

Java EE 7 アプリケーション設計ガイド - WebSocket編



Disclaimer

- この資料は日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム システムズ・エンジニアリング株式会社の正式なレビューを受けておりません。
- 当資料は、資料内で説明されている製品の仕様を保証するものではありません。
- 資料の内容には正確を期するよう注意しておりますが、この資料の内容は2014年9月現在の情報であり、製品の新しいリリース、PTFなどによって動作、仕様が変わる可能性があるにご注意下さい。
- 今後国内で提供されるリリース情報は、対応する発表レターなどをご確認ください。
- I B M、I B Mロゴおよびibm.comは、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。他の製品名およびサービス名等は、それぞれ I B Mまたは各社の商標である場合があります。現時点での I B Mの商標リストについては、www.ibm.com/legal/copytrade.shtmlをご覧ください。
- 当資料をコピー等で複製することは、日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム システムズ・エンジニアリング株式会社の承諾なしではできません。
- 当資料に記載された製品名または会社名はそれぞれの各社の商標または登録商標です。
- JavaおよびすべてのJava関連の商標およびロゴは Oracleやその関連会社の米国およびその他の国における商標または登録商標です。
- Microsoft, Windows および Windowsロゴは、Microsoft Corporationの米国およびその他の国における商標です。
- Linuxは、Linus Torvaldsの米国およびその他の国における登録商標です。
- UNIXはThe Open Groupの米国およびその他の国における登録商標です。

目次

1. WebSocketの概要
2. WebSocket Protocolによる双方向通信について
3. 既存技術との比較(HTTP, Ajax, Comet)
4. WebSocketを用いたアプリケーション開発

1. WebSocketの概要



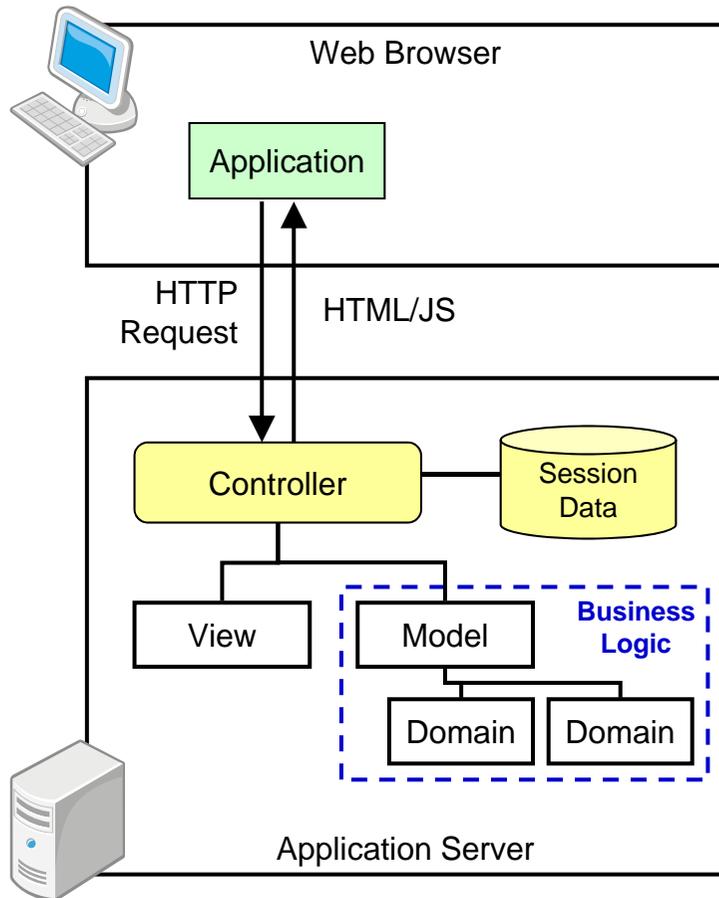
WebSocket登場の背景 – HTML5の登場

- 従来のHTML
 - 画面をレンダリングするための規格
- HTML5 (+ CSS3 / JavaScript)
 - アプリケーションを記述するための規格

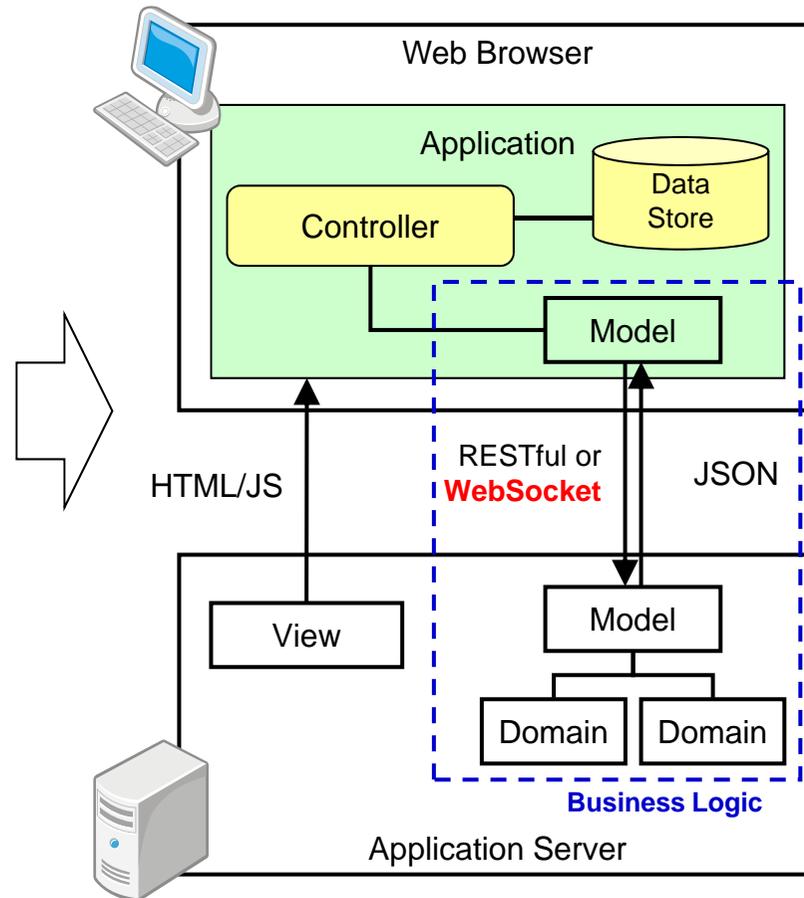


WebSocket登場の背景 - HTML5環境の新しいアプリケーションスタイル

- 「サーバーサイドMVC」から「クライアントMVC」へ
- バックエンドサーバーとの通信技術としてWebSocketが誕生



従来のWebアプリケーション



Single Page Application (SPA)

