



Componentes Web

Sesión 5 - WebSocket



Índice

- WebSocket en Java EE
- Intercambio de mensajes
- Conversión entre Java y mensajes
- Parámetros
- Clientes Javascript



Introducción a WebSocket

- Los servlets encapsulan el mecanismo petición/respuesta de HTTP
 - Si queremos actualizar datos con frecuencia deberemos realizar peticiones periódicas
- Para algunas aplicaciones es conveniente mantener un canal de comunicación abierto
 - Chat online
 - Retransmisión de un evento en directo
- Podemos utilizar WebSocket para establecer un canal de comunicación entre cliente y servidor
 - Importante cuando la frecuencia de actualización deba ser alta
 - Estándar adoptado por la mayoría de navegadores actuales



WebSocket en Java EE 7

- Se define en JSR-356 (*Java API for WebSocket*)
 - Contenida en el paquete `javax.websocket`
 - Permite crear *endpoints* de tipo `WebSocket`
 - La clase principal es `Endpoint`
- Dos tipos posibles de *endpoints*
 - *Endpoints* programados
 - *Endpoints* mediante anotaciones



Endpoints programados

- Heredamos de `Endpoint`
- Podemos sobrescribir los métodos `onOpen`, `onClose` y `onError`

Obligatorio

`onMessage` nos notifica la llegada de un mensaje

```
public class MiEndpoint extends Endpoint {
    @Override
    public void onOpen(final Session session, EndpointConfig config) {
        session.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String msg) {
                // Interactuar con el objeto session para intercambiar datos
                ...
            }
        });
    }
}
```

- Desplegamos el *endpoint* también de forma programada

```
ServerEndpointConfig.Builder.create(MiEndpoint.class, "/socket").build();
```



`ws://localhost:8080/miaplicacion/socket`



Endpoints mediante anotaciones

- Es la forma más sencilla de crearlos
- Clase anotada con `@ServerEndpoint`
 - Se pasa como parámetro la dirección de despliegue
 - El despliegue es automático

No es necesario implementar obligatoriamente `onOpen` ni definir un gestor de mensajes para implementar el método `onMessage`, podemos anotarlo directamente

```
@ServerEndpoint("/socket")
public class EchoEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        // Interactuar con el objeto session para intercambiar datos
        ...
    }
}
```

Destacamos el objeto `Session`. Se utilizará para intercambiar mensajes



Anotación de métodos

- Podemos anotar los métodos del *endpoint* de la siguiente forma:

| Anotación | Descripción |
|-------------------------|--|
| <code>@OnOpen</code> | Se llama al abrirse el canal de datos |
| <code>@OnClose</code> | Se llama al cerrarse el canal de datos |
| <code>@OnError</code> | Se llama al producirse un error en la comunicación |
| <code>@OnMessage</code> | Se llama al recibirse un mensaje |

```
@ServerEndpoint("/socket")
public class EchoEndpoint {

    @OnOpen
    public void open(Session session,
                    EndpointConfig conf) {

    }

    @OnClose
    public void close(Session session,
                    CloseReason reason) {

    }

    @OnError
    public void error(Session session,
                    Throwable error) {

    }

    @OnMessage
    public void onMessage(Session session,
                    String msg) {

    }

}
```



Mantenimiento del estado

- Al contrario que en el caso de los servlets, cada instancia del endpoint atiende un único cliente
 - Podemos utilizar variables de instancia para mantener el estado con seguridad
 - No tendremos problemas de concurrencia
- Otra forma alternativa de mantener el estado es utilizar `session.getUserProperties()`
 - Mapa que guarda información en forma de parejas *<clave, valor>*



Intercambio de mensajes

- Encontramos diferentes tipos de mensajes
 - Envío de datos de texto o binarios
 - *Ping*
 - *Pong*
- El envío se puede realizar de dos formas
 - Síncrono
 - Objeto `RemoteEndpoint.Basic`
 - Se obtiene mediante `session.getRemoteBasic()`
 - Asíncrono
 - Objeto `RemoteEndpoint.Async`
 - Se obtiene mediante `session.getRemoteAsync()`

