Clase del día - 30/04/2021

La clase de hoy explicaremos cómo funciona el servicio estilo REST que creamos la clase anterior.

En la plataforma se publicó los siguientes archivos:

* [**Servicio.zip**](https://m4gm.com/moodle/mod/resource/view.php?id=1842)contiene el código Java y archivos de configuración de un servicio web estilo REST.
* [**prueba.html**](https://m4gm.com/moodle/mod/resource/view.php?id=1843) contiene una aplicación web que invoca el servicio web mediante Javascript.
* [**WSClient.js**](https://m4gm.com/moodle/mod/resource/view.php?id=1844) funciones para invocar el servicio web mediante AJAX
* [**usuario\_sin\_foto.png**](https://m4gm.com/moodle/mod/resource/view.php?id=1845)imagen que se despliega cuando el usuario no tiene foto.

El archivo [**Servicio.zip**](https://m4gm.com/moodle/mod/resource/view.php?id=1842) contiene los siguientes directorios:

* META-INF
* negocio
* WEB-INF

El directorio META-INF contiene el archivo **context.xml**, en el cual se configura lo siguiente:

El atributo **name**de la etiqueta **Resource**, define el nombre del datasource, en este caso el datasource se llama "jdbc/datasource\_Servicio".

Un datasource permite configurar las conexiones que realiza el servicio web, sin tener que escribir estos parámetros de configuración en el código (por ejemplo el nombre y contraseña del usuario de la base de datos).

El atributo **url**define el nombre de la base de datos, en este caso la base de datos se llama "servicio\_web", el atributo **username**define el nombre del usuario de la base de datos y el atributo **password**define la contraseña del usuario.

El directorio WEB-INF contiene el directorio **classes**y el archivo **web.xml**.

El archivo **web.xml** configura lo siguiente:

La etiqueta **load-on-startup** igual a 1 indica que el servicio web se debe cargar cuando inicie el servidor de aplicaciones Tomcat.

La etiqueta **url-pattern** indica la ruta del servicio web. La URL del servicio web se explicará más adelante.

La etiqueta **resource-ref**, indica el nombre del datasource y la etiqueta **res-type** indica el tipo de recurso javax.sql.DataSource.

En el directorio **classes**se colocarán las clases compiladas del servicio web (archivos .class). En este caso el directorio incluye un subdirectorio llamado "negocio", debido a que las clases del servicio web se agrupan en un paquete llamado "negocio".

Al mismo nivel del los directorios META-INF y WEB-INF se encuentra un directorio llamado **negocio**dónde se puede encontrar los archivos que contienen el código fuente del servicio web.

Los archivos incluidos en el directorio **negocio** son los siguientes:

**AdaptadorGsonBase64.java**

La clase AdaptadorGsonBase64 permite modificar la forma en que GSON serializa los campos con tipo byte[].

Por omisión GSON convierte un campo de tipo byte[] en una lista de números separados por comas, lo cual ocupa mucho espacio (y tiempo en la comunicación).

La clase AdaptadorGsonBase64 permite convertir los campos de tipo byte[] a base 64, lo cual produce un texto más compacto.

Notar que jax-rs reemplaza cada "+" por espacio, sin embargo el decodificador Base64 no reconoce el espacio, por lo que es necesario reemplazar los espacios por "+".

**Error.java**

La clase Error va a permitir regresar un mensaje de error dentro de un objeto.

**Foto.java**

La clase Foto encapsula un arreglo de bytes y una clave numérica. El servicio web utiliza esta clase para enviar y recibir imágenes.

**Usuario.java**

La clase Usuario encapsula los datos del usuario. El servicio web utiliza esta clase para recibir los datos del usuario como un objeto.

Es muy importante notar que la anotación @FormParam requiere un método que convierta una String a objeto de tipo Usuario. En este caso se implementa el método **valueOf**para este propósito.

**Servicio.java**

La clase Servicio implementa los métodos del servicio web.

**URL del servicio web**

La URL del servicio web se compone de cuatro elementos:

1. Dominio y el puerto, por ejemplo **localhost:8080**
2. Nombre del archivo .war, en este caso **Servicio**
3. Ruta definida en la etiqueta url-pattern en el archivo web.xml, en este caso la ruta: **rest**
4. Ruta definida en la anotación @Path de la clase Servicio, en este caso se define: **ws**

En este caso la URL completa sería: http://localhost:8080/Servicio/rest/ws

**Pool de conexiones**

Con frecuencia cuándo un servicio web recibe una petición (requerimiento), abre una conexión a la base de datos. Sin embargo realizar una conexión a la base de datos es lento. Por tanto, es conveniente que los servicio web utilicen un pool de conexiones mediante un datasource.

Como se dijo anteriormente, un datasource permite configurar los parámetros de conexión a la base de datos fuera del código del servicio web. No es buena práctica escribir el nombre de usuario de base de datos y la contraseña en el código del servicio web, ya que habría que editar y recompilar el código cada vez que se cambie la contraseña del usuario.

Así mismo, el uso de datasource y pool de conexiones permite establecer un número de conexiones a la base de datos y mantenerlas abiertas siempre. Cada vez que un método del servicio web requiera conectarse a la base de datos, solicita una conexión disponible en el pool.

Cuándo el método termina deberá liberar la conexión invocando el método "close", no obstante, este método no cierra la conexión sino que la regresa al pool de conexiones, para su posterior re-uso.

Es muy importante considerar que cualquier configuración que el método haga a la conexión (por ejemplo activar o desactivar el autocommit) quedará establecida en la conexión, de manera que si otro método re-usa la conexión, esta conexión tendrá una configuración previa.

Debido a lo anterior, un método que obtiene una conexión del pool de conexiones deberá re-configurar la conexión de acuerdo a sus propósitos, y no suponer que la conexión tiene una configuración determinada.

