



# **Cenni di Java Server Faces e Web Socket**

**Su Virtuale:**

**Versione elettronica 1 pagina per foglio = 4.02.WebSocket.pdf**

**Versione elettronica 2 pagine per foglio = 4.02.WebSocket-2p.pdf**

# Java Server Faces (JSF) come evoluzione di JSP

---

Yet another framework per applicazioni Web?

Sostanzialmente sì...

- Tecnologia fortemente basata su *componenti*, sia da inserire nelle pagine Web, sia collegati tramite essi a componenti server-side (*backing bean*)
- *Ricche API* per *rappresentazione componenti e gestione loro stato, gestione eventi*, validazione e conversione dati server-side, definizione percorso navigazione pagine, supporto a internazionalizzazione
- *Ampia libreria di tag* per aggiungere componenti a pagine Web e per collegarli a componenti server-side
- *On top del supporto Java Servlet e come alternativa a JSP*



## Partiamo da un semplice esempio...

---

Si può costruire un backing bean (o managed bean) in modo semplice

- Annotazione `@ManagedBean` regista automaticamente il componente come risorsa utilizzabile all'interno del container JSF, da parte di tutte le pagine che conoscano come riferirlo

```
package hello;  
import javax.faces.bean.ManagedBean;  
@ManagedBean  
public class Hello {  
    final String world = "Hello World!";  
    public String getworld() {  
        return world; }  
}
```

In questo caso bean Hello è super-semplice, ma in generale contiene logica di business (*controller*) il cui risultato finale è, in modo diretto o tramite invocazione di altri componenti, di produrre dati *model*

