}],

“dataset”: [{}]

}

El código de retorno HTTP de la API es siempre 200 (OK).

El body del HTTP Response es un objeto JSON (stringified) con dos elementos:

returnset, que contiene información de resultado de la operación, y dataset, que contiene los datos recuperados.

returnset es un arreglo con una sola ocurrencia de un objeto y tiene contenido en las respuestas de las cuatro operaciones.

dataset es un arreglo con tantas ocurrencias como “registros” se hayan recuperado y tiene contenido solo en la respuesta de la operación GET.

RCode: return code, es el resultado de la operación, puede tener los siguientes valores:

- Éxito, se corresponde con el valor 1 (uno). (RTxt: “OK”)

- Error MySQL, se corresponde con el valor 0 (cero). (Rtxt: “ErrorMySQL”). En este caso los campos RSQLErrNo y RSQLErrTExt contienen el número de error y el texto del mismo tal cual fueron devueltos por MySQL.

- RCode < 0. En este caso es un error específico de la API. RTxt continene la explicación del mismo. Los valores posibles se agrupan en 3 rangos distintos:

    - -1xxx, corresponden a errores de sintaxis y validación de los parámetros enviados contra la metadata.

    - -2xxx, corresponden a errores de la lógica de actualización de la base.

    - -5xxx, corresponden a errores internos inesperados.

    La lista de los errores se detalla a continuación.

- RId: es la Id asignada por la base cuando ésta es “auto: Y” y se produce una inserción.

Tabla de Códigos:

Return	Text
-1000	JSON Malformado
-1001	Recurso inválido
-1002	Verbo no soportado en el recurso
-1003	Body vacío
-1004	Nombre de miembro inválido en body
-1005	PK no informada, no auto
-1006	Version no informada
-1007	Columna requerida no informada
-1008	Tabla referenciada (FK), no encontrada en metadata
-1009	Tabla sin PK
-1010	Pk referenciada es de distinto tipo que la Fk referenciante
-1011	Valor inválido para columna tipo boolean
-1012	Valor inválido para columna tipo date/time/datetime
-1013	Valor no numérico para columna numérica
-1014	Valor no entero para columna entera
-1015	Cantidad de decimales excedida para columna decimal
-1016	Longitud de columna string excedida
-1017	_include y _exclude son excluyentes
-1018	_include no permitido en DELETE
-1019	_include vacío
-1020	Nombre de columna inválido
-1021	_exclude vacío
-1022	_orderby no permitido en DELETE

-1023	_orderby vacío
-1024	_orderby, error de sintaxis
-1025	_orderby, tipo de orden inválido
-1026	_orderby, columna no seleccionada
-1027	Corchetes desbalanceados
-1028	Los corchetes no se pueden anidar
-1029	Valor de query no informado
-1030	Operador de query inválido
-1031	Null no puede ser usado como parámetro, utilizar 'isnull' o 'isnotnull'
-1032	Tipo de dato inválido
-1033	Tipo de dato inválido en lista
-1034	_offset no permitido en DELETE
-1035	_offset debe ser numérico
-1036	_offset inválido
-1037	_limit no permitido en DELETE
-1038	_limit debe ser numérico
-1039	_limit inválido
-2001	Valor duplicado para columna unique
-2002	Valor de FK no encontrado
-2003	No encontrado
-2004	Versiones distintas
-2005	PK con dependencias como FK
-5001	Error interno buscando columna en metadata
-5002	Tabla referenciada no encontrada en metadata
-5003	PK no encontrada en metadata
-5004	Version no encontrada en metadata
-5005	Columna no encontrada en metadata

17/09/2018

Nodejs REST API

Descripción de la API (Concurrencia)

Concurrencia:

El control de concurrencia en la operación update se realiza con el mantenimiento de una columna de “versión” de la fila. Al crearse una nueva fila el valor de su versión es cero.

La estrategia consiste en leer el registro correspondiente con su versión (ordinal) y al ejecutar el update se verifica que la versión sea la misma que la leída.

Si no es la misma la API devuelve el código -2004 (versiones distintas).

Si es la misma se produce el update y la versión es incrementada en uno.

La columna que mantiene la versión en la tabla debe decribirse con rol : “V” en la metadata.

18/09/2018

Nodejs REST API

Uso de vistas (views)

La API interpreta (como SQL) a las vistas como tablas de “solo lectura”.

Para su uso:

1. Crear la vista con su nombre.
2. Crear un archivo de metadata con la misma estructura que un archivo metadata de tabla asignándole al atributo “table” el nombre de la vista y describiendo las columnas que serán seleccionadas por defecto (esto luego podrá ser afectado por “_exclude /_include”). En el arreglo “verbs” colocar solo GET (G). Ej: “verbs”: [“G”]
3. Agregar el archivo de metadata con su sentencia “required” correspondiente en “e-tangram.js” (línea 36 aprox.)
4. Reiniciar el server.
5. Utilizar el GET con selección, ordenamiento, etc.
6. Se adjunta un ejemplo sobre la base de prueba.

20/09/2018

Nodejs REST API

JWT (JSON Web Token)

Atención:

La instalación de esta versión requerirá cambios en los requests a la API.

JSON Web Token provee un método sencillo de intercambio de requests con la API una vez efectuado el Log In.

La implementación es un “middleware” que se procesa en cada request.