WebSocket登場の背景 - HTML5におけるクライアント・サーバー連携

■ 通信方法

– RESTful

- HTTPの原則にしたがった
簡潔なシステム間連携の設計手法

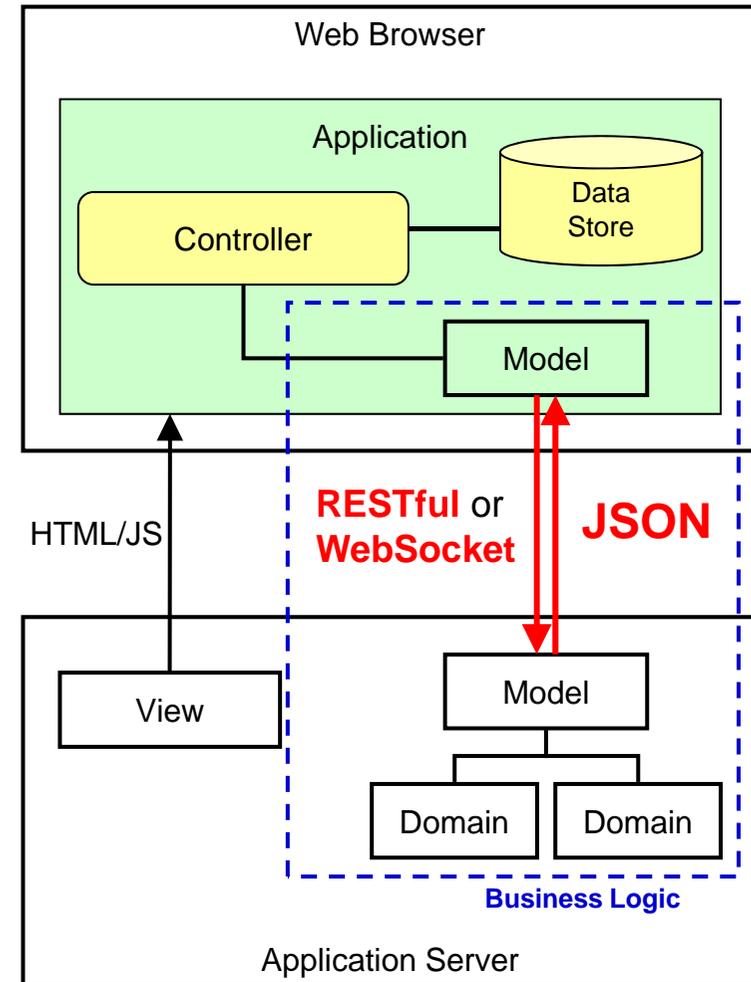
– WebSocket

- ブラウザとWebサーバーの間で
双方向通信を可能にする
通信プロトコル

■ データ型

– JSON (JavaScript Object Notation)

- JavaScriptのオブジェクトとして
そのままパースできる
テキスト形式のデータ表記法



WebSocketとは



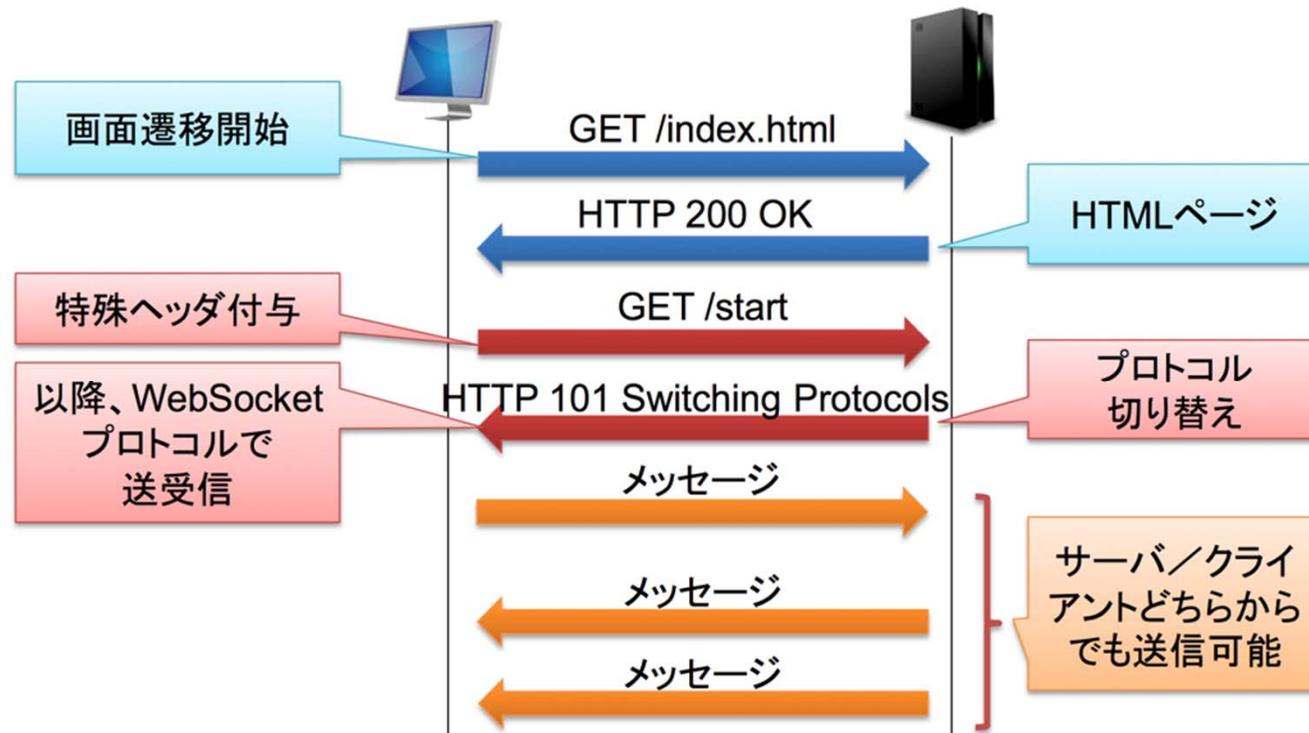
- クライアントとサーバー間の双方向通信用の技術規格
- TCP上で動く
- HTTP UpgradeリクエストによりWebSocketにプロトコルスイッチする