---

## Partiamo da un semplice esempio...

---

- ❑ Poi facile costruzione di pagina Web, *scritta in XHTML*, che usi il backing bean
- ❑ Connessione tra pagina Web e componente tramite espressioni in Expression Language (EL)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>Facelets Hello World</title>
</h:head>
<h:body>
    #{hello.world}
</h:body>
</html>
```

Esempio di pagina beanhello.xhtml

*Facelets come linguaggio per costruzione di view JSF e di alberi di componenti* (supporto a XHTML, tag library per Facelets/JSF, supporto per EL, templating per componenti e pagine Web)

---

## Partiamo da un semplice esempio...

---

- In tecnologia JSF, è *inclusa servlet predefinita, chiamata FacesServlet*, che si occupa di gestire richieste per pagine JSF
- Serve mapping tramite solito descrittore di deployment (web.xml)

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

# Ciclo di vita di una applicazione Facelets

---

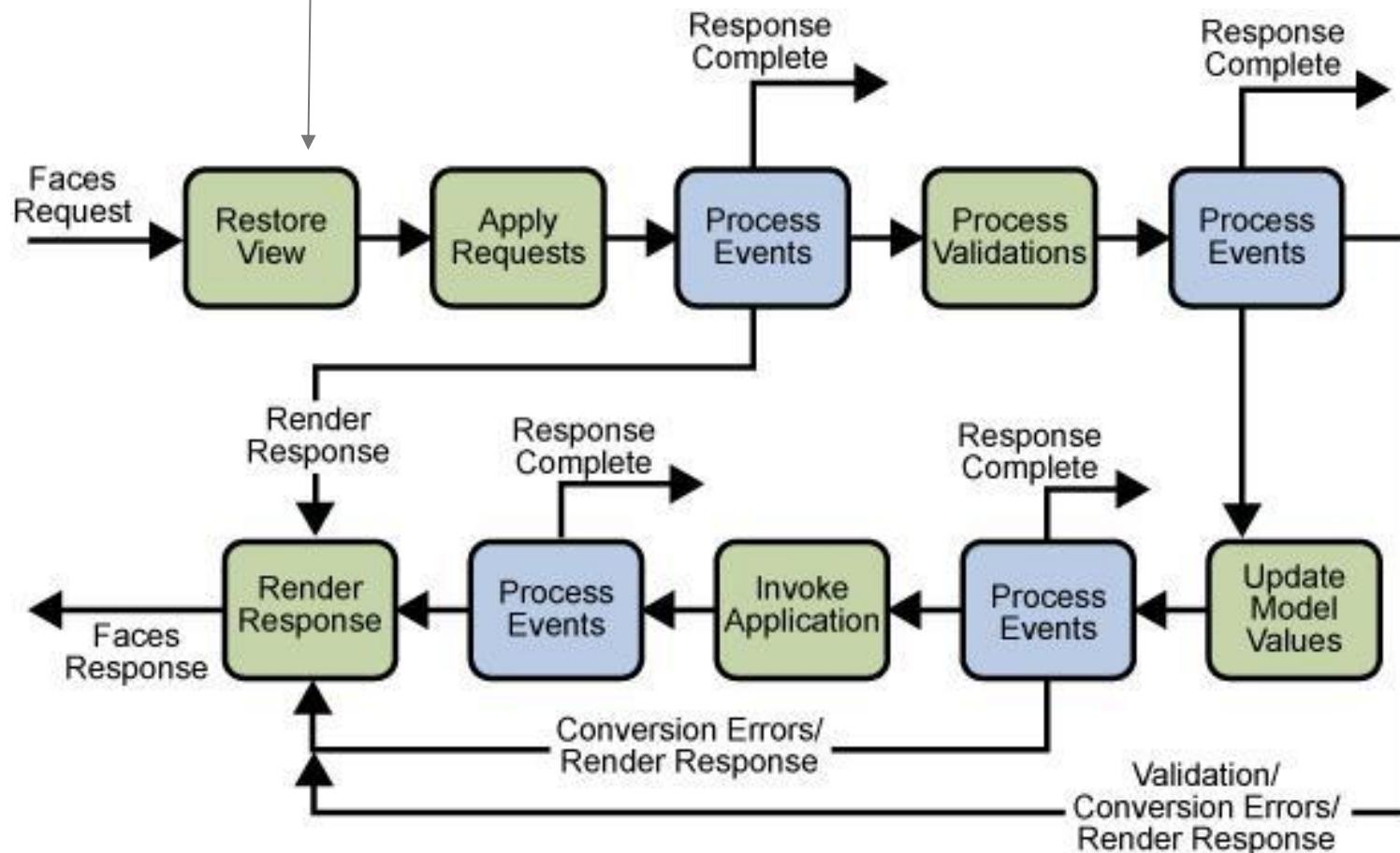
Programmatore può anche non voler avere visibilità della gestione del ciclo di vita dell'applicazione Facelets, svolta automaticamente dal container per JSF (solito container JSP/servlet, con supporto JSF)

Tipico ciclo di vita:

- Deployment dell'applicazione su server; prima che arrivi prima richiesta utente, applicazione in stato non inizializzato (anche non compilato...)
- Quando arriva una richiesta, *viene creato un albero dei componenti contenuti nella pagina* (messo in FacesContext), *con validazione e conversione dati automatizzata*
- Albero dei componenti viene *popolato con valori da backing bean* (uso di espressioni EL), *con possibile gestione eventi e handler*
- Viene costruita una view sulla base dell'albero dei componenti
- Rendering della vista al cliente, basato su albero componenti