Mecanismo:

1. El cliente envía un log in (usuario/password) en el “body” de un POST.
2. El server recibe, valida y genera un JWT utilizando un “payload” y la clave “secreta”. Este token será válido por un determinado tiempo, expresado en segundos.
3. El server envía el token al cliente.
4. Para todo request el cliente debe enviar el token recibido en el header de dicho request con key “x-access-token” y value <token>.
5. El server recibe el request, verifica la presencia y validez del JWT y decodifica el “payload”.
6. En caso de ser inválido, el server retorna un código de error (-6002 / -6003).
7. Si es exitoso el middleware inserta el “payload” decodificado en el request y continúa el procesamiento del mismo(next()).

Implementación:

Deberá instalarse los packages:

npm install jsonwebtoken --save

npm install bcrypt --save

Se ha agregado el endpoint http://<miserver>/login para el verbo POST.

El mismo espera un JSON

{

“usuario”: <string>

“pass”: <string>

}

El endpoint valida el contenido contra un arreglos de objetos (usuarios, línea 33) y si el par usuario/password es válido genera un JWT con payload = usuarioId y clave secreta JWTSecret (línea 30), devolviendo al cliente el jwt en el dataset.

Los request sucesivos a cualquier endpoint serán procesados por el “middleware” (rest_token.js). De ser correcto, insertará el valor del payload (usuarioId) en el request y continuará la ejecución de acuerdo a lo programado en cada endpoint.

En este ejemplo se puede ver como, a partir de la validación y del encriptado del payload, puede obtenerse el valor de usuarioId sin necesidad de acceder a la DB.

Mensajes:

Se agregan los siguientes códigos de retorno:

Código	Texto
-6001	Usuario/Password inválido
-6002	Debe proveerse un token
-6003	Token inválido

26/09/2018

Nodejs REST API

Stored Procedures Bridge

Objetivo:

Se agrega una nueva funcionalidad a la API que consiste en la capacidad solicitar ejecuciones de Stored Procedures de la base de datos.

Implementación:

Se ha creado un nuevo endpoint para procesar estos “request”:

http://<miserver>/api/proc/<nombre>

El verbo a ejecutar es POST.

Los parámetros de ejecución se envían en el body.

<nombre>: nombre del “servicio” definido en la metadata.

Metadata:

Como los “recursos” aplicables a la REST API, los stored procedures también requieren metadata para efectuar la validación de los parámetros recibidos y el ordenamiento de los mismos en la invocación al SP (recordemos que los parámetros de un llamado a un SP son posicionales).

La diferencia importante con la metadata de “recursos” es que los parámetros no necesitan corresponderse con nombres de columnas de tablas de la base de datos.

Cada archivo de metadata para stored procedures puede contener “n” definiciones de parámetros, es decir metadata para más de un SP. Cada archivo comprende “lógicamente” a los procedimientos que afecten a un recurso en un sentido amplio.

La metadata para SPs es un archivo JSON con el siguiente formato:

```
{
    "resource”:<string> (nombre del recurso)
    "sps":<array> (de definiciones de SP)
      [
        {
            "servicio": <string> (nombre “externo” del SP),
            "type": <string> “C”,”R”,”U”, “D” (tipo de operación del SP)
            "sp": <string> (nombre “interno”, en la base, del SP),
            "params": <array> (de parámetros ordenados, para ejecución)
            [
                {
                  "name":<string> (nombre del parámetro),
                  "required": <string> “Y”/”N” (si es requerido o no)
                }, …
            ]
        }, …
    ],
    "columns":<arreglo> (de descriptores de columnas, parámetros)
      [
        {
            "name": <string> (nombre del parámetro)
            "type": <string> (tipo de dato, idem tablas),
            "length": <int> (longitud),
            "decimals": <int> (decimales
        },…
      ]
}
```

El arreglo “columns” permite las definiciones de tipo y longitud para los parámetros, mientras que el arreglo “params” indica, para cada stored procedure, que columna debe incluirse, en qué orden y la obligatoriedad de informarlo en el request.

Se adjunta un ejemplo de metadata para cuatro SPs sobre la tabla “producto” (producto_sp.json).

Los archivos de metadata de definición de SPs residirán en la misma carpeta (.\metadata) que los archivos de metadata de tablas.

Carga de Metadata:

Esta versión cambia el procedimiento de carga de la metadata, eliminando la necesidad de tocar el código del server cada vez que se agrega o quita un archivo de metadata.

Para esto se crea un catálogo de metadata en la misma carpeta (.\metadata), con el nombre “meta_catalogo.json”.

Este archivo de catálogo consiste de un arreglo bajo el nombre “catalog” donde cada elemento es un objeto con dos miembros:

- name: nombre del archivo de metadata a cargar (sin extensión)
- type: tipo de metadata (“T”:tabla, “V”:vista(view) y “S”:stored procedure)

Una vez creada la metadata para una tabla, vista o grupo de SPs, deberácrearse la entrada en el catálogo para que sea cargado en el server, cada vez que éste se reinicie.

Se envía el ejemplo actualizado para el conjunto de ejemplos.

De los Stored Procedures:

Los SPs deberán construirse con una estructura de códigos de retorno consistentes con el esquema general de retorno de la API, tanto para que la misma sepa en qué orden retornar estos códigos como para que la aplicación cliente pueda dialogar de una forma independiente de los procesos subyacentes.

Se adjuntan cuatro Stored Procedures (Insert, Delete, Update y Retrieve) para la tabla “productos” como ejemplo de estructura y retornos.

Estos SPs son coincidentes con el archivo de definición de los mismos (producto_sp.json)