

# Les web sockets

# Web sockets

- Un mécanisme de communication client/serveur bidirectionnelle asynchrone.
- Le client est le navigateur – sans HTTP
- Utilise:
  - TCP pour le transport
  - HTTP pour l'initialisation
- Moins de surcharge que HTTP.
  - En-têtes plus petits que HTTP

# API Web Socket

- L'API permet l'envoi des messages du client à un serveur et vice versa en full duplex
- Efficace
- Syntaxe simple
- Standard w3c
- Nouveau paradigme de communication client/serveur (nouveau modèle)

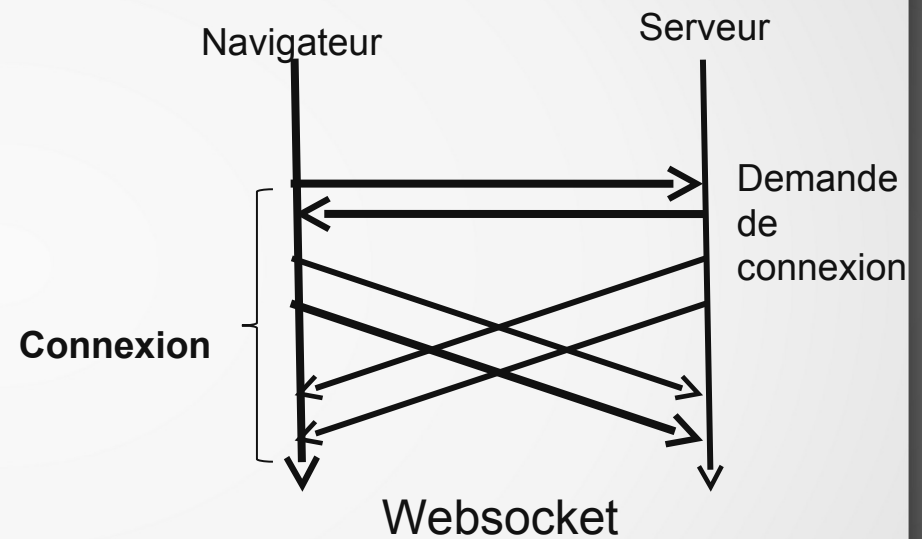
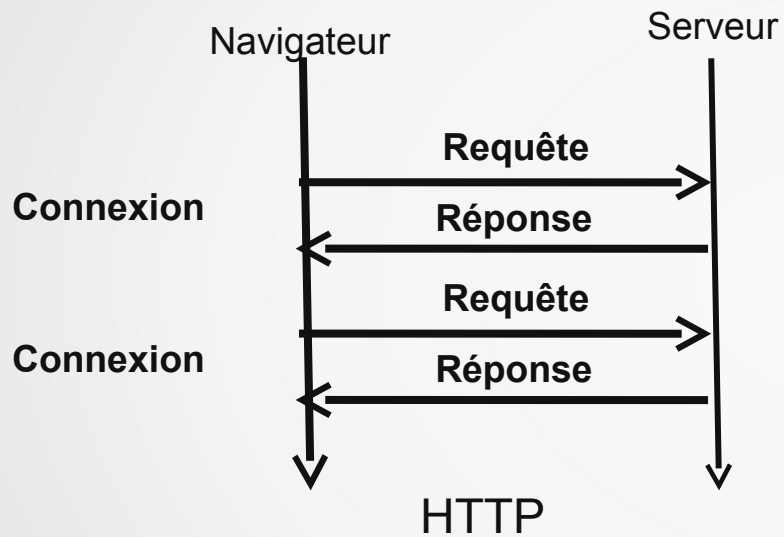
# Web socket vs Ajax

- WebSocket est une alternative à AJAX
- AJAX exige une demande par le client pour chaque transaction
- Dans WebSocket, le client et le serveur peuvent émettre les messages

# Paradigme

- Le client demande une connexion sous forme d'en-têtes HTTP contenant les informations sur la WebSocket choisie
- Le serveur répond avec un en-tête HTTP
- Une connexion permanente est ouverte
- Les deux échangent des messages sur cette connexion.

# Paradigme



# Utilisation de web sockets

- Le navigateur doit supporter l'API websocket avec JavaScript:
  - C'est le cas de chrome, firefox, ...
- Le serveur web doit comprendre le protocole websocket
  - Exemples:
    - API websocket pour Java
    - Application QT >=QT5.3
    - Module websocket dans node.js

# Classe WebSocket (côté client)

WebSocket
<ul style="list-style-type: none"><li>+ url : USVString</li><li>+ CONNECTING : unsigned short</li><li>+ OPEN : unsigned short</li><li>+ CLOSING : unsigned short</li><li>+ CLOSED : unsigned short</li><li>+ readyState : unsigned short</li><li>+ bufferedAmount : unsigned long long</li><li>+ onopen : EventHandler</li><li>+ onerror : EventHandler</li><li>+ onclose : EventHandler</li><li>+ extensions : DOMString</li><li>+ protocol : DOMString</li><li>+ onmessage : EventHandler</li><li>+ binaryType : BinaryType</li></ul>
<ul style="list-style-type: none"><li>+ close()</li><li>+ send(in data: USVString)</li><li>+ send(in data: Blob)</li><li>+ send(in data: ArrayBuffer)</li><li>+ send(in data: ArrayBufferView)</li><li>+ WebSocket(in url: USVString)</li></ul>