- Albero componenti deallocato automaticamente
- In caso di richieste successive (anche postback), l'albero viene ri-allocato

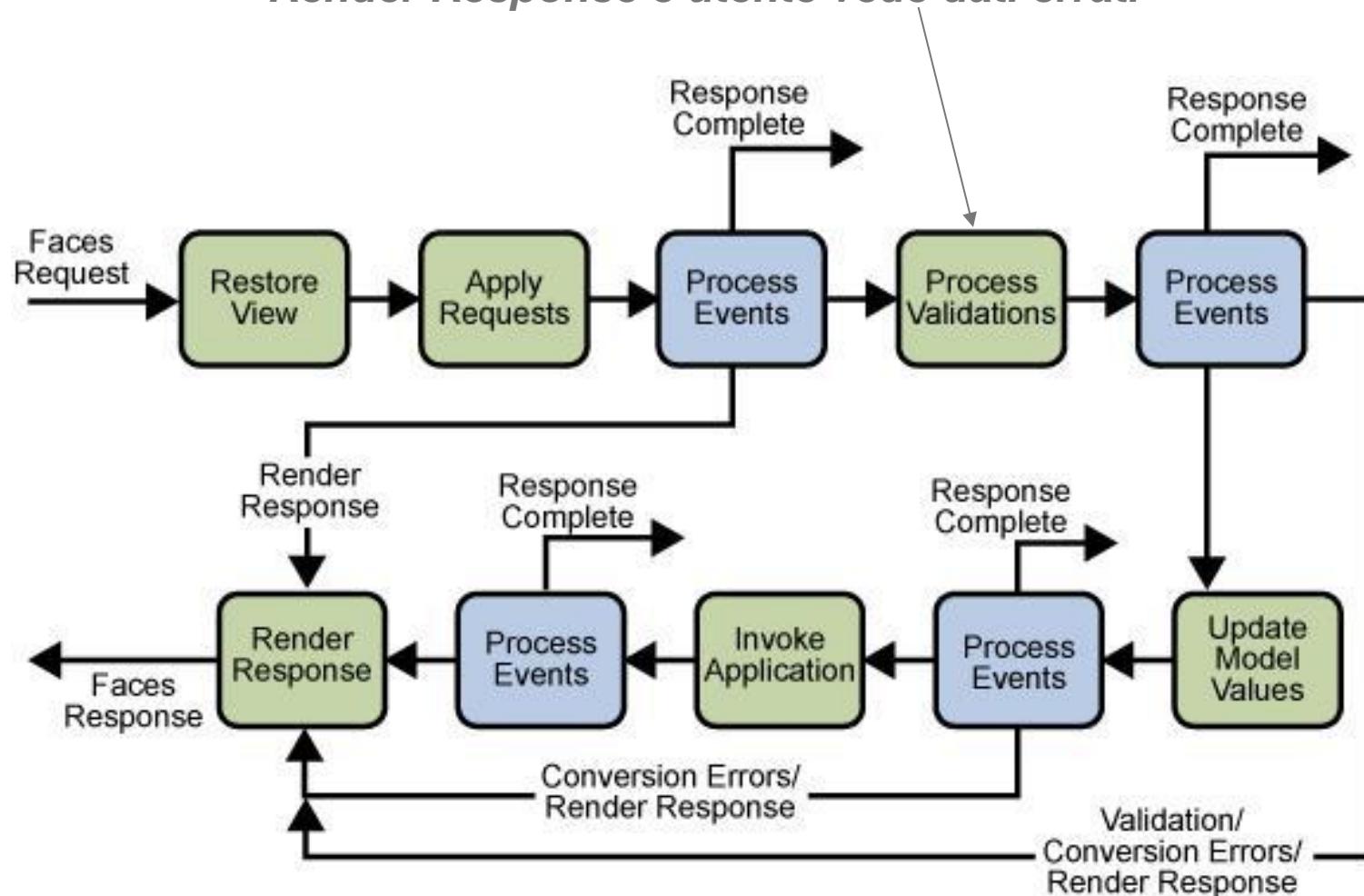
# Ciclo di vita JSF

*Recupera una vecchia o costruisce  
una nuova pagina (view)*

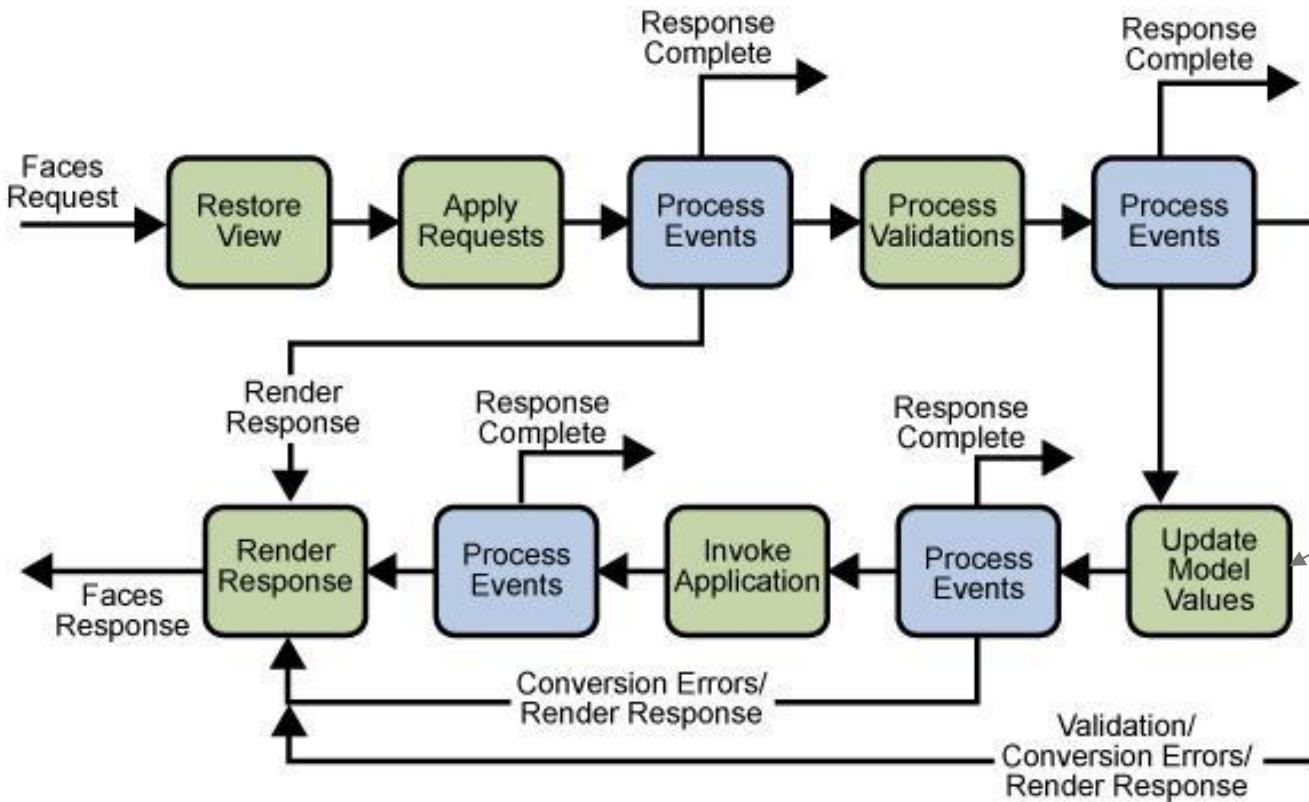


# Ciclo di vita JSF

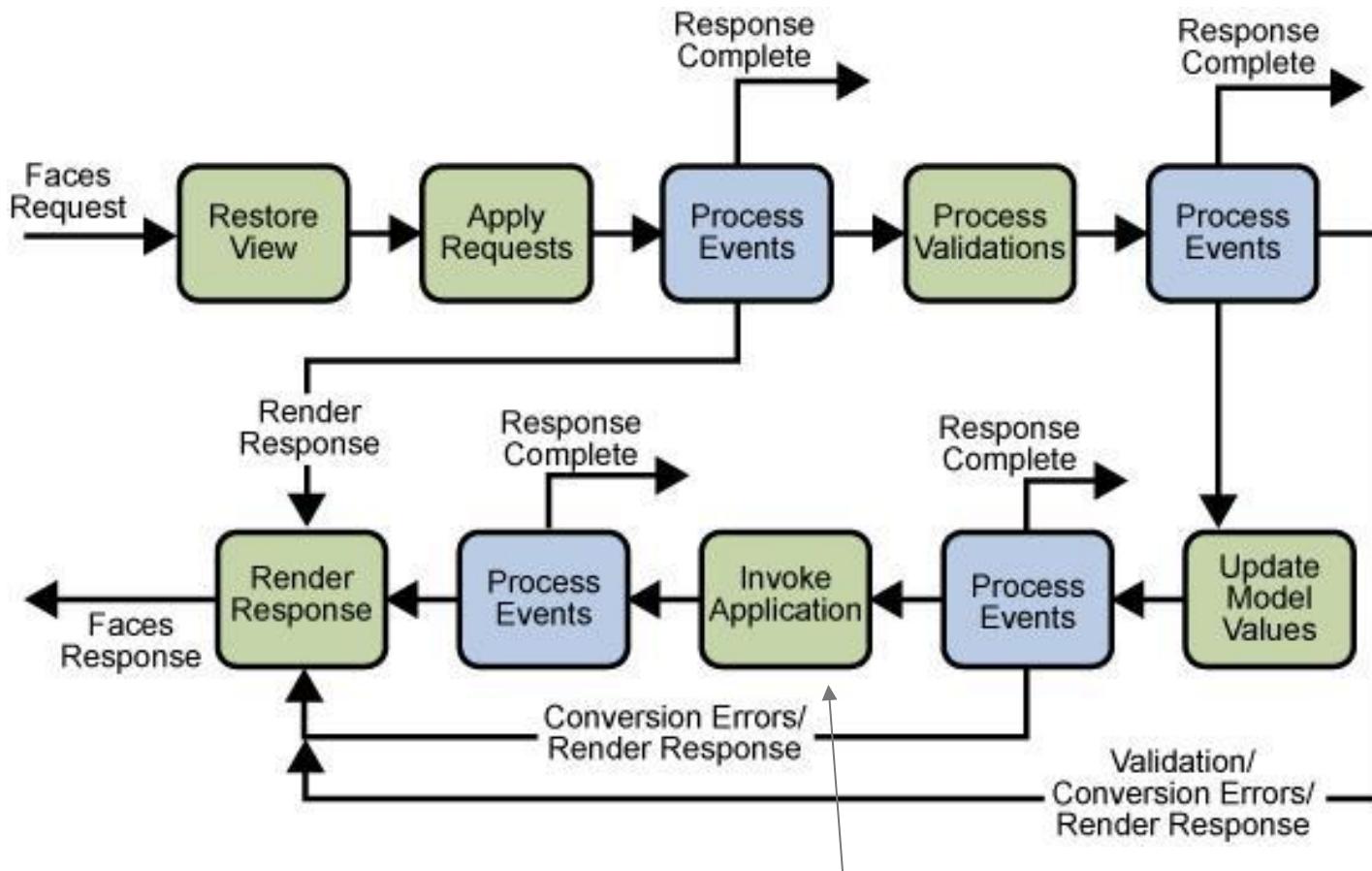
*Valori inviati sono memorizzati come “valori locali”.  
Se dati non validi o conversioni impossibili,  
Render Response e utente vede dati errati*



# Ciclo di vita JSF

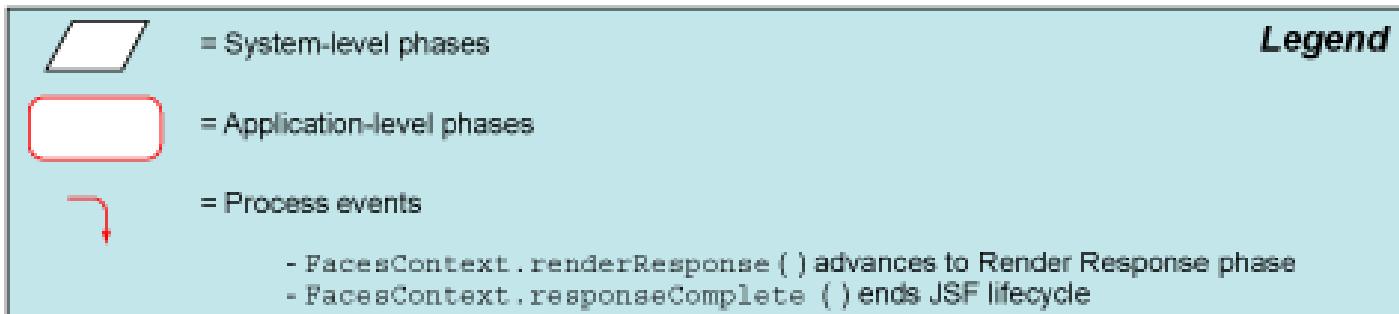
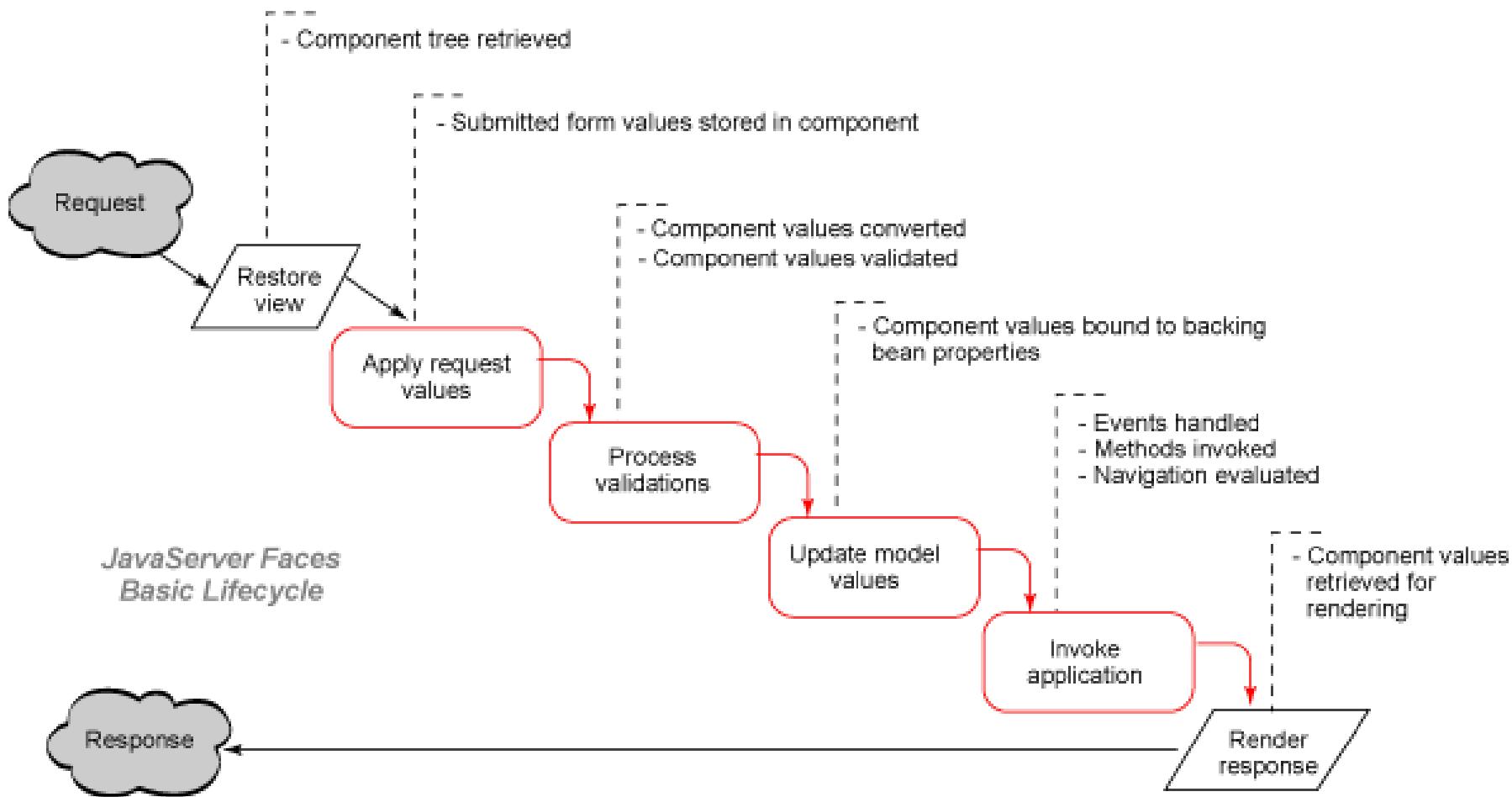


# Ciclo di vita JSF

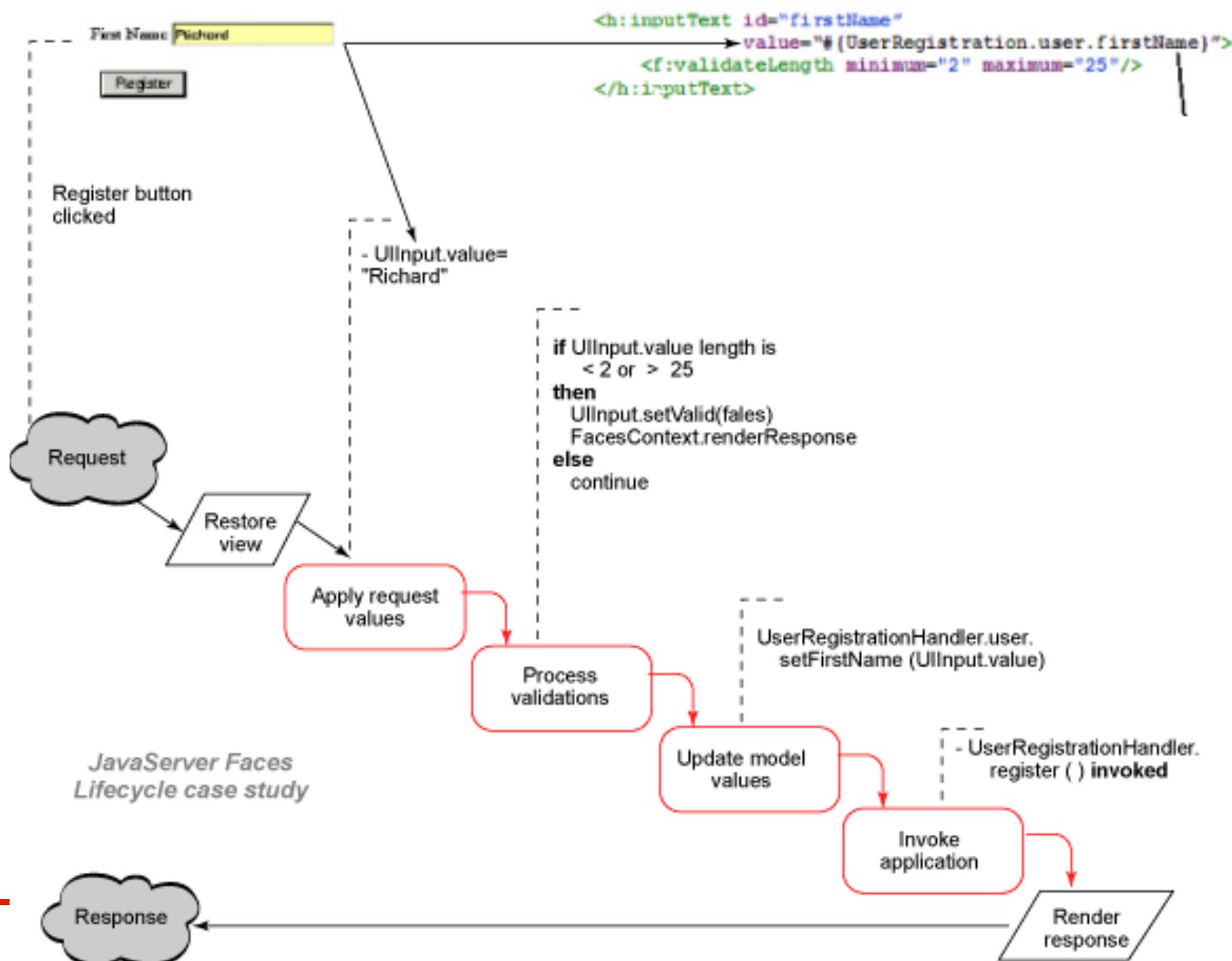


- *Action method associato con bottone o link che ha causato form submission viene eseguito*
- *Restituisce stringa per navigation handler*
- *Navigation handler usa stringa per determinare pagina seguente*

# Ciclo di vita JSF



# Ciclo di vita JSF (esempio)



# JSF Managed Bean

---

Configurati nella seconda parte di faces-config.xml

Semplici JavaBean che seguono regole standard:

- Costruttore senza argomenti (empty)
- No variabili di istanza public
- Metodi “accessor” per evitare accesso diretto a campi
- Metodi `getXxx()` e `setXxx()`

***JSF Managed Bean hanno anche metodi cosiddetti “action”***

- Invocati automaticam. in risposta ad ***azione utente o evento***
- Simili a classi Action in STRUTS

4 possibili scope:

- ***Application*** – singola istanza per applicazione
- ***Session*** – nuova istanza per ogni nuova sessione utente
- ***Request*** – nuova istanza per ogni richiesta
- ***Scopeless*** – acceduta anche da altri bean e soggetta a garbage collection come ogni oggetto Java

# Configurazione JSF Managed Bean

---

## Anche annotazioni equivalenti