- 仕様
 - WebSocket Protocol
 - 元々はHTML5の仕様の一部として策定されていた
 - その後HTML5から切り離され、単独のプロトコルとしてIETF RFC 6455で仕様化
 - WebSocket API
 - JavaScript APIはW3Cで仕様化
 - Java API for WebSocket 1.0はJava EE 7(JSR 356) で仕様化

- 用途
 - リアルタイムアプリケーション
 - Single Page Applicationとして画面遷移することなく、画面表示をリアルタイムに更新できる
 - 例) オンライントレード、モニタリング、オンラインゲーム、チャット、SNS、画面共有など

WebSocketの特徴

- クライアントとサーバーの間で行う一連のやり取りを単一のTCPコネクション内で行う
- クライアントだけでなく、サーバー側からのプッシュ配信が可能
- 任意の回数・タイミングでメッセージがやり取りできる



Webブラウザの対応状況

- モダンなブラウザの大部分がサポート



MSIE
10 (2013/2) ~



Chrome
16 (2011/12) ~



Firefox
11 (2012/3) ~



Safari
6 (2012/7) ~



iOS
6.0 (2012/9) ~



Android
4.4 (2013/10) ~

WebSphere Application ServerのWebSocket対応状況 (2014年9月時点)

- WebSphere Application Server ～V8.5.5はWebSocketに対応していない
 - Full ProfileおよびLiberty Profile共にJava EE 7はサポートされていないため
Java API for WebSocketには対応していない

- WAS Liberty Profile BetaがJava API for WebSocketに対応
 - Liberty Betaの製品モジュールは下記URLからダウンロード可能
<https://developer.ibm.com/wasdev/>
 - WebSocketのサンプルアプリケーションが公開されている

WebサーバープラグインによるWebSocketの負荷分散

- Webサーバープラグイン V8.5.5.3以降
- WebSocketのプロキシとしてアプリケーションサーバーへの割振りが可能
- httpからWebSocketへプロトコルスイッチするためのupgradeヘッダーに対応

PI17642: PREPARE FOR WEBSOCKET SUPPORT TO BE ADDED TO WEBSPPHERE APPLICATION SERVER.
<http://www-01.ibm.com/support/docview.wss?uid=swg1PI17642>

2. WebSocket Protocolによる 双方向通信について

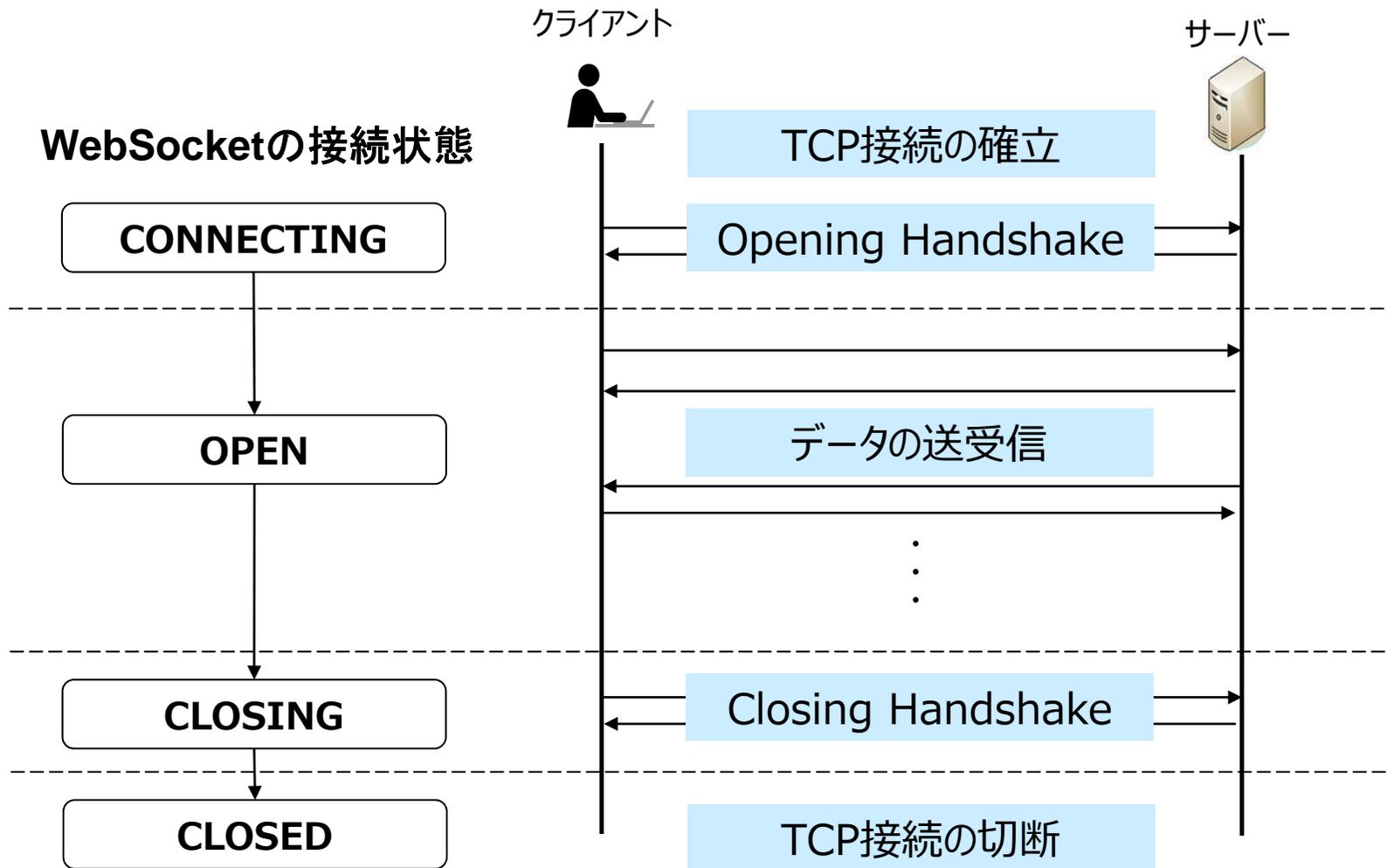


WebSocket URI

- URIスキームとして以下を使用する
 - ws:// WebSocketによるNon-SSL通信(80ポート)
 - wss:// WebSocketによるSSL通信 (443ポート)
- httpと同様に80/443ポートを使用するため、Firewallの追加設定は不要

