Sistemas Distribuidos e Internet
Servicios Web Rest (Cliente Java)
Sesión- 10.1

Curso 2017/ 2018

Cliente Java

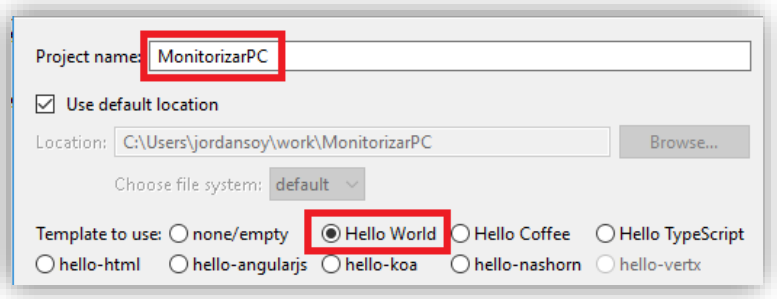
Vamos a desarrollar una aplicación Java que consuma un servicio web REST desarrollado con Node JS.

Servicio Web (MonitorizarPC)

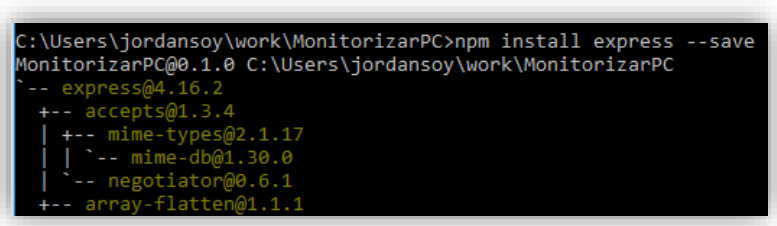
En primer lugar vamos a implementar un pequeño servicio Web en Node.js .

Creamos un nuevo proyecto **File -> New -> Node.js Project**

El proyecto se llamará de **MonitorizarPC** , nos basamos en la plantilla **Hello World**



Una vez creado, abrimos una consola de comandos CMD en el directorio raíz del proyecto e instalamos el módulo express con el comando **npm install express --save**



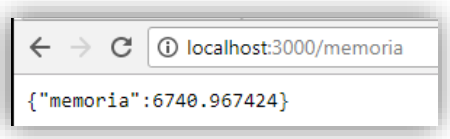
Modificamos el contenido del fichero **hello-world-server.js** , creamos una aplicación express y una función que responda a **GET /memoria** , retornando un JSON { “memoria” : 4783.929 } con la memoria libre en el equipo.

Para obtener la memoria de nuestro equipo usamos el módulo **os**, no hace falta descargarlo ya que se trata de un módulo nativo. Este módulo nos da a acceso a varias funciones del sistema. https://nodejs.org/api/os.html

***Importante:** en lugar de responder el consumo de memoria inmediatamente posponemos la respuesta 10 segundos, utilizando para ello un **setTimeout**. Vamos a utilizar esta pausa para emular un servicio computacionalmente costoso, por ejemplo, que maneje muchos datos.

```
var express = require('express');
var app = express();
var os = require('os');
var puerto = 3000;

app.get('/memoria', function(req, res){
    setTimeout(function() { // Espera de 10 segundos

        console.log(os.freemem());
        var memoriaLibre = os.freemem() / 1000000;  //pasar a MB
        res.status(200);
        res.json({
            memoria : memoriaLibre
        });

    }, 10000);
});

app.listen(puerto,  function() {
   console.log("Servidor listo "+puerto);
});
```

Ejecutamos la aplicación y accedemos a http://localhost:3000/memoria deberíamos obtener una respuesta una vez hayamos esperado unos 10 segundos.