```
<managed-bean>
    <managed-bean-name>library</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.model.Library
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>usersession</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.session.UserSession
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>loginform</managed-bean-name>
    <managed-bean-class>com.oreilly.jent.jsf.library.backing.LoginForm
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

# Un esempio un po' più realistico (1)

---

```
package guessNumber;
import java.util.Random;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped

public class UserNumberBean {
    Integer randomInt = null; Integer userNumber = null;
    String response = null; private long maximum=10;
    private long minimum=0;
    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(10));
        System.out.println("Numero estratto: " + randomInt);
    }
    public void setUserNumber(Integer user_number) {
        userNumber = user_number; }
    public Integer getUserNumber() {
        return userNumber; }
```

## Un esempio un po' più realistico (2)

---

```
public String getResponse() {  
    if ((userNumber != null) &&  
        (userNumber.compareTo(randomInt) == 0)) {  
        return "Indovinato!";  
    } else { return "Mi dispiace, "+userNumber+" non corretto"; }  
}  
  
public long getMaximum() {  
    return (this.maximum); }  
  
public void setMaximum(long maximum) {  
    this.maximum = maximum; }  
  
public long getMinimum() {  
    return (this.minimum); }  
  
public void setMinimum(long minimum) {  
    this.minimum = minimum; }  
}
```

L'annotazione `@SessionScoped` fa sì che lo scope del bean sia la sessione. Altre possibilità: `request`, `application`, `scopeless`

---

# Un esempio un po' più realistico (3)

---

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">

<h:head>
    <title>Guess Number Facelets Application</title>
</h:head>

<h:body> <h:form>
    <h:graphicImage value="#{resource['images:wave.med.gif']}'/>
    <h2> Sto pensando a un numero fra #{userNumberBean.minimum}
        e #{userNumberBean.maximum}. Vuoi indovinarlo? <p></p>
    <h:inputText id="userNo"
        value="#{userNumberBean.userNumber}">
        <f:validateLongRange minimum="#{userNumberBean.minimum}"
            maximum="#{userNumberBean.maximum}" />
    </h:inputText>
```

# Un esempio un po' più realistico (4)

---

```
<h:commandButton id="submit" value="Submit"  
    action="response.xhtml"/>  
  
<h:message showSummary="true" showDetail="false"  
    style="color: red; font-family: 'New Century Schoolbook',  
serif;  
    font-style: oblique; text-decoration: overline"  
id="errors1" for="userNo"/>  
</h2></h:form></h:body>
```

Notare:

- i tag HTML Facelets per *aggiungere componenti* alla pagina (cominciano con `h:`)
- il tag `f:validateLongRange` per validazione automatica dell'input utente

Inoltre, *utilizzo di funzionalità di navigazione implicita*, ridirezione della risposta verso `response.xhtml` (prossimo lucido)

---

# Un esempio un po' più realistico (5): response.xhtml

---

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>Guess Number Facelets Application</title>
</h:head>

<h:body><h:form>
<h:graphicImage value="#{resource['images:wave.med.gif']}"/>
<h2> <h:outputText id="result"
    value="#{userNumberBean.response}" /> </h2>
<h:commandButton id="back" value="Back" action="greeting.xhtml"/>
</h:form> </h:body>
</html>
```

# JSF e templating

---

Facilità di *estensione e riuso* come caratteristica generale di JSF

*Templating*: utilizzo di pagine come base (o *template*) per altre pagine, anche mantenendo look&feel uniforme

Ad esempio, tag:

- **ui:insert** – parte di un template in cui potrà essere inserito contenuto (tag di amplissimo utilizzo)
- **ui:component** – definisce un componente creato e aggiunto all'albero dei componenti
- **ui:define** – definisce contenuto con cui pagina “riempie” template (vedi insert)
- **ui: include** – incapsula e riutilizza contenuto per pagine multiple
- **ui: param** – per passare parametri a file incluso

# Esempio di template (template.xhtml)

---

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

<h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <link href=".//resources/css/default.css" rel="stylesheet" type="text/css"/>
    <link href=".//resources/css/cssLayout.css" rel="stylesheet" type="text/css"/>
<title>Facelets Template</title>
</h:head>
<h:body>
    <div id="top" class="top">
        <ui:insert name="top">Top Section</ui:insert></div>
    <div>
        <div id="left">
            <ui:insert name="left">Left Section</ui:insert></div>
        <div id="content" class="left_content">
            <ui:insert name="content">Main Content</ui:insert></div>
    </div></h:body></html>
```

# Esempio di pagina XHTML che utilizza template

---

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html">

<h:body>
    <ui:composition template=". /template.xhtml">
        <ui:define name="top"> Welcome to Template Client Page
        </ui:define>
        <ui:define name="left">
            <h:outputLabel value="You are in the Left Section"/>
        </ui:define>
        <ui:define name="content">
            <h:graphicImage value="#{resource['images:wave.med.gif']}"/>
            <h:outputText value="You are in the Main Content Section"/>
        </ui:define>
    </ui:composition>
</h:body></html>
```

# faces-config.xml

---

Necessità di file configurazione specifico per JSF: faces-config.xml

Soprattutto per configurazione *navigation rule e managed bean*

```
<?xml version='1.0' encoding='UTF-8'?>
...
<faces-config>
    <application>
        <locale-config>
            <default-locale>en</default-locale>
        </locale-config></application>
    <validator>
        <validator-id>ISBNValidator</validator-id>
        <validator-class>com.oreilly.jent.jsf.library.validator.
            ISBNValidator </validator-class> </validator>
    <navigation rule> ...
    <managed bean> ...
</faces-config>
```

# Navigation rule

---

Ogni regola di navigazione è come un *flowchart con un ingresso e uscite multiple possibili*

Un singolo <from-view-id> per fare match con URI

Quando restituito controllo, stringa risultato viene valutata (ad es. success, failure, verify, login)

- <from-outcome> deve fare match con stringa risultato
- <to-view-id> determina URI verso cui fare forwarding

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-action>#{LoginForm.login}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/storefront.xhtml</to-view-id>
    </navigation-case>

    <navigation-case>
        <from-action>#{LoginForm.logon}</from-action>
        <from-outcome>failure</from-outcome>
        <to-view-id>/logon.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Caso 1 di navigazione

Caso 2 di navigazione

## Ulteriori dettagli e manualistica

---

Per ulteriori dettagli su:

- Tag vari per componenti da inserire in pagine e loro attributi (ampio set supportato)
- Tag per *componenti command* per azioni di navigazione
- Core tag (*JSF core tag library*) per definire/usare *listener*, *converter* e *validator*

vedi

- D. Geary, C. Horstmann, “Core Java Server Faces”, Prentice Hall, 2007
- C. Schalk, E. Burns, “Java Server Faces – The Complete Reference”, McGraw-Hill, 2007
- Oracle, “Java EE 6 Tutorial”,  
<http://download.oracle.com/javaee/6/tutorial/doc/>, 2011



# WebSocket

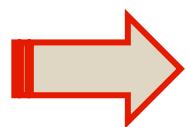
# Limiti HTTP «tradizionale» e Web Socket

---

A valle di quanto abbiamo visto nel corso...

*Limiti del modello di interazione HTTP quando abbiamo bisogno di usare **HTTP per comunicazione 2-way**:*

- Polling
- Long polling
- Streaming/forever response
- Connessioni multiple



*Idea di estensione (proprietaria) e tecnologia Web  
Socket*

Web Socket possono servire a migliorare sviluppo (più facile e «naturale») ed esecuzione runtime di applicazioni Web bidirezionali e *non strettamente request-response*

## Limiti HTTP «tradizionale» e Web Socket

---

Prima soluzione proprietaria integrata Javascript in alcuni browser (Google Chrome) e poi supportata da specifici Web server. Poi successo e sforzo di standardizzazione...