Métodos de `RemoteEndpoint`

Descripción

`sendText (String)`

Envía un mensaje de texto al *endpoint* remoto de forma bloqueante o no bloqueante (según el tipo de `RemoteEndpoint`)

`sendBinary (ByteBuffer)`

Envía un mensaje binario al *endpoint* remoto de forma bloqueante o no bloqueante (según el tipo de `RemoteEndpoint`)

`sendPing (ByteBuffer)`

Envía un *ping* al *endpoint* remoto

`sendPong (ByteBuffer)`

Contesta con un *pong* al *endpoint* remoto



Envío de un mensaje de texto

- Podemos enviar un mensaje de texto como respuesta a otro mensaje

```
@ServerEndpoint("/socket")
public class MiEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        session.getBasicRemote().sendText("Recibido " + msg);
    }
}
```

- Podemos enviar un mensaje sin haber recibido un mensaje entrante
 - Guardar el objeto `Session` como variable de instancia en `OnOpen`
 - Utilizarlo para enviar un mensaje en cualquier momento



Difusión de mensajes

- Podemos enviar un mensaje a todas las sesiones abiertas
 - Útil en aplicaciones tipo *chat*

```
@ServerEndpoint("/chat")
public class ChatEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        try {
            for (Session s : session.getOpenSessions()) {
                if (s.isOpen())
                    s.getBasicRemote().sendText(msg);
            }
        } catch (IOException e) { }
    }
}
```

- Podríamos filtrar sesiones
 - Por ejemplo utilizando propiedades de `session.getUserProperties()`



Recepción de mensajes

- Podemos sobrecargar el método `OnMessage`
 - Mensajes de texto (`String`, `Reader`)
 - Mensajes binarios (`byte []`, `ByteBuffer`, `InputStream`)
 - Mensajes *pong* (`PongMessage`)

```
@ServerEndpoint("/socket")
public class ReceiveEndpoint {
    @OnMessage
    public void texto(Session session, String msg) {
    }

    @OnMessage
    public void binario(Session session, ByteBuffer msg) {
    }

    @OnMessage
    public void pong(Session session, PongMessage msg) {
    }
}
```



Conversión de datos de mensaje a Java

- Podemos convertir el formato de los mensajes a clases Java de forma automática
 - Los mensajes pueden estar en cualquier codificación (JSON, XML, codificación propia)
- Debemos definir los objetos:
 - `Encoder (Java > Mensaje)`
 - `Decoder (Mensaje > Java)`
- Por ejemplo, suponemos que trabajamos con el siguiente tipo de datos

```
public class Pelicula {  
    String titulo;  
    String director;  
    int duracion;  
  
    // Getters y setters  
    ...  
}
```



Encoders

- Transforma películas al formato `<titulo>;<director>;<duracion>`

```
public class PeliculaAJsonEncoder implements Encoder.Text<Pelicula> {
    @Override
    public void init(EndpointConfig ec) { }

    @Override
    public void destroy() { }

    @Override
    public String encode(Pelicula p) throws EncodeException {
        String msg = p.getTitulo() + ";" +
                    p.getDirector() + ";" +
                    p.getDuracion();

        return msg;
    }
}
```



Decoders

- Recupera una película a partir del formato `<titulo>;<director>;<duracion>`

```
public class JsonAPeliculaDecoder implements Decoder.Text<Pelicula> {
    @Override
    public void init(EndpointConfig ec) { }

    @Override
    public void destroy() { }

    @Override
    public Pelicula decode(String string) throws DecodeException {
        String [] items = string.split(";");
        Pelicula p = new Pelicula();
        p.setTitulo(items[0]);
        p.setDirector(items[1]);
        p.setDuracion(Integer.parseInt(items[2]));

        return p;
    }

    @Override
    public boolean willDecode(String string) {
        boolean mensajeValido = string.matches("[A-Za-z0-9]+;[A-Za-z0-9]+;[0-9]+");
        return mensajeValido;
    }
}
```