WebSocketのライフサイクル

- WebSocketの接続、データの送受信、切断までのライフサイクルは以下のとおり



WebSocket接続の確立 : Opening Handshake

- HTTPでWebSocketのハンドシェイクを行う
 - クライアントがHTTP GET要求でOpening Handshakeを送信し、サーバーがHTTP応答を返すことでHTTPからWebSocketにプロトコルスイッチする
 - WebSocketの接続確立はクライアントが起点となるが、データの送受信は双方向で実施する
- Opening Handshakeのやりとりは以下のとおり



Opening Handshakeで利用されるHTTPヘッダー

- WebSocket Protocolで新しく定義されたHTTPヘッダーは以下のとおり

ヘッダーフィールド	リクエスト/ レスポンス	説明
Sec-WebSocket-Key	リクエスト	クライアント側でSec-WebSocket-Keyを発行してサーバーに送信することで、有効なOpening Handshakeであることを立証するために使われる
Sec-WebSocket-Accept	レスポンス	Sec-WebSocket-Keyの値から生成したSec-WebSocket-Acceptをレスポンスヘッダーで受信することでWebSocket接続を開始することができる
Sec-WebSocket-Protocol	リクエスト レスポンス	アプリケーションサブプロトコルをネゴシエートするために使われる。クライアントがサポートされるプロトコルリストを送信し、サーバーが受け付け可能なサブプロトコル(ひとつあるいはnull)を返す
Sec-WebSocket-Version	リクエスト レスポンス	WebSocketプロトコルのバージョンを表す クライアントが指定したバージョンにサーバーが対応していない場合は対応バージョンをレスポンスヘッダで通知する
Sec-WebSocket-Extensions	リクエスト レスポンス	WebSocket extensionsを使用する場合に付与される

WebSocket接続の切断 : Closing Handshake

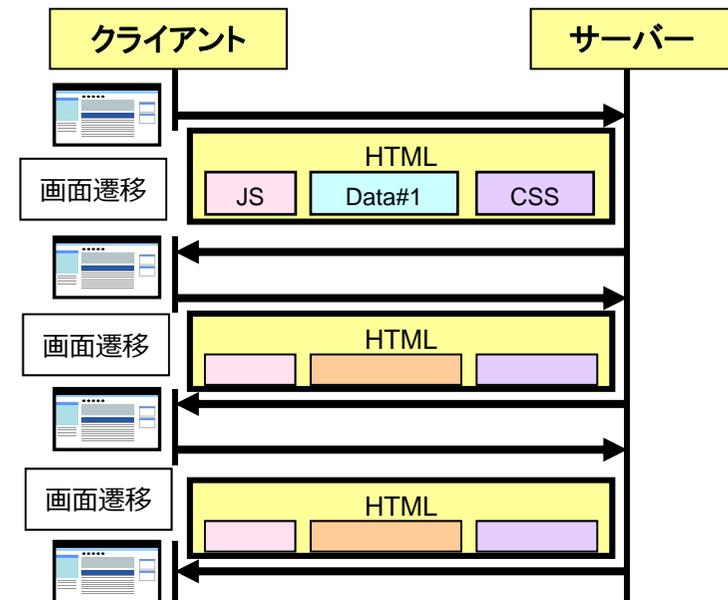
- WebSocketのClosing HandshakeはOpening Handshakeと比べて単純
- Close制御フレームを送信または受信することでWebSocket接続はCLOSING状態に入る

3. 既存技術との比較(HTTP, Ajax, Comet)



HTTPについて

- 処理形式
 - クライアントがサーバーにHTTP要求を送信し、サーバーがクライアントにHTTP応答を返す
 - クライアントからのリクエストなしにサーバーがデータを送信することはできない
- 接続
 - 基本的には1回のTCP接続で1回のHTTP要求/応答を行う
 - HTTP KeepAlive接続により複数HTTPリクエストを投げることが可能
- 通信内容
 - HTTP要求/応答にはHTTPヘッダー情報が付加される
- HTTPを利用した単純なWebアプリケーション
 - 画面の更新は画面遷移によって実施
 - HTMLやCSSを毎回読み込む必要がある
 - 画面遷移中はユーザー操作を行えない



Ajax (Asynchronous JavaScript + XML)

概要

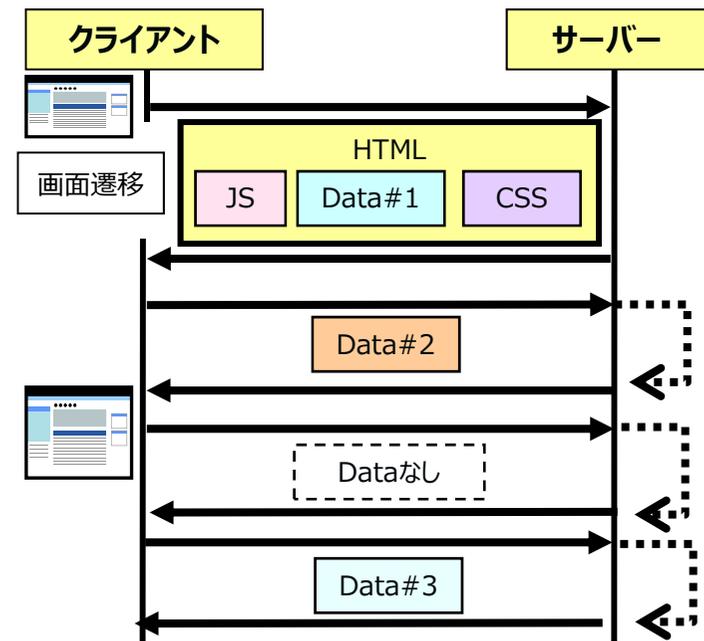
- Webブラウザ上でXMLHttpRequestオブジェクト(Javascript)を使用し非同期通信処理を行う例)Google Map

メリット

- 画面遷移することなく部分的に画面更新ができる
- ユーザーの操作を継続しながら、サーバーからデータを非同期に取得できる

デメリット

- クライアントからのHTTPリクエストで動作するためサーバーが自動的にデータをPush配信できない。定期的なポーリング(Pull型)によりサーバー側のイベントを検知する必要がある
⇒双方向のリアルタイム通信には向いていない
- HTTP通信の度に発生するTCP接続やHTTPヘッダーのやりとりによるオーバーヘッドが生じる
- 1回の接続で1回のデータ配信 (HTTP要求/応答の仕組みのため)