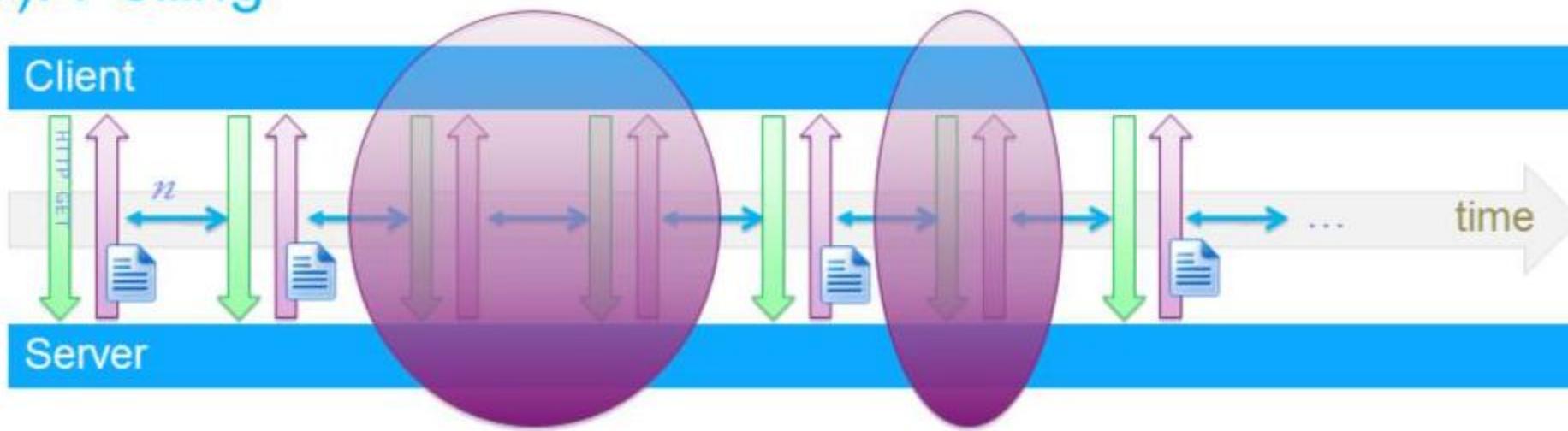
- Protocollo Web Socket (basato su TCP/IP) – RFC 6455
- Integrazione di Web Socket in HTML5 (via Javascript) e in JEE (a partire da v7)
- Web Socket API per Java definita in JSR 356

*Perché? Limiti del modello di interazione HTTP quando vogliamo usare HTTP per comunicazione 2-way:*

- Polling
- Long polling
- Streaming/forever response
- Connessioni multiple

# Polling

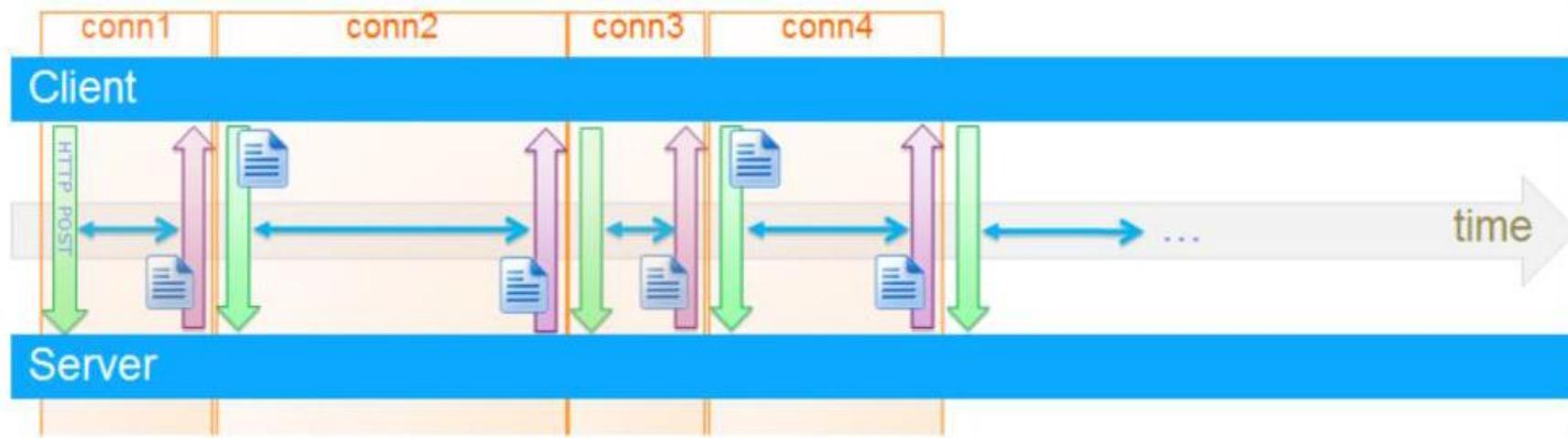
## (1): Polling



- Ad esempio realizzabile in Javascript
- **Cliente fa polling a intervalli prefissati e server risponde immediatamente**
- Soluzione ragionevole quando periodicità nota e costante
- Inefficiente ovviamente quando il server **NON** ha dati da trasferire

# Long Polling

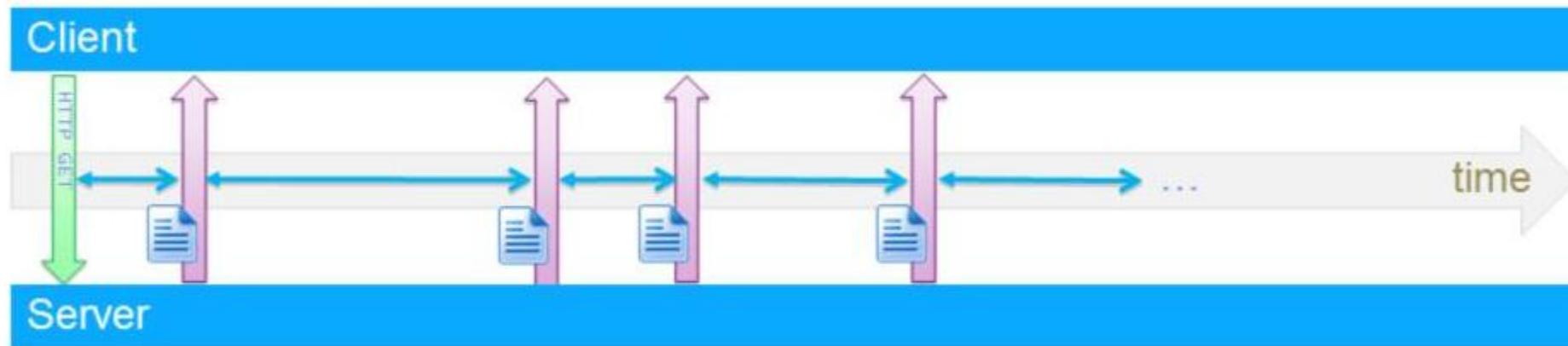
## (2): Long Polling



- Cliente manda *richiesta iniziale* e *server attende fino a che ha dati da inviare*
- Quando il cliente riceve risposta, reagisce *mandando immediatamente nuova richiesta*
- Ogni request/response si appoggia a nuova connessione

# Streaming/forever response

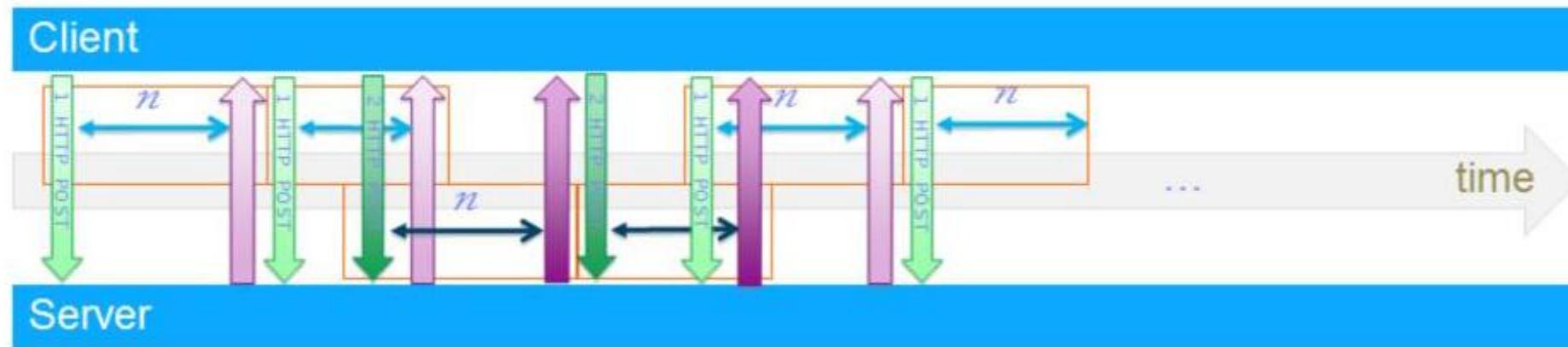
## (3): Streaming / forever response



- Cliente manda *richiesta iniziale* e *server attende fino a che ha dati da inviare*
- *Server risponde con streaming su una connessione mantenuta sempre aperta per aggiornamenti push (risposte parziali)*
- **Half-duplex** – solo server-to-client
- Proxy intermedi potrebbero essere in difficoltà con risposte parziali...

