# WebSocket

# WebSocket

Introduction
WebSocket et JEE
Points d’accès programmés ou annotés
Envoyer et recevoir des messages
Codage des messages
URL paramétrées
Exemple : un forum

# Introduction : pourquoi des WebSocket

HTTP : protocole client serveur classique
le client initie la conversation : il demande une ressource et le serveur envoie la réponse

Le serveur ne peut envoyer aucune donnée sans sollicitation du client

Limitation vraiment importante quand le client a besoin de plus d'interaction

Le protocole WebSocket permet d’y remédier en fournissant un canal de communication bidirectionnel entre le client et le serveur

# Introduction : le principe

- Le serveur publie un point d'accès WebSocket
- Le client utilise l'URI de ce point d'accès pour se connecter au serveur
- Une fois la connexion établie, le protocole est symétrique
- Le client et le serveur peuvent s'envoyer des messages
- 2 phases dans le protocole : l'établissement de la connexion et l'échange de données

# Introduction : le principe

le client initie l'établissement de la connexion par l'envoi d'une requête à l'URI du point d'accès :

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

réponse du serveur :

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: K7DJLdLooIwIG/MOpvWFB3y3FE8=
```

# Introduction : le principe

vérification de la connexion avec les champs `Sec-WebSocket-Key` et `Sec-WebSocket-Accept`

forme des uri pour les points d'accès :

- `ws://host:port/path?query`
- `wss://host:port/path?query`

messages échangés :

- texte en UTF-8
- binaires

# WebSocket et JEE

Java EE inclue Java API for WebSocket (JSR 356)
les packages utilisés :

- `javax.websocket.server` : annotations, classes et interfaces pour créer et configurer des points d'accès côté serveur

- `javax.websocket` : annotations, classes, interfaces et exceptions communes au client et au serveur

- les points d'accès sont instances de la classe `javax.websocket.EndPoint`, on peut les créer par programmation ou par annotation

# WebSocket et JEE

créer et déployer un point d’accès websocket :

1. créer une classe de point d’accès

2. implémenter les méthodes du cycle de vie du point d’accès

3. ajouter le code métier

4. déployer le point d’accès dans une application web

# Points d’accès programmés

## extension de la classe EndPoint

```
public class EchoEndpoint extends Endpoint {
   @Override
   public void onOpen(final Session session, EndpointConfig config) {
      session.addMessageHandler(new MessageHandler.Whole<String>() {
         @Override
         public void onMessage(String msg) {
            try {
               session.getBasicRemote().sendText(msg);
            } catch (IOException e) { ... }
         }
      });
   }
}
```

# Points d’accès programmés

- réemet chaque message reçu
- la classe `EndPoint` définit 3 méthodes `onOpen`, `onClose`, `onError`
  `onOpen` est la seule méthode abstraite de `EndPoint`
- `Session` : représente une conversation entre ce point d’accès et le point d’accès distant
- `addMessagehandler` : enregistre les traitants de messages
- `getBasicRemote` : renvoie l’objet représentant le point d’accès distant

# Points d’accès programmés

- le traitant de message est ici implanté par une classe interne anonyme
- la méthode onMessage est appelée à la réception d’un message
- pour déployer le point d’accès, il faut utiliser ce code :
  `ServerEndPointConfig.Builder.create(EchoEndpoint.class,”/echo”).build()`
- il sera accessible à l’URI :
  `ws://<host>:<port>/<application>/echo`

# Points d’accès annotés

même point d’accès annoté :

```
@ServerEndpoint("/echo")

public class EchoEndpoint {

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

        try {

            session.getBasicRemote().sendText(msg);