Comet

概要

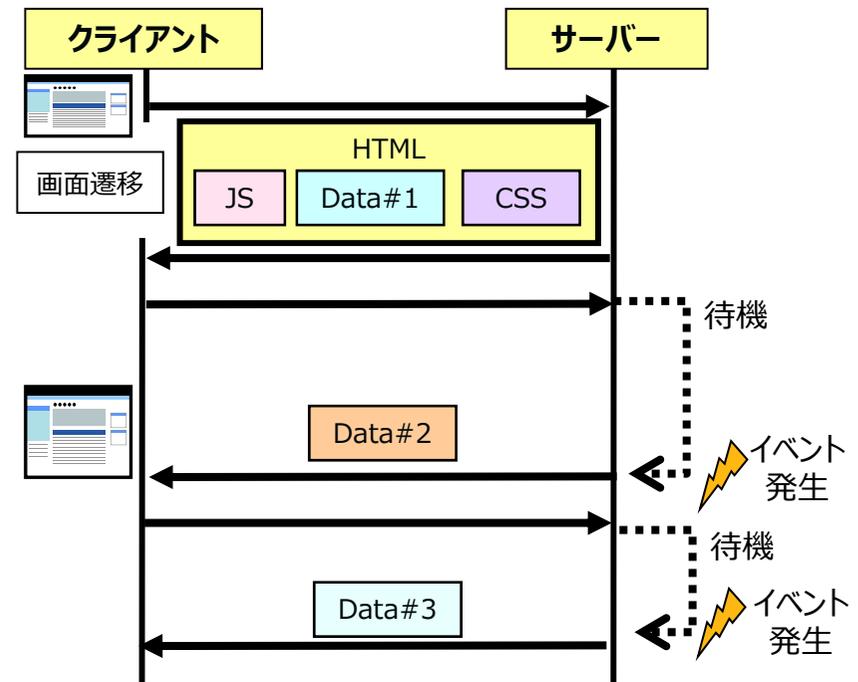
- HTTPによるロングポーリングによってサーバー側の擬似的なPush配信を可能にする
- ロングポーリングとは
クライアントからのHTTPリクエストに対して保留状態にしておき、サーバ側のイベントが発生したタイミングでHTTPレスポンスを送信する仕組み。保留状態でタイムアウトが発生すると再度ロングポーリングのリクエストを送信して待機する。

メリット

- サーバーからクライアントへのデータ配信についてリアルタイム性が向上する
- クライアントとサーバーの双方向のデータ送信が可能になる

デメリット

- Ajaxと同様にHTTPをベースにしているためHTTP通信の度に発生するTCP接続やHTTPヘッダーのやりとりによるオーバーヘッドが生じる
- 1回の接続で1回のデータ配信 (HTTP要求/応答の仕組みのため)



Ajax/Cometとの比較、WebSocketのメリット

- 双方向リアルタイム通信における技術的な課題をWebSocketが解決している

比較項目	Ajax	Comet	WebSocket
使用プロトコル	HTTP	HTTP	WebSocket(ハンドシェイク時にHTTPを利用)
1回のTCP接続で可能なデータ取得回数(*)	1回(要求/応答の1往復)	1回(要求/応答の1往復)	複数回
クライアントからのPull型データ取得	可能	可能	可能
サーバーからのPush型データ配信	不可能	ロングポーリングにより擬似的Push配信が可能	可能
リアルタイム通信	不向き(定期的なポーリングが必要)	ロングポーリングにより擬似的リアルタイム通信が可能	可能
HTTPヘッダーのオーバーヘッド	あり	あり	なし
TCPハンドシェイクによるオーバーヘッド	あり(*)	あり(*)	なし