# Connessioni multiple

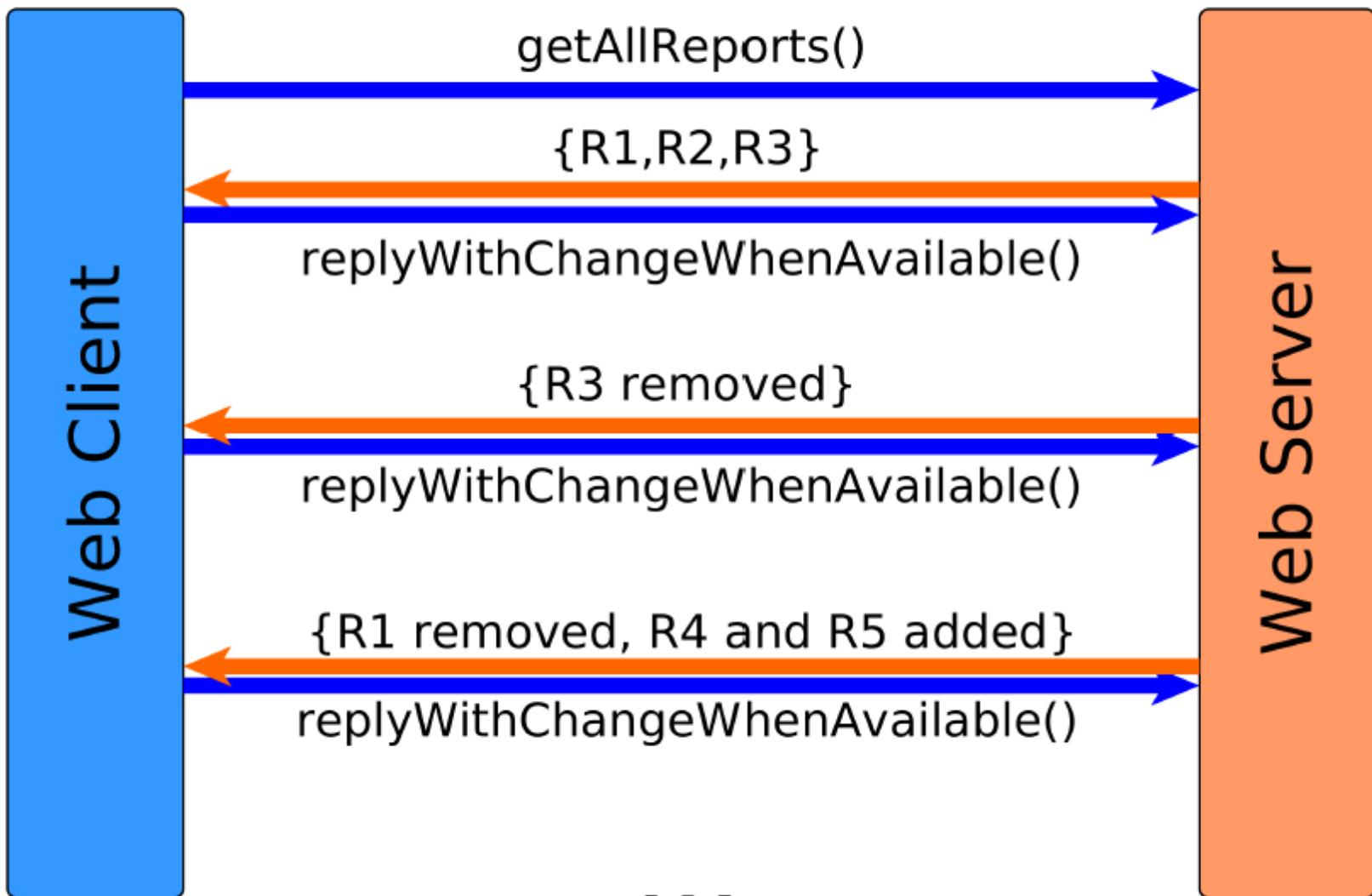
## (4): Multiple connections



- Long polling su due connessioni HTTP separate
  - Una per long polling «tradizionale»
  - Una per dati da cliente verso servitore
- Complesso coordinamento e gestione connessioni
- Overhead di due connessioni per ogni cliente

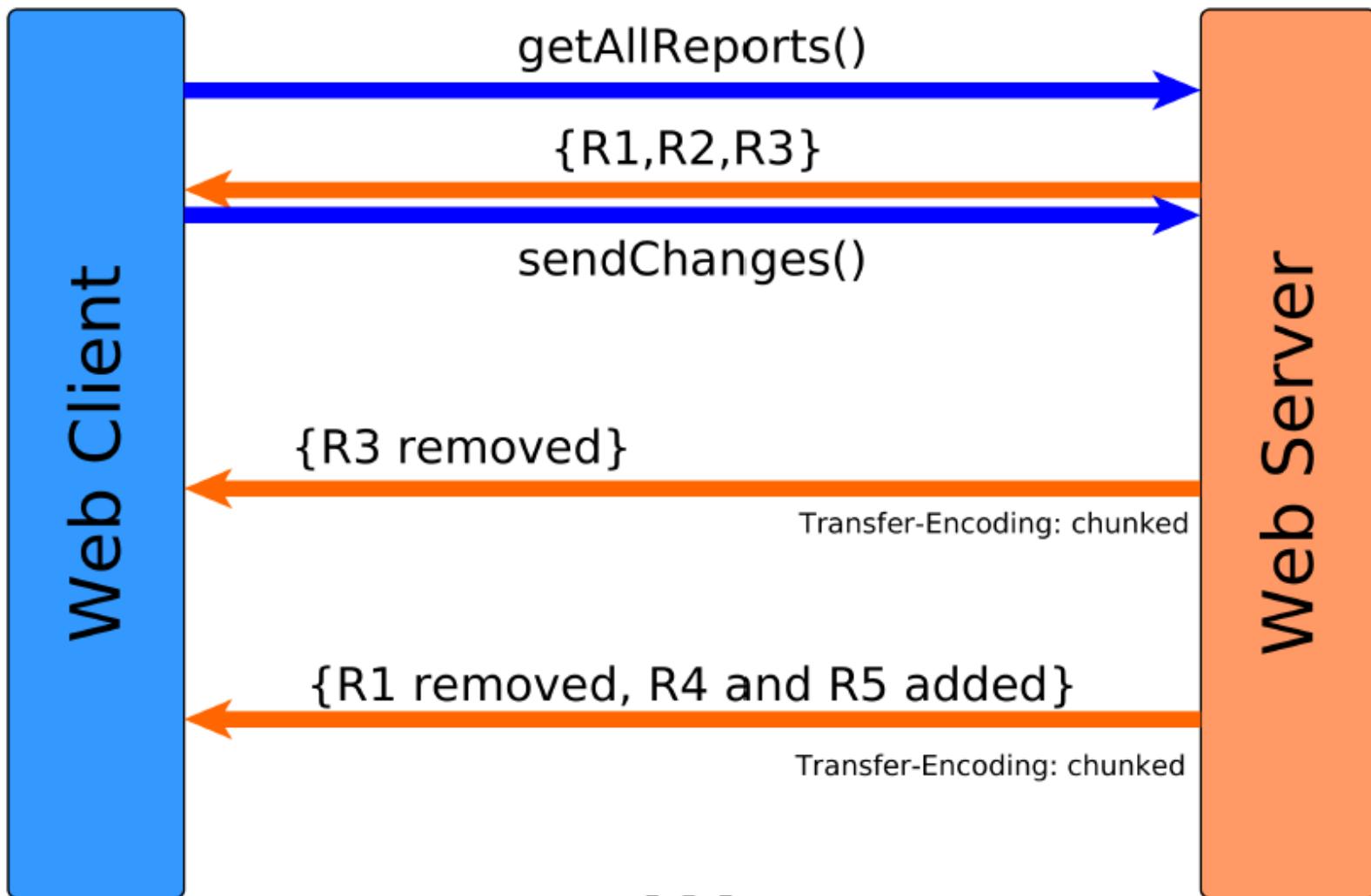
## Comportamento simile implementabile con AJAX...

Better Solution Using HTTP – AJAX push (Long Poll)



## Comportamento simile implementabile con AJAX...

Better Solution Using HTTP – AJAX push (Streaming)



# Web Socket: principali caratteristiche

---

## □ Bi-direzionali

- Client e server possono scambiarsi messaggi quando desiderano

## □ Full-duplex

- Nessun requisito di interazione solo come coppia request/response e di ordinamento messaggi

## □ Unica connessione long running

## □ Visto come «upgrade» di HTTP

- Nessuno sfruttamento di protocollo completamente nuovo, nessun bisogno di nuova «infrastruttura»