# URL websocket (côté client)

- Utilise une socket TCP
  - Non sécurisée : URI **ws://**
  - Sécurisée : URI **wss://**
- Exemple:
  - `var socket = new WebSocket('ws://172.18.58.15:8888');`

# Événements à gérer

- Après la création de la WebSocket, il y a 4 événements à gérer:
  - Open
  - Message
  - Close
  - Error

# Événements à gérer

- Open
  - Action à effectuer lorsque la connexion est établie.
  - Se gère avec **onopen**.
- Message
  - Action à effectuer lorsque le client reçoit un message du serveur.
  - Se gère avec **onmessage**.
- Close
  - Action à effectuer lorsque la connexion est fermée.
  - Se gère avec **onclose**.
- Error
  - Action à effectuer lorsqu'une erreur se produit.
  - Se gère avec **onerror**.

# Méthodes de la WebSocket

- 2 methodes
  - send
  - close

# Exemple de client

```
$(function ()  
{  
    var maWebsocket;  
    // l'API WebSocket est-elle installée ?  
    if (window.WebSocket)  
    {  
        // création de la WebSocket  
        maWebsocket = new WebSocket('ws://172.18.58.148:8888');  
  
        //Gestion des événements de la WebSocket  
        maWebsocket.onopen = function ()  
        {  
            console.log("ouverture websocket");  
        };  
        maWebsocket.onclose = function (event)  
        {  
            console.log("code de la fermeture : " + event.code);  
            console.log("raison de la fermeture : " + event.reason);  
        };  
        maWebsocket.onerror = function ()  
        {  
            console.log("erreur sur la websocket");  
        };  
        maWebsocket.onmessage = function (donneesRecues)  
        {  
            console.log("réception de données : " + donneesRecues.data);  
            console.log("origine : " + donneesRecues.origin);  
        };  
    }  
    $("#demande").click(function(){  
        // envoyer la chaîne "date" au serveur de WebSocket  
        maWebsocket.send("date");  
    });  
});
```