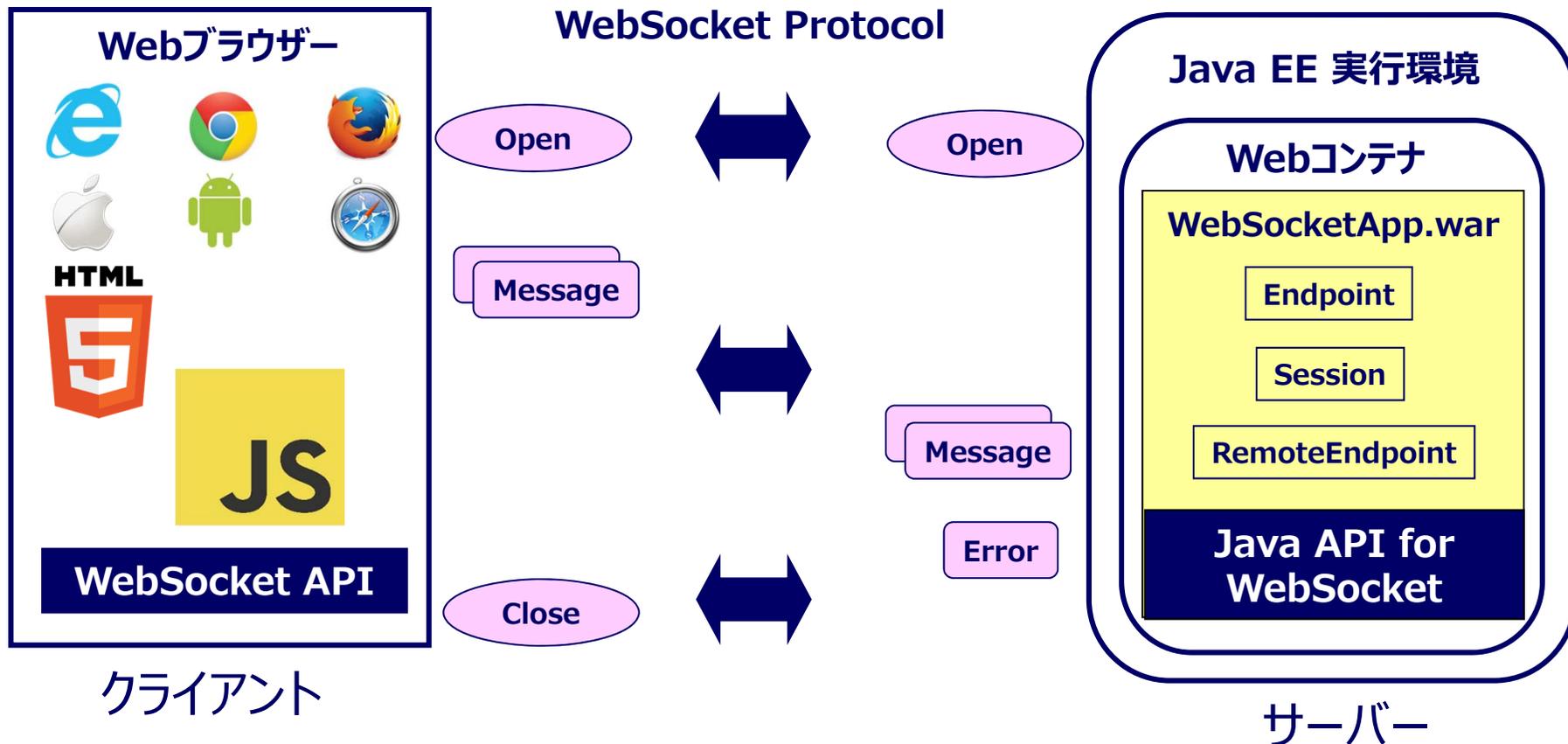
(*)HTTP KeepAliveを使用しない場合

4. WebSocketを用いたアプリケーション開発



WebSocketを利用したアプリケーションの開発

- クライアント側はJavaScript (WebSocket API)で実装
- サーバー側はJava EE (Java API for WebSocket)で実装



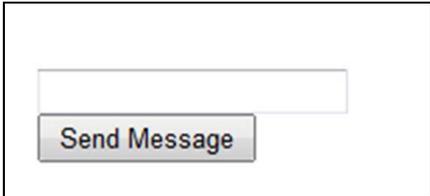
JavaScriptによるクライアント側の実装(WebSocket API)

- 一般的なWebアプリケーションにおいてクライアント側はJavaScriptを使用する

JavaScriptのWebSocket APIを使用したクライアント側の実装例

```
<body>
....
<div>
<input id="inputmessage" type="text" />
</div>
<div>
<input type="submit" value="Send Message" onclick="send()" />
</div>
<div id="messages"></div>
<script language="javascript" type="text/javascript">
//WebSocketオブジェクトのインスタンス化
var websocket = new WebSocket('ws://' + window.document.location.host + '/WebsocketApp/hello');
//WebSocketライフサイクルに対応したイベントハンドラ
websocket.onerror = function(event) { ... };
websocket.onopen = function(event) { ... };
websocket.onmessage = function(event) { ... };
websocket.onclose = function(event) { ... };
//メッセージの送信
function send() {
    var txt = document.getElementById('inputmessage').value;
    websocket.send(txt);
    return false;
}
</script>
</body>
```

Webブラウザからメッセージを送信する



Java API for WebSocket

■ 用語

– WebSocketエンドポイント(endpoint)

- WebSocketの接続が確立された2つのコンポーネント(クライアント、サーバー)のうち片側のコンポーネントを指す。

– WebSocket接続(connection)

- WebSocketプロトコルを使用した2つのエンドポイント間のネットワーク接続を指す。

– ピア(peer)

- WebSocketで接続する相手のエンドポイントを指す。

– WebSocketセッション(session)

- 1つのエンドポイントと1つのピアの間におけるWebSocketセッションを指す。

– クライアントエンドポイントとサーバーエンドポイント(client endpoints and server endpoints)

- クライアントエンドポイントがピアに対して接続を開始し、サーバーエンドポイントがピアからの接続を受け入れる。その逆のフローは存在しない。

Java EEにおけるサーバーエンドポイントの実装

- サーバーエンドポイント
 - サーバー側はJava EE(@ServerEndpoint)を使用する

```
@ServerEndpoint("/hello")
public class HelloEndpoint{
    Session currentSession = null;
    //接続がオープンしたとき
    @OnOpen
    public void onOpen(Session session, EndpointConfig ec) {
        currentSession = session;
    }
    //メッセージを受信したとき
    @OnMessage
    public void receiveMessage(String msg) throws IOException {
        //メッセージをクライアントに送信する
        currentSession.getBasicRemote().sendText("Hello" + msg);
    }
    //接続がクローズしたとき
    @OnClose
    public void onClose(Session session, CloseReason reason) { ... }
    //接続エラーが発生したとき
    @OnError
    public void onError(Throwable t) { ... }
}
```

Java EEにおけるクライアントエンドポイントの実装

- クライアントエンドポイント

- Java EEでもクライアントエンドポイントとして@ClientEndpointアノテーションを定義している

```
@ClientEndpoint
class HelloClient {
    @OnOpen
    public void open(Session session) { ... }
```

- クライアント側から接続する必要があるため、クライアント側のJavaコードの中でサーバーエンドポイントとのセッションを確立するコードが必要となる

```
ContainerProvider.getWebSocketContainer().connectToServer(
    HelloClient.class, URI.create("ws://localhost:9080/hello"));
```

Programmatic Endpoints

- アノテーションを使用せずにEndpointクラスを継承してエンドポイントを作成する方法は以下のとおり

```
public class HelloEndpoint extends Endpoint {
    @Override
    public void onOpen(final Session session, EndpointConfig config) {
        session.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String msg) { ... }
        });
    }
}
```

Java API for WebSocketで定義されているアノテーション

- @ServerEndpoint
 - サーバー側のエンドポイントクラスに付与する
- @ClientEndpoint
 - クライアント側のエンドポイントクラスに付与する
- @OnOpen / @OnMessage / @OnError / @OnClose
 - WebSocketの接続確立から切断までの各ライフサイクルのイベントに対応したメソッドに付与する

アノテーション	イベント	例
@OnOpen	接続がオープンしたとき	@OnOpen Public void open(Session session, EndpointConfig conf) {}
@OnMessage	メッセージを受信したとき	@OnMessage Public void message(Session session, String msg) {}
@OnError	接続エラーが発生したとき	@OnError public void error(Session session, Throwable error) {}
@OnClose	接続がクローズしたとき	@OnClose Public void close(Session session, CloseReason reason) {}

- @PathParam
 - WebSocketリクエストURIのパスの一部をパラメータとして取得する際に使用する

```
@ServerEndpoint(value = "/sample/{name}")
Public class sampleEndpoint {
    @OnOpen
    public void open(Session session,EndpointConfig ec, @PathParam("name") String name) { ... }
}
```