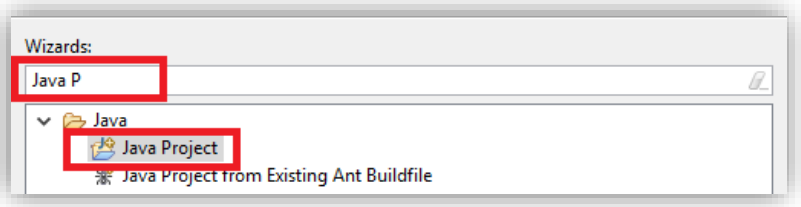


Cliente Java

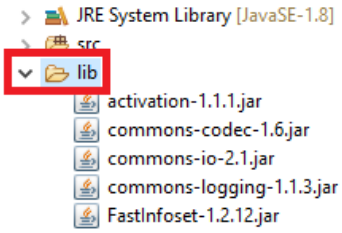
El cliente va a obtener el consumo de memoria de un equipo y lo mostrará en una ventana. Para la parte gráfica de la aplicación utilizaremos **Swing** y para realizar las llamadas al servicio web nos basaremos en la librería **JAX-RS** (Existen otras muchas librerías para invocar servicios web REST desde Java).

Abrimos el STS (u otra versión de eclipse) y creamos un nuevo proyecto **Java** al que llamaremos **ClienteMonitorizar**

File -> New -> Other . Después seleccionamos un proyecto de tipo **Java Project**



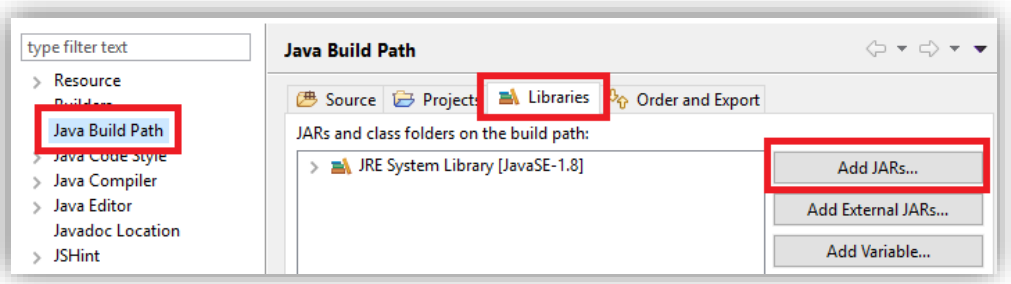
Creamos un nuevo Folder en el proyecto y lo llamamos **lib**, copiamos en el directorio las librerías que nos descargamos del campus virtual. Entre ellas se encuentra **JAX-RS** y otras utilidades.



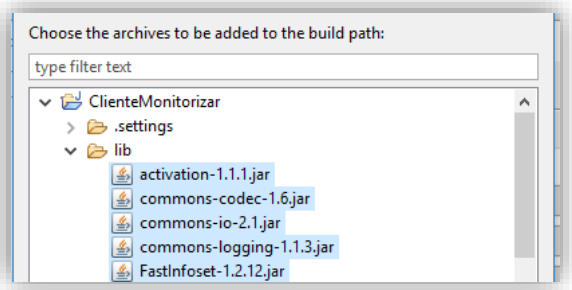
Agregamos al **build path** del proyecto todas las librerías de la carpeta **/lib**

Botón derecho sobre el **proyecto -> properties.**

Seleccionamos en el menú de la izquierda la sección **Java Build Path,** después la pestaña **Libraries** y pulsamos el botón **Add JARs…**



Seleccionamos <u>todos</u> los **.jar** de la carpeta **/lib**



Finalmente pulsamos en el botón **Apply and Close**.

Creamos una nueva clase Java llamada **Ventana** en el paquete **com.sdi**

Implementamos una ventana con **swing** en la que definimos:

- Un Frame y un Panel
- Un botón actualizar, que posteriormente una vez pulsado realizará la llamada al servicio Web para consultar el consumo de memoria actual
- Un botón apagar, que muestra un mensaje por pantalla (<u>no tendrá funcionalidad real es solo para comprobar si el interfaz de la aplicación responde</u>)
- Un Label, en el que se mostrará la memoria libre actual.