## □ Uso efficiente di banda e CPU

- Messaggi possono essere del tutto dedicati a dati applicativi

# Elementi base del protocollo

## □ Handshake

- Cliente comincia connessione e servitore risponde *accettando upgrade*

## □ Una volta stabilita connessione Web Socket

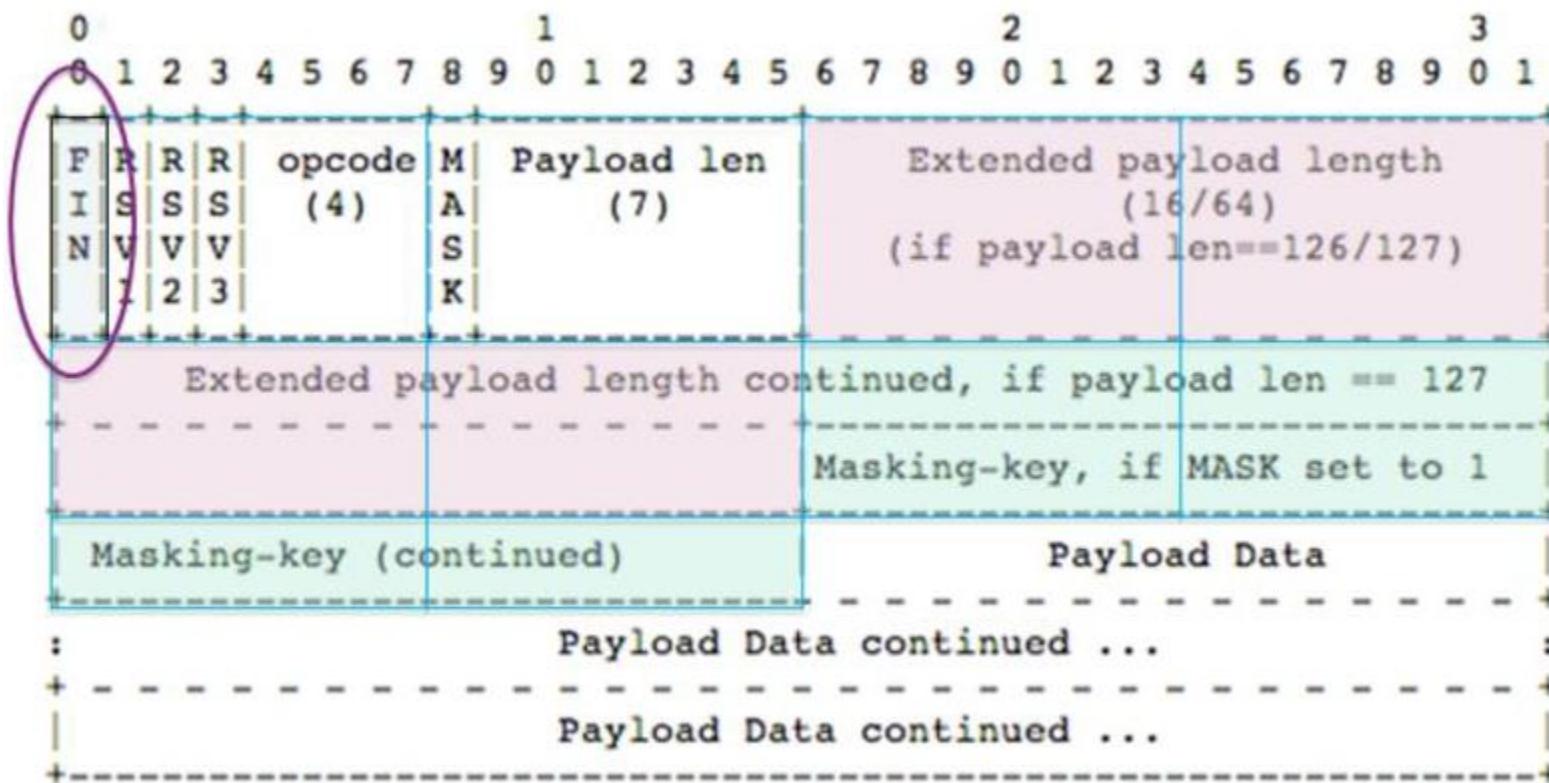
- Entrambi endpoint notificati che socket è aperta
- Entrambi endpoint possono *inviare messaggi e chiudere socket* in ogni istante

```
GET ws://server.org/wsendpoint
HTTP/1.1
Host: server.org
Connection: Upgrade
Upgrade: websocket
Origin: http://server.org
Sec-WebSocket-Version: 13
Sec-WebSocket-Key:
GhkZiCk+0/91FXIbUuRlVQ==
Sec-WebSocket-Extensions:
permessage-deflate;
client_max_window_bits
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept:
jpwu9a/SXDrsoRR26Oa3JUEFchY=
Sec-WebSocket-Extensions:
permessage-deflate;client_max_window
...
```

# Ottimizzazione messaggi

- Dati trasmessi con *minimo overhead in termini di header*
- Possibilità di *frammentazione in più frame* (un frame non può comunque ospitare più messaggi)



Approccio integrato con *Javascript lato cliente* e  
programmazione *JEE lato servitore* (uso di *annotazioni*)

Java API for WebSocket (JSR-356) lato servitore

- *Gestione ciclo di vita*
  - onOpen, onClose, onError
- *Comunicazione* tramite messaggi
  - onMessage, send
- Possibilità di uso di sessione
- Encoder e decoder per formattazione messaggi  
(messaggi  $\Leftrightarrow$  oggetti Java)

## Lato Server, a partire da JEEv7

---

```
@ServerEndpoint("/actions")
public class WebSocketServer {

    @OnOpen
    public void open(Session session) { ... }

    @OnClose
    public void close(Session session) { ... }

    @OnError
    public void onError(Throwable error) { ... }

    @OnMessage
    public void handleMessage(String message, Session session) {
        // actual message processing
    }
}
```

# Inviare e ricevere messaggi su una WebSocket in Java

---

Gli endpoint WebSocket possono inviare/ricevere messaggi sotto forma di testo o binary

## Invio:

- **Ottenere oggetto Session dalla connessione**

Disponibile come parametro in molti metodi. Ad esempio, nel metodo che ha ricevuto un messaggio (metodo annotato con `@OnMessage`); oppure, come variabile di istanza della classe endpoint nel metodo `@OnOpen`

- **Usare oggetto Session per ottenere un RemoteEndpoint**

`Session.getBasicRemote` e `Session.getAsyncRemote` restituiscono `RemoteEndpoint.Basic` e `RemoteEndpoint.Async` rispettivamente

- `void RemoteEndpoint.Basic.sendText(String text)`
- `void RemoteEndpoint.Basic.sendBinary(ByteBuffer data)`
- `void RemoteEndpoint.Basic.sendPing(ByteBuffer appData)`

# Inviare e ricevere messaggi su una WebSocket in Java

---

## Inviare messaggi a tutti i peer connessi a un Endpoint

Ogni istanza di classe endpoint class è normalmente associata con una connessione e un peer; tuttavia, è possibile anche associare una istanza a una pluralità di peer connessi, per esempio per applicazioni di chat

=> Uso dell'interfaccia Session e del metodo  
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) { ... } } }
```

# Inviare e ricevere messaggi su una WebSocket in Java

---

## Ricevere messaggi

Si possono avere al massimo 3 metodi annotati con `@OnMessage` in un endpoint, uno per ogni tipo di messaggio, ovvero text, binary e pong

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

# Mantenimento dello stato del cliente

---

Il container lato server crea una istanza della classe endpoint per ogni connessione; quindi **si possono usare variabili di istanza per salvare stato cliente**

Inoltre, il metodo **Session.getUserProperties** restituisce una modifiable map per memorizzare proprietà utente

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

Per info comuni a tutti i client, si possono anche utilizzare variabili di classe (**static**); in questo caso, responsabilità dello sviluppatore assicurare che accesso sia **thread-safe**

---

## Uso di encoder e decoder

---

- Java API per WebSocket forniscono supporto per conversion di messaggi WebSocket  $\leftrightarrow$  oggetti Java tramite encoder e decoder
- Automatizzazione dei processi di serializzazione e deserializzazione
  - Ad esempio, encoder tipici generano rappresentazioni JSON, XML, o binarie a partire da oggetti Java

# Uso di encoder

---

- **Implementare una di queste interfacce:**  
Encoder.Text<T> per messaggi testuali  
Encoder.Binary<T> per messaggi binary
- **Queste interfacce specificano il metodo di encode;** occorre implementare una encoder class per ogni tipo Java custom che si vuole inviare come messaggio WebSocket
- **Aggiungere il nome delle classi encoder al parametro opzionale della annotazione ServerEndpoint**
- **Usare il metodo sendObject(Object data) di RemoteEndpoint.Basic o di RemoteEndpoint.Async**  
Il container cerca un encoder che faccia match con il tipo e lo usa per la conversione verso un messaggio WebSocket

# Uso di encoder

---

Ad esempio, per inviare due tipi Java (MessageA e MessageB) come messaggi testuali:

```
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;  
    }  
}  
...
```

Similmente per MessageBTextEncoder

# Uso di encoder

---

```
...  
@ServerEndpoint(  
    value = "/myendpoint",  
    encoders = { MessageATextEncoder.class, MessageBTextEncoder.class  
}  
)  
public class EncEndpoint { ... }  
...  
MessageA msgA = new MessageA(...);  
MessageB msgB = new MessageB(...);  
session.getBasicRemote().sendObject(msgA);  
session.getBasicRemote().sendObject(msgB);
```

Come per gli endpoint, le istanze di encoder sono associate con una connessione e un peer WebSocket

=> Quindi c'è un solo thread ad eseguire il codice di una istanza di encoder ad ogni istante

# Uso di decoder

---

Si procede in modo analogo per i decoder; interfacce:

- Decoder.Text<T> per messaggi testuali
- Decoder.Binary<T> per messaggi binary

```
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) {  
        return canDecode;  
    } }
```

# Uso di decoder

---

```
@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...  
    }  
}
```

Come per gli endpoint, le istanze di decoder sono associate con una sola connessione e un solo peer WebSocket; quindi un solo thread esegue il codice di una istanza di decoder in ogni istante

---

# Lato browser cliente, integrazione Javascript

---