        } catch (IOException e) { ... }

    }

}
```

# Points d'accès annotés

| annotation | événement | exemple |
|---|---|---|
| `OnOpen` | ouverture de connexion | `@OnOpen`<br>`public void open(Session session,`<br>`EndpointConfig conf) { }` |
| `OnMessage` | réception d'un message | `@OnMessage`<br>`public void message(Session session,`<br>`String msg) { }` |
| `OnError` | erreur de connexion | `@OnError`<br>`public void error(Session session,`<br>`Throwable error) { }` |
| `OnClose` | fermeture de connexion | `@OnClose`<br>`public void close(Session session,`<br>`CloseReason reason) { }` |

# Envoi de messages

1. obtenir l'objet `Session` depuis la connexion
   soit comme paramètre de l'une  des méthodes du cycle de vie du point d'accès
   soit conserver sa référence dans un attribut à l'ouverture de connexion
2. obtenir l'objet `RemoteEndPoint` à partir de l'objet `Session` à partir des méthodes `Session.getBasicRemote` et `Session.getAsyncRemote` qui renvoient un objet `RemoteEndPoint.Basic` ou `RemoteEndPoint.Async`
3. utiliser l'objet `RemoteEndPoint` pour envoyer des messages avec les méthodes :
   - `void RemoteEndpoint.Basic.sendText(String text)`
   - `void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)`
   - `void RemoteEndpoint.sendPing(ByteBuffer appData)`
   - `void RemoteEndpoint.sendPong(ByteBuffer appData)`

# Envoi de messages

- une instance de `EndPoint` est associée avec une et une seule connexion avec un pair

- envoyer un message à tous les pairs connectés au `EndPoint` avec la méthode `getOpenSessions`

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

# Réception de messages

l'annotation `onMessage` déclare la méthode en charge de la réception des messages, trois méthodes peuvent être annotées :

```
@ServerEndpoint("/receive")
public class ReceiveEndpoint {

    @OnMessage
    public void textMessage(Session session, String msg) {
        System.out.println("Text message: " + msg);
    }

    @OnMessage
    public void binaryMessage(Session session, ByteBuffer msg) {
        System.out.println("Binary message: " + msg.toString());
    }

    @OnMessage
    public void pongMessage(Session session, PongMessage msg) {
        System.out.println("Pong message: " + msg.getApplicationData().toString());
    }
}
```

# Réception de messages : maintenir un état client

- le container crée une instance de `EndPoint` pour chaque connexion : on peut utiliser des attributs pour conserver de l'information sur l'état du client
- on peut aussi utiliser `Session.getUserProperties` qui renvoie un dictionnaire où l'utilisateur peut conserver de l'information.

```
@ServerEndpoint("/delayedecho")
public class DelayedEchoEndpoint {
   @OnOpen
   public void open(Session session) {
      session.getUserProperties().put("previousMsg", " ");
   }
   @OnMessage
   public void message(Session session, String msg) {
      String prev = (String) session.getUserProperties().get("previousMsg");
      session.getUserProperties().put("previousMsg", msg);
      try {
         session.getBasicRemote().sendText(prev);
      } catch (IOException e) { ... }
   }
}
```

# Codage des messages

Il s'agit de convertir des messages `WebSocket` en types Java et vice versa

Pour utiliser des encodeurs dans un point d'accès :

1. implémenter l'une de ces interfaces :
   `Encoder.Text<T>` pour les messages texte
   `Encoder.Binary<T>` pour les messages binaires
   ces interfaces spécifient la méthode `encode`.
2. ajouter les noms des encodeurs au paramètre `encoders` de l'annotation `ServerEndpoint`
3. utiliser la méthode `sendObject(Object data)` de `RemoteEndpoint.Basic` ou `RemoteEndPoint.Async` pour envoyer les objets comme des messages. Le container recherchera l'encodeur correspondant au type de l'objet.

# Codage des messages

```
public class MessageATextEncoder implements Encoder.Text<MessageA> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(MessageA msgA) throws EncodeException {
      // Access msgA's properties and convert to JSON text...
      return msgAJsonString;
   }
}

public class MessageBTextEncoder implements Encoder.Text<MessageB> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(MessageA msgB) throws EncodeException {
      // Access msgB's properties and convert to JSON text...
      return msgBJsonString;
   }
}
```

# Codage des messages

au niveau du point d'accès :

```
@ServerEndpoint(
    value = "/myendpoint",
    encoders = { MessageATextEncoder.class, MessageBTextEncoder.class }
)
public class EncEndpoint { ... }
```

pour envoyer des messages :

