# WebSocket

Auch WebSockets können von Plugins ganz einfach genutzt werden. Dabei können sogar erweiterte Funktionalitäten wie Subprotokolle verwendet oder der Handshake modifiziert werden.

```
fun registerWebSocket(socketId: String, handler: WebSocketHandler)
```

`socketId` ist eine frei wählbare ID, die über alle WebSockets eines Plugins hinweg eindeutig sein muss. Es ist kein Problem, wenn mehrere Plugins einen WebSocket mit derselben ID bereitstellen, da zum Ansprechen eines WebSockets auch die `id` des Plugins herangezogen wird.

Der WebSocket wird dann unter einem Pfad bereitgestellt, der nach folgendem Schema aufgebaut ist.

```
/.well-known/bx-websockets/<plugin ID>/<socketId>
```

Als `handler` übergibt man eine Instanz, die `com.batix.plugins.WebSocketHandler` implementiert.

> **Thread-Safety**
> Dieselbe Instanz wird für alle Requests verwendet, muss also thread-safe implementiert werden.

Im Handler können eine oder mehrere der folgenden Methoden überschrieben werden. Die Methodennamen sind selbsterklärend.

```
fun onOpen(client: WebSocketContext)
fun onClose(client: WebSocketContext, closeReason: CloseReason)
fun onTextMessage(client: WebSocketContext, message: String)
fun onBinaryMessage(client: WebSocketContext, data: ByteArray)
fun onError(client: WebSocketContext, throwable: Throwable) // ab Framework v2.7.8
```

Mittels `client` kann der Gegenüber identifiziert werden. In `WebSocketContext` stecken die `httpSession` (falls eine beim Öffnen des WebSockets vorhanden war) sowie die `webSocketSession` mit der z. B. Nachrichten geschickt oder der Socket geschlossen werden kann. Außerdem gibt es noch die Methode `getAllClients()` mit der alle aktuell zu diesem WebSocket Handler verbundenen Clients aufgelistet werden (das schließt auch `client` mit ein).

Der häufigste Fall ist eine Textnachricht zu versenden, dies kann wie folgt erledigt werden.

```
client.webSocketSession.basicRemote.sendText("Text...")
```

Eine Verbindung kann mittels `close()` geschlossen werden, dabei ist auch die Angabe einer `CloseReason` möglich.

```
client.webSocketSession.close(CloseReason(
  CloseReason.CloseCodes.NORMAL_CLOSURE, "Verbindung beendet."
))
```

# Erweiterte Funktionalitäten

Für diese Funktionalitäten muss der WebSocket schon beim Start des Frameworks initialisiert werden. Dafür ist es nötig die entsprechenden Informationen in der `plugin.yaml` Datei zu hinterlegen und das Plugin beim Systemstart zu laden. Dazu wird in `plugin.yaml` ein neuer Hauptkey `configuredWebSockets` ergänzt.

```
configuredWebSockets:
  advanced-socket:
    subprotocols:
      - proto1
      - proto2
    configurator: com.company.plugin.MySocketConfigurator
```

Dies ist eine Map, wobei der Key (im Beispiel hier `advanced-socket`) der `socketId` entspricht. Diese muss dann durch das Plugin noch registriert werden (wie normale WebSockets). Dem Beispiel folgend, müsste also noch folgender Aufruf erfolgen.

```
registerWebSocket("advanced-socket", SomeHandler())
```

`subprotocols` und `configurator` können beide zusammen oder auch einzeln pro WebSocket verwendet werden.

> **Nur nach Framework-Neustart verfügbar**
> Diese speziellen WebSockets sind nur nach einem Neustart des Frameworks, und nur wenn das Plugin zu diesem Zeitpunkt automatisch geladen wird, verfügbar. Damit Änderungen an `subprotocols` und eine neue `configurator` Instanz wirksam werden, muss das Framework neugestartet werden.