Debemos comprobar si el formato es legible



Uso de *encoders* y *decoders*

- Para que la transformación se aplique de forma automática debemos incluir nuestros *encoders* y *decoders* en la anotación del *endpoint*

```
@ServerEndpoint(  
    value = "/socket",  
    encoders = { PeliculaAJsonEncoder.class }  
    decoders = { JsonAPeliculaDecoder.class }  
)  
public class MiEndpoint {  
    @OnMessage  
    public void libro(Session session, Pelicula p) {  
        Pelicula nuevaPelicula = new Pelicula();  
        nuevaPelicula.setTitulo("Copia de " + p.getTitulo());  
        session.getBasicRemote().sendObject(nuevaPelicula);  
    }  
}
```



Segmentos variables del *path*

- Podemos especificar segmentos variables en la ruta a la que está mapeado el *endpoint*

```
@ServerEndpoint("/tiempo/{cp}")  
public class TiempoEndpoint {  
    ...  
}
```

- Serán válidas cualquiera de las siguientes direcciones

<http://localhost:8080/miaplicacion/tiempo/03001>

<http://localhost:8080/miaplicacion/tiempo/03004>

<http://localhost:8080/miaplicacion/tiempo/03690>

- Podemos inyectar el texto introducido en el segmento variable en cualquier método

```
@ServerEndpoint("/tiempo/{cp}")  
public class ChatEndpoint {  
    String cp;  
    Session session;  
  
    @OnOpen  
    public void open(Session session, EndpointConfig c,  
                    @PathParam("cp") String cp) {  
        this.cp = cp;  
        this.session = session;  
    }  
}
```



Parámetros de la *query*

- También tenemos acceso a los parámetros de la *query*

- Parámetros con el formato:

<http://localhost:8080/miaplicacion/tiempo?hora=18>

- Podemos obtenerlos a través de un mapa de parámetros

```
String hora = session.getRequestParameterMap().get("hora").get(0);
```

- Siempre nos proporcionan como valor una lista porque pueden ser multivaluados

```
if(session.getRequestParameterMap().get("cp")!=null &&  
    session.getRequestParameterMap().get("cp").size() > 0) {  
    String cp = session.getRequestParameterMap().get("cp").get(0);  
}
```



Cliente JavaScript

- JavaScript dispone de un objeto WebSocket que puede crearse a partir del *endpoint*

```
websocket = new WebSocket("ws://localhost:8080/miaplicacion/socket");
```

- Podemos enviar mensajes con
- Podemos definir el *callback* `onMessage`

```
websocket.send("Mensaje");
```

```
websocket.onmessage = onMessage;
```

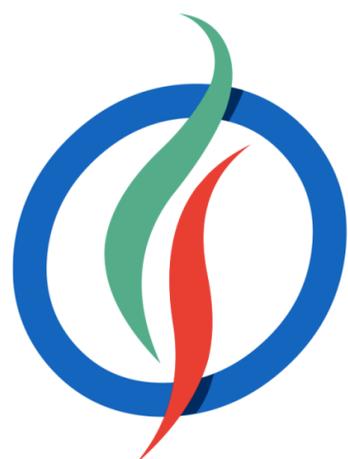
- Ejemplo completo:

```
var websocket;

function connect() {
  websocket = new WebSocket("ws://localhost:8080/miaplicacion/socket");
  websocket.onmessage = onMessage;
}

function onMessage(event) {
  var items = event.data.split(";");
  document.getElementById("titulo").innerHTML = items[0];
  document.getElementById("director").innerHTML = items[1];
}

window.addEventListener("load", connect, false);
```



¿Preguntas?