```
MessageA msgA = new MessageA(...);
MessageB msgB = new MessageB(...);
session.getBasicRemote.sendObject(msgA);
session.getBasicRemote.sendObject(msgB);
```

# Codage des messages

pour le décodage :

1. implémenter l'une de ces interfaces :
   `Decoder.Text<T>` pour les messages texte
   `Decoder.Binary<T>` pour les messages binaires
   ces interfaces spécifient les méthodes `willDecode` et `decode`.

2. ajouter les noms de vos encodeurs au paramètre `decoders` de l'annotation `ServerEndpoint`

3. utiliser l'annotation `onMessage` sur une méthode qui prend en paramètre le type Java résultat du décodage

# Codage des messages

```
public class MessageTextDecoder implements Decoder.Text<Message> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public Message decode(String string) throws DecodeException {
      // Read message...
      if ( /* message is an A message */ )
         return new MessageA(...);
      else if ( /* message is a B message */ )
         return new MessageB(...);
   }
   @Override
   public boolean willDecode(String string) {
      // Determine if the message can be converted into either a
      // MessageA object or a MessageB object...
      return canDecode;
   }
}
```

# Codage des messages

```
@ServerEndpoint(
   value = "/myendpoint",
   encoders = { MessageATextEncoder.class, MessageBTextEncoder.class },
   decoders = { MessageTextDecoder.class }
)
public class EncDecEndpoint {

...

  @OnMessage
  public void message(Session session, Message msg) {
     if (msg instanceof MessageA) {
        // We received a MessageA object...
     } else if (msg instanceof MessageB) {
        // We received a MessageB object...
     }
  }
}
```

# URL paramétrées

L’annotation ServerEndPoint permet d’utiliser des modèles d’URI qui pourront être utilisées comme paramètres de l’application.

Exemple d’un chat :

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
    ...
}
```

déploiement dans une application de nom chatapp sur un serveur local sur le port 8080, le client peut accéder aux URI suivantes :

http://localhost:8080/chatapp/chatrooms/currentnews
http://localhost:8080/chatapp/chatrooms/music
http://localhost:8080/chatapp/chatrooms/cars
http://localhost:8080/chatapp/chatrooms/technology

# URL paramétrées

on peut utiliser les paramètres dans les méthodes annotées par `onOpen`, `onMessage` et `onClose`

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
   @OnOpen
   public void open(Session session,
                    EndpointConfig c,
                    @PathParam("room-name") String roomName) {
      // Add the client to the chat room of their choice ...
   }
}
```

# Exemple

L’application monForum est composée d’une webSocket, d’un EJB et d’une page HTML :

- le point d’accès accepte les connexions des clients et leur envoie les nouveaux messages disponibles

- l’EJB ajoute les nouveaux messages

- la page HTML utilise javascript pour se connecter au point d’accès, traite les messages reçus et met à jour les messages sans rechargement de page

# Exemple : le point d'accès

```
@ServerEndpoint("/forumep")
public class ForumEndpoint {

   /* Queue for all open WebSocket sessions */
   static Queue<Session> queue = new ConcurrentLinkedQueue<>();

   /* ForumBean calls this method to send updates */
   public static void send(String newmess) {
      try {
         /* Send updates to all open WebSocket sessions */
         for (Session session : queue) {
            session.getBasicRemote().sendText(newmess);
         }
      } catch (IOException e) {
      }
   }
   ...
```

# Exemple : le point d’accès

```
...
@OnOpen
public void openConnection(Session session) {
   /* Register this connection in the queue */
   queue.add(session);
}

@OnClose
public void closedConnection(Session session) {
   /* Remove this connection from the queue */
   queue.remove(session);
}

@OnError
public void error(Session session, Throwable t) {
   /* Remove this connection from the queue */
   queue.remove(session);
}
}
```

# Exemple : EJB

```
@Stateless
public class ForumBean implements ForumRemote {

  ...

  public void addMessage(String message) {
    ...
    ForumEndPoint.send(message);
  }

  ...

}
```

# Exemple : page HTML

```
<!DOCTYPE html>
<html>
<head>...</head>
<body>
  ...
  <table>
    ...
    <td id="lastmessage">---</td>
    ...
  </table>
</body>
</html>
```

# Exemple : page HTML

```
var wsocket;

function connect() {
   wsocket = new WebSocket("ws://localhost:8080/Forum/forumep");
   wsocket.onmessage = onMessage;
}

function onMessage(evt) {
   document.getElementById("lastmessage").innerHTML = evt;
}

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