メッセージの送信(1/2)

1つのエンドポイントに対してメッセージを送信するステップは以下の通り

1. connectionからSessionオブジェクトを取得する

- ピアからのメッセージに対する応答を送信する場合は@OnMessageアノテーションが付けられたメソッド内でSessionオブジェクトが利用できる

```
@OnMessage
```

```
public void message(Session session, String msg) {}
```

- 送信メッセージがピアへの応答ではない場合、@OnOpenアノテーションが付けられたメソッド内でsessionオブジェクトをインスタンス変数としてストアしておく

```
@OnOpen
```

```
public void onOpen(Session session, EndpointConfig ec) {  
    currentSession = session;  
}
```

2. SessionオブジェクトからRemoteEndpointオブジェクトを取得する

- Session.getBasicRemoteメソッドでRemoteEndpoint.Basicオブジェクトが返される(同期)
- Session.getAsyncRemoteメソッドでRemoteEndpoint.Asyncオブジェクトが返される(非同期)

3. RemoteEndpointオブジェクトを使用してピアに対してメッセージを送信する

- テキストメッセージを送信する場合
void RemoteEndpoint.Basic.sendText(String text)
- バイナリメッセージを送信場合
void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)
- pingフレームを送信する場合
void RemoteEndpoint.sendPing(ByteBuffer appData)
- pongフレームを送信する場合
void RemoteEndpoint.sendPong(ByteBuffer appData)

メッセージの送信(2/2)

- エンドポイントに接続されたすべてのピアに対してメッセージを送信したい場合
例) チャットアプリケーションやオンラインオークションなど
 - getOpenSessionsメソッドを利用する
 - 受信したメッセージを接続された全ピアに送信する場合の例は以下のとおり

```
@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) { ... }
    }
}
```

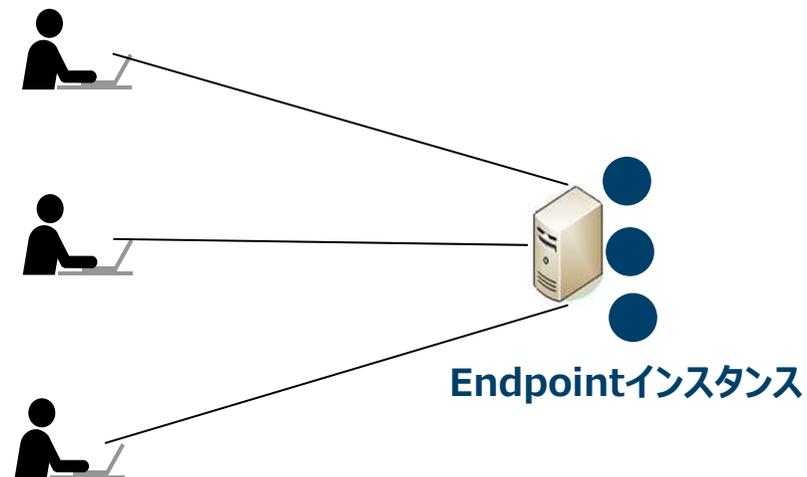
メッセージの受信

- @OnMessageアノテーションにより受信したメッセージをハンドリングすることができる
- メッセージタイプによって以下のようなコードでメッセージを受信する

```
@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());
    }
}
```

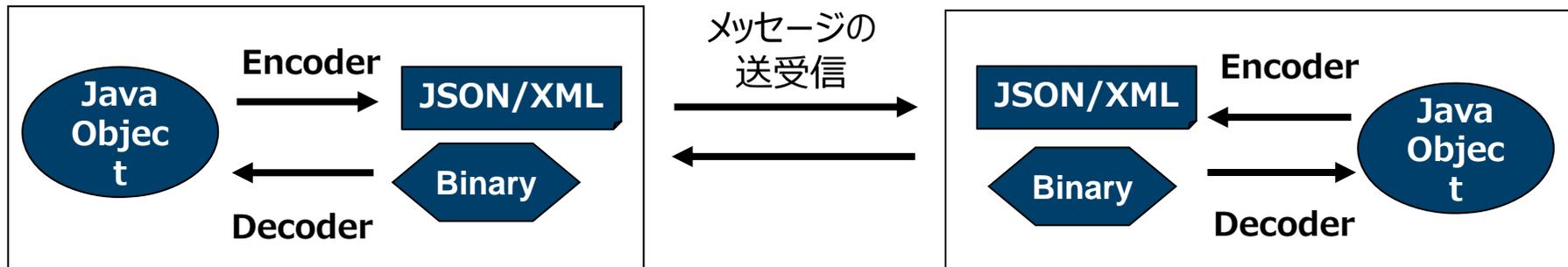
WebSocket Endpoint インスタンスの数とスレッド

- Servletとは違い、WebSocket Endpointクラスのインスタンスは1つだけではなく、デプロイメントURIへの接続毎に複数のインスタンスが生成される
- それぞれのインスタンスは1対の接続に関連付けられている
- Endpointインスタンスのコードを実行するスレッドは常に1つであるため、各コネクションにおけるユーザー状態の保持や開発が容易になる



WebSocketで扱えるメッセージ形式とEncoder/Decoderの実装

- WebSocketで扱えるメッセージ形式はテキストとバイナリ
- EncoderとDecoderの実装
 - EncoderとDecoderを実装することでJavaオブジェクトをWebSocketメッセージとして送受信することができる
 - 任意のデータ型のJavaオブジェクトをテキスト形式(JSON/XMLなど)やバイナリ形式に変換することができる
 - EncoderクラスでJava API for JSON Processing(JSR 353)を使用すればJSON形式に変換することが可能



WebSocketエンドポイントとDependency Injection

- Java EEプラットフォーム上で稼働するWebSocketエンドポイントはCDI仕様のDependency Injection(@Inject)をサポートしている

- 例：CDIを利用したWebSocketとJMS/MDBとの連携
 - エンドポイントが受信したWebSocketメッセージをJMS宛先に送信する
 - WebSocketサーバーエンドポイントにおけるJMS宛先のメッセージ監視はCDIイベントを監視する
 - 詳細は下記URLを参照
 - Integrating WebSockets and JMS with CDI Events in Java EE 7
 - https://blogs.oracle.com/brunoborges/entry/integrating_websockets_and_jms_with