En el caso de la clase Servicio, el pool de conexiones se crea en la inicialización estática de la clase. Notar que el datasource se identifica con el nombre que se le dio en el archivo context.xml

**El método POST**

Como vimos en la clase anterior, un servicio web de estilo REST utiliza los métodos GET, POST, DELETE y PUT para realizar operaciones determinadas. En este caso, el servicio web que implementamos se aparta un poco del estándar ya que TODOS los métodos web utilizan el método POST, por esta razón todos los métodos de la clase Servicio incluyen la anotación **@POST**.

**El servicio web produce JSON**

El el código de la clase Servicio puede observarse que cada método incluye una anotación **@Produces**(MediaType.APPLICATION\_JSON). Esto significa que el método regresará los resultados en formato JSON.

Para poder serializar y des-serializar JSON utilizaremos GSON (la implementación de JSON de Google), se declara una variable estática de tipo Gson y se inicializa con una instancia creada mediante el método create de la clase GsonBuilder:

static Gson j = new GsonBuilder()  
        .registerTypeAdapter(byte[].class,new AdaptadorGsonBase64())  
        .setDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS")  
        .create();

Notar que en este caso se configura la instancia de Gson para utilizar el adaptador AdaptadorGsonBase64 para serializar y des-serializar arreglos de bytes, y se define el formato de fecha como ISO8601.

**El servicio web consume URL encoded**

Cada método incluye la anotación **@Consumes**(MediaType.APPLICATION\_FORM\_URLENCODED) la cual indica que los parámetros del método se recibirán codificados como URL. Esta codificación reemplaza algunos caracteres especiales como ampersam &, igual =, espacio, etc. los cuales son caracteres reservados cuando se construye una URL.

**El nombre del método web**

Cada método incluye la anotación **@Path**, esta anotación define el nombre del método web el cual puede ser diferente al nombre del método. Por ejemplo, el método "alta\_usuario" se invocará como el método web "alta".

**Los nombres de los parámetros del método web**

Para indicar el nombre de cada parámetro del método web se utiliza la anotación **@FormParam**.

El nombre del parámetro de método web puede ser diferente al nombre del parámetro del método Java.

**El método web regresa Response**

El método web regresa una instancia de la clase Response. Básicamente hay tres tipos de respuesta:

**El método no regresa resultados**. El método termina sin error pero no regresa resultados al cliente (por ejemplo el método "alta\_usuario). El método deberá regresar al cliente la siguiente instancia de la clase Response: Response.ok().build() esta instancia incluye el código HTTP 200 indicando que no hubo error.

**El método regresa resultados**. El método termina sin error y regresa algún resultado al cliente. Debido a que se definió en la anotación @Produces que el método regresa JSON, el método deberá regresar al cliente la siguiente instancia de la clase Response: Response.ok().entity(j.toJson(r)).build() en este caso r sería un objeto o un arreglo, el cual será serializado mediante GSON. El cliente recibe el código HTTP 200 indicando que no hubo error.

**El método regresa un mensaje de error**. El método termina con error. Para regresar un mensaje de error al cliente se utiliza la clase Error. Debido a que se definió en la anotación @Produces que el método regresa JSON, el método deberá regresar al cliente la siguiente instancia de la clase Response: Response.status(400).entity(j.toJson(new Error("Mensaje de error"))).build() el mensaje de error será de acuerdo al error que se produjo. El cliente recibe el código HTTP 400 indicando que hubo error. Adicionalmente, si así se desea se podría agregar a la clase Error un código numérico.

**El método web debe cerrar la conexión a la base de datos**

Un error frecuente de los programadores es que no cierran las conexiones a la base de datos. En general, los DBMS cuentan con un número limitado de conexiones. Si una aplicación no cierra las conexiones que no utiliza, se agotarán las conexiones en el DBMS y el sistema dejará de funcionar.

Para evitar esta situación, es necesario que el programador cierre las conexiones a la base de datos, archivos, sockets y cualquier otro recurso que haya abierto el método web. Esto debe hacerse mediante un cuidadoso diseño del manejo de excepciones mediante try-finally-catch.

**El archivo**[**WSClient.js**](https://m4gm.com/moodle/mod/resource/view.php?id=1844)

Este archivo contiene código Javascript utilizado por [prueba.html](https://m4gm.com/moodle/mod/resource/view.php?id=1843) para consumir el servicio web mediante AJAX (XMLHttpRequest).

La función WSClient() incluye la función "post" la cual recibe los siguientes parámetros: el nombre del método web a invocar, un arreglo con los argumento del método web y una función callback que se invocará cuándo el método web termine.

**El archivo**[**prueba.html**](https://m4gm.com/moodle/mod/resource/view.php?id=1843)

Este archivo contiene una Single-Page Application (SPA) escrita en HTML-Javascrit, la cual invoca los métodos web del servicio web "Servicio".

Actividades individuales a realizar

1. Agregar un método web llamado "borrar\_usuarios" al archivo Servicio.java, este método deberá borrar todos los usuarios de la tabla "usuarios". El método deberá regresar un mensaje de error si no pudo borrar los usuarios.
2. Agregar un botón al archivo [prueba.html](https://m4gm.com/moodle/mod/resource/view.php?id=1843). Al dar clic a este botón se deberá invocar el método web "borrar\_usuarios".
3. Compilar el archivo Servicio.java
4. Construir el archivo Servicio.war
5. Borrar el archivo Servicio.war y el subdirectorio Servicio los cuales se encuentran en el directorio webapps de Tomcat.
6. Copiar el archivo Servicio.war al directorio webapps de Tomcat.
7. Utilizando un navegador probar el nuevo botón que invoca el método web "borrar\_usuarios".