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>TODO supply a title</title>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <script src="js/libs/jquery/jquery.js" type="text/javascript"></script>  
    <script src="js/exempleWebSocket.js" type="text/javascript"></script>  
  </head>  
  <body>  
    <div id="demande">Cliquez ici</div>  
    <div id="ladate"></div>  
  </body>  
</html>
```

Modifiez le code afin que les données reçues apparaissent dans la div "ladate"

# Côté Serveur / La classe QWebSocketServer

## Fonctionnement identique à QTcpServer

enum QWebSocketServer::SslMode

Indicates whether the server operates over wss (SecureMode) or ws (NonSecureMode)

Constant	Value	Description
QWebSocketServer::SecureMode	0	The server operates in secure mode (over wss)
QWebSocketServer::NonSecureMode	1	The server operates in non-secure mode (over ws)

QWebSocketServer::QWebSocketServer(const QString &serverName, QWebSocketServer::SslMode secureMode, QObject \*parent = nullptr)

Constructs a new QWebSocketServer with the given serverName. The serverName will be used in the HTTP handshake phase to identify the server. It can be empty, in which case no server name will be sent to the client. The secureMode parameter indicates whether the server operates over wss (SecureMode) or over ws (NonSecureMode).

parent is passed to the QObject constructor.

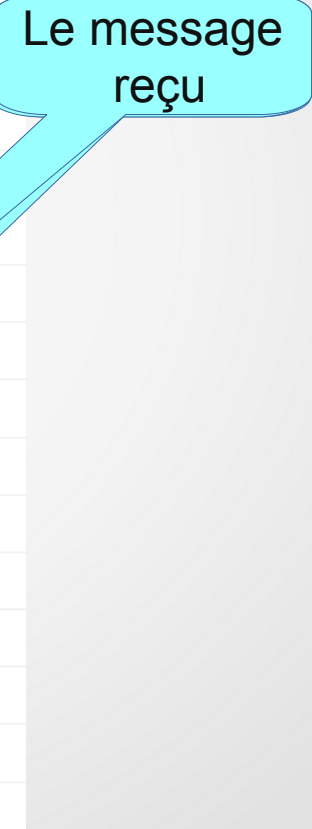
# QWebSocket / Reception

## Fonctionnement identique à QTcpSocket

### Signals

void	<b>aboutToClose()</b>
void	<b>binaryFrameReceived</b> (const QByteArray &frame, <del>bool isLastFrame</del> )
void	<b>binaryMessageReceived</b> (const QByteArray &message)
void	<b>bytesWritten</b> (qint64 bytes)
void	<b>connected()</b>
void	<b>disconnected()</b>
void	<b>error</b> (QAbstractSocket::SocketError error)
void	<b>pong</b> (qint64 elapsedTime, const QByteArray &payload)
void	<b>preSharedKeyAuthenticationRequired</b> (QSslPreSharedKeyAuthentication *authenticator)
void	<b>proxyAuthenticationRequired</b> (const QNetworkProxy &proxy, QAuthenticator *authenticator)
void	<b>readChannelFinished()</b>
void	<b>sslErrors</b> (const QList<QSslError> &errors)
void	<b>stateChanged</b> (QAbstractSocket::SocketState state)
void	<b>textFrameReceived</b> (const QString &frame, bool isLastFrame)
void	<b>textMessageReceived</b> (const QString &message)

Le message  
reçu



# QWebSocket / Émission

## Fonctionnement identique à QTcpSocket

qint64	<code>sendBinaryMessage(const QByteArray &amp;data)</code>
qint64	<code>sendTextMessage(const QString &amp;message)</code>



# Serveur de WebSocket avec QT

En vous aidant de la documentation sur les classes **QWebSocket** et **QWebSocketServer**, codez l'application tel que:

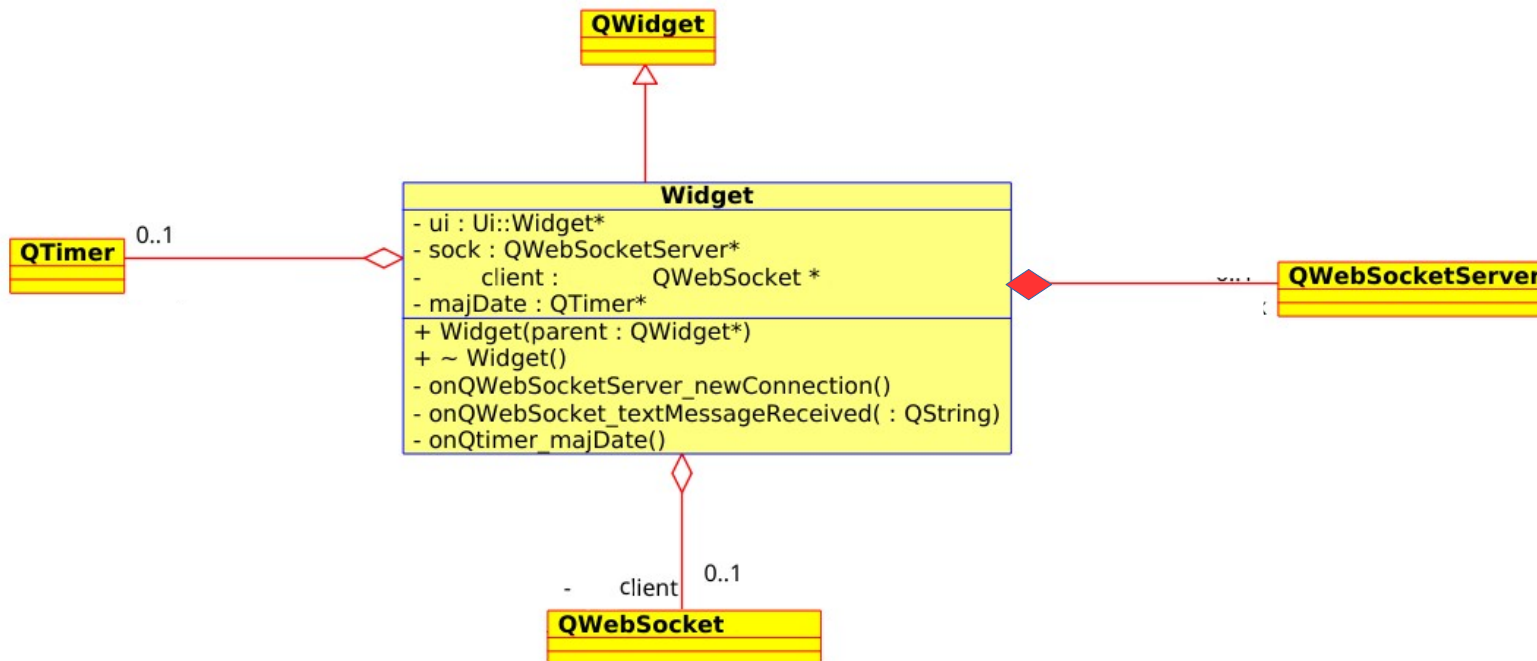
- Le serveur envoie la date du système toutes les deux secondes au client qui a envoyé la chaîne de caractères "date".

Pour avoir la date du système sous forme de chaîne de caractères:

```
QDateTime::currentDateTime().toString()
```

Ci-dessous, le diagramme de classe de l'application attendue.

- Modifiez le serveur afin qu'il puisse répondre à plusieurs clients (voir TD3 QTcpServer)



# Documentation de référence

- <https://html.spec.whatwg.org/multipage/web-sockets.html>
- <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>