# Subprotokolle

Falls der Server mehrere Subprotokolle unterstützt, können diese unter `subprotocols` aufgelistet werden (Liste von Strings). Schickt ein Client seinerseits auch Subprotokolle mit, wird das erste verwendet, welches auch der Server unterstützt. Diese Auswahl kann optional durch einen Configurator angepasst werden (s. u.).

Das ausgehandelte Subprotokoll kann folgendermaßen ausgelesen werden. Wurde sich auf kein Subprotokoll geeinigt (oder gab es keine), ist dieser Wert leer.

```
client.webSocketSession.negotiatedSubprotocol
```

# Configurator

Weitere Details können durch einen sogenannten Configurator angepasst werden. Dies ist eine von `jakarta.websocket.server.ServerEndpointConfig.Configurator` abgeleitete Klasse.

> **javax / jakarta**
> Ab Framework v3.0 müssen die `jakarta` anstatt der `javax` Klassen verwendet werden.

Der vollqualifizierte Name der Klasse (d. h. inklusive Package) ist als `configurator` des entsprechenden WebSockets in der `plugin.yaml` anzugeben. Hier gibt es verschiedene Methoden, die überschrieben werden können. Es folgt eine Auswahl.

```
fun getNegotiatedSubprotocol(supported: List<String>, requested: List<String>): String
```

Hiermit kann die Auswahl des Subprotokolls getroffen werden. Es ist ein String zurückzugeben, der sowohl in `supported` (Liste der Subprotokolle des **Servers**), als auch in `requested` (Liste der Subprotokolle des **Clients**) vorkommt. Ist kein Subprotokoll akzeptabel, muss ein Leerstring zurückgegeben werden.

```
fun checkOrigin(originHeaderValue: String): Boolean
```

Mit dieser Methode kann der Origin-Header des Clients überprüft werden, falls für diesen nur bestimmte Werte zugelassen sein sollen. Der Rückgabewert ist, ob der Check erfolgreich war. Diesen Header schicken ziemlich alle Browser mit, andere Clients aber ggf. nicht (können ihn auch fälschen).

```
fun modifyHandshake(
  sec: ServerEndpointConfig,
  request: HandshakeRequest,
  response: HandshakeResponse
)
```

Hierdurch kann die HTTP-Response des Verbindungsaufbaus angepasst werden. Subprotokolle und der Origin-Check sind an dieser Stelle schon durchlaufen wurden.

# Beispiel

Das Beispiel implementiert den simpelst-möglichen Broadcast-WebSocket, d. h. eine eingehende Nachricht wird an alle verbundenen Clients geschickt (inkl. dem Sender).

```
override fun load() {
  registerWebSocket("broadcast", Broadcaster())
}
```

Lautet die Plugin-ID `com.batix.website:import-mitarbeiter`, so ist der WebSocket dann unter folgender URL erreichbar.

```
wss://domain.tld/.well-known/bx-websockets/com.batix.website:import-mitarbeiter/broadcast
```

> **ws/wss und Domain auslesen**
> Um in JavaScript das passende Protokoll (`ws` oder `wss`) und die Domain herauszufinden, kann folgendes Snippet verwendet werden (Standardports vorausgesetzt).

```
const wsPath = "/.well-known/bx-websockets/<plugin ID>/<socketId>";
const protocolPrefix = (window.location.protocol === 'https:') ? 'wss:' : 'ws:';
const socket = new WebSocket(protocolPrefix + "//" + location.host + wsPath);
```

> Grundsätzlich empfiehlt sich eine verschlüsselte HTTPS-Verbindung, damit auch die WebSocket-Verbindung verschlüsselt ist.

Im Handler ist lediglich die `onTextMessage` Methode zu überschreiben.

```
import com.batix.plugins.WebSocketContext
import com.batix.plugins.WebSocketHandler
```

```
class Broadcaster : WebSocketHandler {
  override fun onTextMessage(client: WebSocketContext, message: String) {
    client.allClients.forEach {
      it.webSocketSession.basicRemote.sendText(message)
    }
  }
}
```