Incluimos la función **main** en la cual creamos una instancia de la **Ventana**.

```
public class Ventana {
	JFrame frame;
	JPanel panel;
	JButton botonActualizar;
	JButton botonApagar;
	public JLabel textoMemoria;
	int peticiones = 0;

	public static void main(String[] args) throws InterruptedException {
		new Ventana();
	}

	public Ventana() {
		// Frame
		frame = new JFrame("Aplicación Monitorización");
		frame.setSize(500, 200);
		frame.setLocationRelativeTo(null);

		// Panel
		panel = new JPanel();
		panel.setBorder(new EmptyBorder(10, 10, 10, 10));
		panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
		frame.add(panel);

		// Botón Actualizar
		botonActualizar = new JButton("Actualizar Memoria");
		botonActualizar.setBorder(new EmptyBorder(10, 10, 10, 10));
		botonActualizar.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent arg0) {

			}
		});
		panel.add(botonActualizar);

		// Botón Actualizar
		botonApagar = new JButton("Apagar Equipo");
		botonApagar.setBorder(new EmptyBorder(10, 10, 10, 10));
		botonApagar.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent arg0) {
				JOptionPane.showMessageDialog(frame, "Enviado apagar!");
			}
		});
		panel.add(botonApagar);

		// Texto memoria
		textoMemoria = new JLabel();
		textoMemoria.setBorder(new EmptyBorder(10, 10, 10, 10));
		textoMemoria.setText("Memoria libre:");
		panel.add(textoMemoria);

		// Propiedades visibilidad frame
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setVisible(true);
	}
}
```

Si ejecutamos la aplicación deberíamos ver la ventana.

Nos falta por implementar la lógica que se ejecuta al pulsar el **botonActualizar**, deberíamos:

- Hacer una invocación al servicio Rest **GET / http://localhost:3000/memoria .**
- Procesar el objeto JSON recibido {"memoria":6740.967424}
- Mostrar la memoria consumida en el componente **textoMemoria**

La API JAX-RX está pensada para consumir servicios REST gestionando las peticiones HTTP de forma relativamente directa. Para realizar una petición debemos:

- Crear un objeto cliente (**ClientBuilder.newClient())**
- Especificar la URL a la que accedemos (**.target(…))**
- A partir de la URL, crear una petición **(.request())**
- Añadir los tipos MIME en los que deseamos recibir el contenido **(.accept(…))**
- Ejecutar el tipo de petición que necesitemos: **GET, PUT, POST, DELETE (.get())**
- Recoger los datos en el formato Java que necesitemos **(.readEntity(…))**

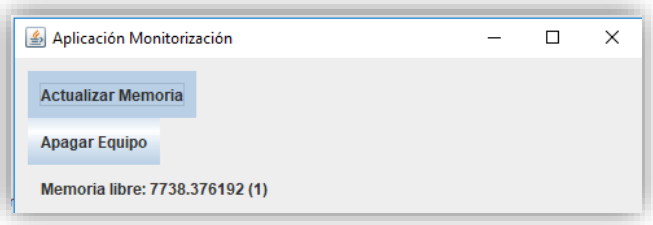
Completamos los parámetros para la petición a nuestro servicio. En el caso del parámetro especificado en la función **readEntity**() utilizaremos un tipo de objeto **ObjectNode**.

ObjectNode es un objeto genérico de la librería **codehaus** que nos sirve para almacenar objetos JSON de cualquier tipo. (También podríamos haber creado una nueva clase **RespuestaMemoria** con un atributo **memoria y funciones (get/set)** la cual hubiese servido para que se parseara automáticamente el JSON a un objeto de ese tipo)

```
@Override
public void actionPerformed(ActionEvent arg0) {
    peticiones++;

    ObjectNode respuestaJSON;
    respuestaJSON = ClientBuilder.newClient()
            .target("http://localhost:3000/memoria")
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .get()
            .readEntity(ObjectNode.class);

    String memoria = respuestaJSON.get("memoria").toString();
    textoMemoria.setText("Memoria libre: "+memoria +" ("+peticiones+")");
}
```

Ejecutamos la aplicación y comprobamos lo que sucede al pulsar el botón **Actualizar Memoria**.