WebSocketにおけるサーバーセキュリティ

- WebSocketはWebコンテナのセキュリティモデルにて保護される
- WebSocketエンドポイントにアクセスする際の認証、ロールベースのアクセス認可、通信の暗号化はOpening handshakeのHTTP GETリクエストURIが対象となる
- 設定はweb.xmlに対して行う

– 認証

- WebSocketの認証に関する仕様は定義されていない
- 認証要件がある場合、WebSocketのOpening handshakeの前にHTTP認証(基本認証やフォーム認証など)で認証を行う
- 保護されたWebSocketに対して未認証のOpening handshakeリクエストを送信した場合、401(Unauthorized)を返す

– 認可

- Opening handshakeリクエストに対してロールベースのアクセス認可が可能
- web.xmlに<security-constraint>エレメントを追加し、<url-pattern>にOpening handshakeのリクエストURIを定義する
- http-methodはGETを指定する必要がある

– 通信の暗号化

- web.xml内の<transport-uarantee>にNONE/CONFIDENTIALを設定する
- ws://の場合はNONE
- wss://の場合はCONFIDENTIAL

WebSocketサンプルアプリケーションの紹介

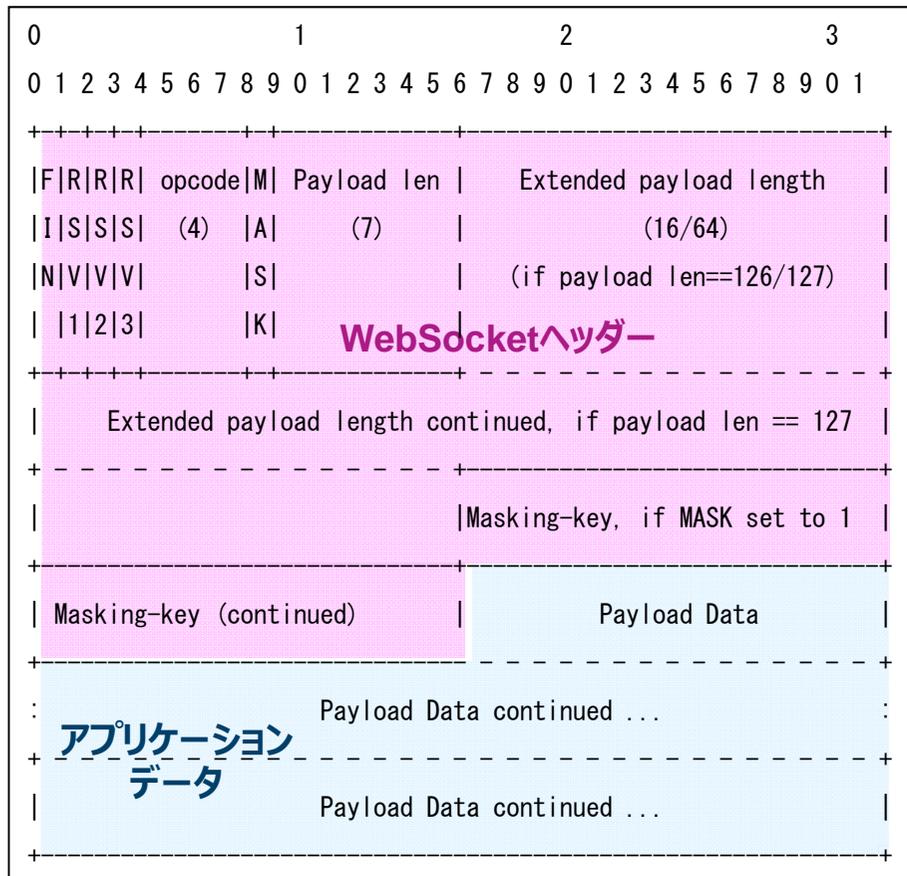
- Liberty ProfileのプロダクトサンプルとしてWebSocket Sampleが公開されている
 - 下記URLよりダウンロード可能
 - https://developer.ibm.com/wasdev/downloads/#asset/samples-WebSocket_Sample
- サンプルの概要
 - WebSocketエンドポイントのセットアップ
 - 接続のオープン、メッセージの送受信、接続のクローズ
 - Encoder、Decoderの利用
 - @PathParamアノテーションの利用
 - pongメッセージの処理

参考資料

- IETF RFC 6455 The WebSocket Protocol
- <http://tools.ietf.org/html/rfc6455>
- JSR 356: Java API for WebSocket
- <https://jcp.org/en/jsr/detail?id=356>
- The Java EE 7 Tutorial
- <http://docs.oracle.com/javaee/7/tutorial/doc/home.htm>
- Liberty Profile Beta
- <https://developer.ibm.com/wasdev/>
- PI17642: PREPARE FOR WEBSOCKET SUPPORT TO BE ADDED TO WEBSPPHERE APPLICATION SERVER.
- <http://www-01.ibm.com/support/docview.wss?uid=swg1PI17642>

(補足資料)WebSocketのデータ送受信 : データフレーム

- WebSocketの接続確立後、メッセージをフレーム単位で送受信する
 - データフレーム形式は以下の通り
 - WebSocketヘッダーのサイズはHTTPヘッダーと比較して非常に小さい



パラメータ	サイズ	説明
FIN	1bit	メッセージの終わりかどうか
RSV1,2,3	各1bit	Extensionを使用するかどうか
opcode	4bits	Payload Dataの説明 (次項で説明)
MASK	1bit	Payload Dataがマスクされているかどうか(クライアントから送信される場合はマスクが必須)
Payload length	7bits, 7+16bits, or 7+64bits	Payload Dataの長さ
Masking-key	0 or 4bytes	マスクする際に使用する 32bitのキーデータ
Payload Data		アプリケーションデータ (Extensionデータと Applicationデータで構成される)

(補足資料) WebSocketのデータ送受信 : opcode

- opcodeはPayload Dataの説明であり、フレームの種類が指定されている
 - %x0 継続フレーム
 - %x1 テキストフレーム
 - %x2 バイナリーフレーム
 - %x3-7 追加の非制御フレームとして予約済み
 - %x8 接続切断
 - %x9 ping(*)
 - %xA pong(*)
 - %xB-F 追加の制御フレームとして予約済み
- } 制御フレーム

(*)pingフレームとpongフレームはクライアントとサーバー間で接続維持の確認、あるいは通信相手が応答可能かどうかを検証するために使用される