Durante los 10 segundo que tarda en completarse la petición la aplicación está completamente bloqueada, lo sabemos porque no podemos pulsar el botón **Apagar Equipo.**



En la aplicación anterior hemos lanzado la petición al servicio desde el hilo principal. Este hilo es el que se encarga de actualizar la interfaz gráfica de la aplicación, mientras está esperando por la respuesta toda la interfaz se mantiene bloqueada.

Nunca deberíamos realizar llamadas a servicios web ni operaciones de lógica de negocio desde el hilo principal.

Petición desde hilo

Modificaremos la implementación anterior para que la petición se realice desde un nuevo hilo. Creamos la nueva clase **ActualizarMemoriaThread**, en esta clase implementamos un hilo (clase que extiende hereda de Thread).

Creamos un constructor que reciba una referencia a **Ventana** y una función **run()** donde se haga la misma invocación al servicio que antes realizábamos desde el botón.

Una vez obtenido el consumo de memoria vamos a invocar al método **actualizarMemoria(memoria);** del objeto **Ventana** está función aun no ha sido implementada, se va a encargar de actualizar la memoria que se muestra por pantalla.

```
public class ActualizarMemoriaThread extends Thread {
    Ventana ventana;

    public ActualizarMemoriaThread(Ventana ventana) {
        this.ventana = ventana;
    }

    public void run(){
        ObjectNode respuestaJSON;
        respuestaJSON = ClientBuilder.newClient()
                .target("http://localhost:3000/memoria")
                .request()
                .accept(MediaType.APPLICATION_JSON)
                .get()
                .readEntity(ObjectNode.class);

        String memoria = respuestaJSON.get("memoria").toString();
        ventana.actualizarMemoria(memoria);
     }

}
```

Volvemos a la clase **Ventana**. Sustituimos el código que se ejecuta al pulsar el **botonActualizar**. Crearemos una instancia del hilo **ActualizarMemoriaThread** y lo lanzaremos (función **start()**)

```
@Override
public void actionPerformed(ActionEvent arg0) {
    peticiones++;
    ActualizarMemoriaThread hilo = new ActualizarMemoriaThread(Ventana.this);
    hilo.start();
}
```

Solo nos falta implementar el método **actualizarMemoria(memoria)**, en el que se actualiza la etiqueta de texto **textoMemoria**. La implementación más sencilla podría ser la siguiente:

```
public  void actualizarMemoria(String memoria) {
   textoMemoria.setText("Memoria libre: "+memoria +" ("+peticiones+")");
}
```

Aunque este código puede que en ocasiones funcione correctamente, si lo utilizamos estaríamos cayendo en un error. **ActualizarMemoria()** va a ser ejecutado desde un hilo

secundario, no desde el hilo principal que recordamos que es el encargado de actualizar la Interfaz de usuario, si dos hilos modificaran simultáneamente el mismo elemento de interfaz se produciría una excepción. Muchas tecnologías ni siquiera permiten que hilos distintos al principal modifiquen la interfaz de usuario.

Para conseguir que ese fragmento de código sea ejecutado/sincronizado por el hilo principal utilizamos el **SwingUtilities.invokeLater (Runnable).**

```
public  void actualizarMemoria(String memoria) {
    SwingUtilities.invokeLater(new Runnable(){
        public void run(){
            textoMemoria.setText("Memoria libre: "+memoria +" ("+peticiones+")");
        }
    });
}
```

Ejecutamos la aplicación y debería funcionar sin bloqueos de interfaz, permitiéndonos pulsar los dos botones.