```
var socket = new WebSocket("ws://server.org/
  wsendpoint");
socket.onmessage = onMessage;

function onMessage(event) {
  var data = JSON.parse(event.data);
  if (data.action === "addMessage") {
    ...
    // actual message processing
  }
  if (data.action === "removeMessage") {
    ...
    // actual message processing
  }
}
```

# Web Socket API in Javascript

---

Costruttore:

**WebSocket(url[, protocols])**

Alcune proprietà principali:

- **WebSocket.bufferedAmount**  
sola lettura, numero di byte di dati accodati
  - **WebSocket.onclose**  
listener all'evento di chiusura della connessione
  - **WebSocket.onerror**  
listener all'evento di errore sull'uso della WebSocket
  - **WebSocket.onmessage**  
listener all'evento di ricezione di un messaggio dal server
  - **WebSocket.onopen**  
listener all'evento di connessione aperta
  - **WebSocket.protocol**  
sola lettura, sub-protocol selezionato dal servitore
-

# Web Socket API in Javascript

---

- ...
- WebSocket.readyState
  - sola lettura, stato corrente della connessione  
(`WebSocket.CONNECTING` 0, `WebSocket.OPEN` 1,  
`WebSocket.CLOSING` 2, `WebSocket.CLOSED` 3)
- `WebSocket.url`
  - sola lettura, URL assoluto associato

## Metodi

- WebSocket.close([code[, reason]])
  - chiude la connessione
- WebSocket.send(data)
  - accoda nuovi dati per l'invio

# Web Socket API in Javascript

---

## Eventi

Possibile agganciarsi a questi eventi usando `addEventListener()` o assegnando un event listener alla proprietà *onNomeEvento*

- **close**

Evento di chiusura connessione, anche disponibile tramite proprietà `onclose`

- **error**

Evento di errore che ha prodotto la chiusura di `WebSocket`, ad esempio con mancato invio di un dato; anche disponibile tramite proprietà `onerror`

- **message**

Evento associato alla ricezione di un messaggio dal server, anche disponibile tramite proprietà `onmessage`

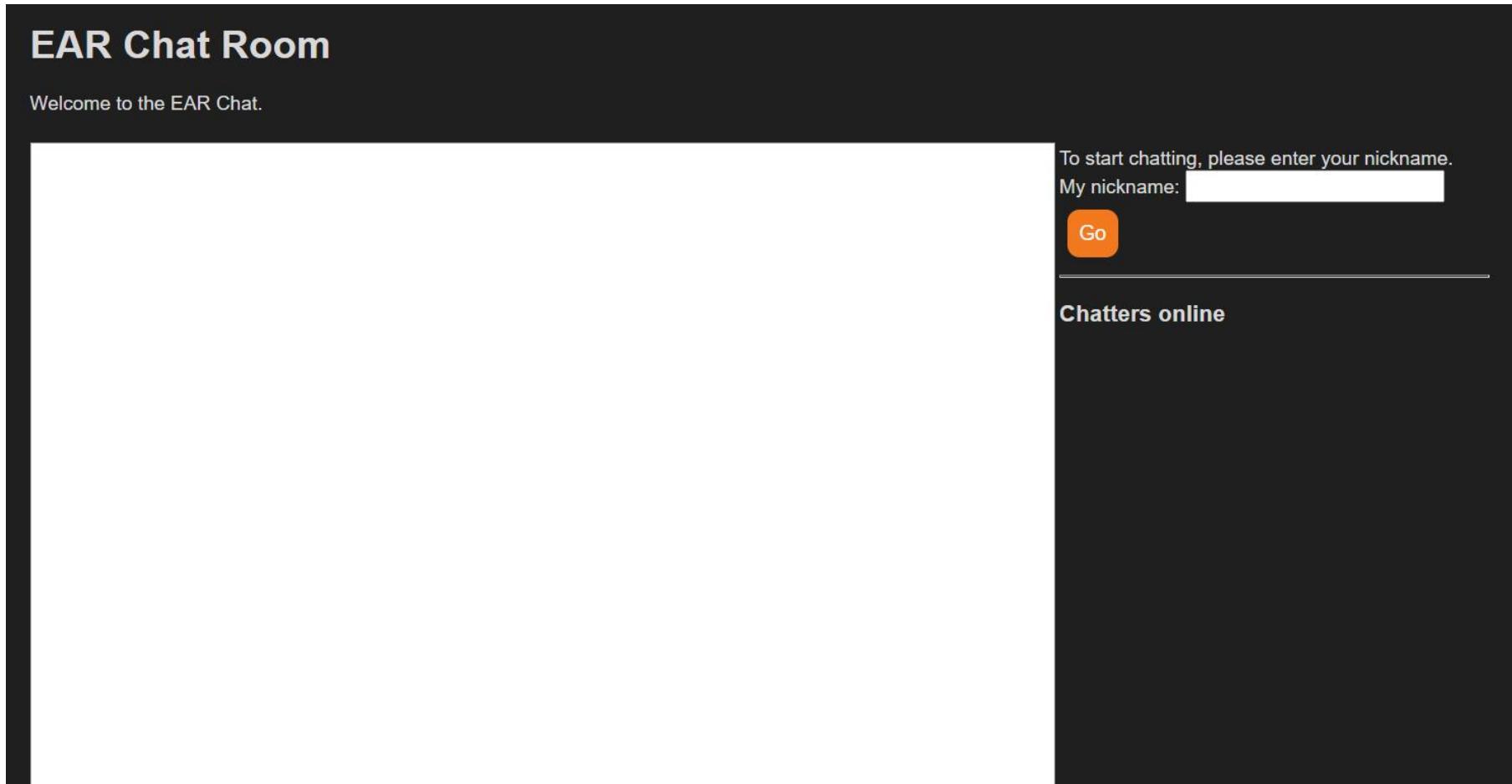
- **open**

Evento di apertura di una connessione `WebSocket`, anche disponibile tramite proprietà `onopen`

## Esempio di chat - <http://dev.inbas.cz:18080/chat/>

---

Chat come classico esempio di applicazione Web che beneficia di canale bidirezionale...



# Esempio di chat (parte cliente js) basata su WebSocket

---

```
var myChatterId;  
  
window.onload = init;  
var socket = new  
WebSocket("ws://dev.inbas.cz:18080/chat/actions");  
socket.onmessage = onMessage;  
  
function onMessage(event) {  
    var data = JSON.parse(event.data);  
    if (data.action === "addMessage") { printMessage(data); }  
    if (data.action === "addChatter") { printChatterElement(data); }  
    if (data.action === "setChatterId") { myChatterId = data.id;  
        printCurrentChatter(myChatterId); }  
    if (data.action === "removeChatter") {  
        document.getElementById("chatter"+data.id).remove();  
    }    }
```

## Esempio di chat basata su WebSocket (2)

---

```
function addMessage(text) {  
    var MessageAction = {  
        action: "addMessage", text: text, chatterId: myChatterId  
    };  
    socket.send(JSON.stringify(MessageAction)); }
```

```
function addChatter(nickName) {  
    var ChatterAction = { action: "addChatter", nickName: nickName,  
    };  
    socket.send(JSON.stringify(ChatterAction)); }
```

```
function removeChatter() {  
    var ChatterAction = { action: "removeChatter", id: myChatterId  
    };  
    socket.send(JSON.stringify(ChatterAction)); }
```

## Esempio di chat basata su WebSocket (3)

---

```
function printMessage(message) {  
    var content = document.getElementById("messages");  
    content.value = content.value + "(" + message.timestamp  
+ "," + message.chatterNickName + ") " + message.text + "\n"  
    content.scrollTop = content.scrollHeight;  
}  
  
function printCurrentChatter(chatterId) {  
    var eChatter = document.getElementById("chatter"+chatterId);  
    eChatter.setAttribute("class", "currentChatter");  
    var logout = document.createElement("button");  
    logout.setAttribute("id", "logout"+chatterId);  
    logout.setAttribute("class", "button");  
    logout.innerHTML = "Logout";  
    eChatter.appendChild(logout);
```

## Esempio di chat basata su WebSocket (4)

---

```
var eChatterForm = document.getElementById("addChatterForm");
  var eMessageForm =
document.getElementById("addMessageForm");
eChatterForm.hidden = true;
eMessageForm.hidden = false;
logout.onclick = function() {
  eChatterForm.hidden = false;
  eMessageForm.hidden = true;
  removeChatter(); };
}

function printChatterElement(chatter) {
  var content = document.getElementById("chatters");
  var chatterDiv = document.createElement("div");
  chatterDiv.setAttribute("id", "chatter"+chatter.id);
  content.appendChild(chatterDiv);
```

## Esempio di chat basata su WebSocket (5)

---

```
... var chatterNickName = document.createElement("span");
chatterNickName.setAttribute("class", "chatterNick");
chatterNickName.innerHTML = chatter.nickName;
chatterDiv.appendChild(chatterNickName); }

function addMessageSubmit() {
    var form = document.getElementById("addMessageForm");
    var text = form.elements["message_text"].value;
    document.getElementById("addMessageForm").reset();
    addMessage(text); }

function loginChatterSubmit() {
    var form = document.getElementById("addChatterForm");
    var nickName = form.elements["chatter_nickname"].value;
    document.getElementById("addChatterForm").reset();
    addChatter(nickName); }
```

## Esempio di chat basata su WebSocket (6)

---

```
function init() {  
    console.log("Initializing EAR Chat application");  
    console.log(document.getElementById("chatters").childNodes);  
    myChatterId = undefined;  
    document.getElementById("chatters").innerHTML="";  
    console.log(document.getElementById("chatters").childNodes);  
    document.getElementById("messages").childNodes=[];  
}  
}
```

## Altre opzioni di integrazione per WebSocket

---

- Spring offre ampio support per WebSocket tramite annotazioni custom
  - **spring-websocket module**
- React.js ha un suo modulo separato per integrazione con WebSocket
  - **react-websocket module** (contiene listener per eventi relativi a WebSocket)

# Riferimenti

---

- **RFC 6455 – il protocollo delle WebSocket**
  - <https://tools.ietf.org/html/rfc6455>
- **JSR 356: Java API for WebSocket**
  - <https://jcp.org/en/jsr/detail?id=356>
- **Java EE 7: costruire applicazioni Web basate su WebSocket, JavaScript e HTML5**
  - <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/HomeWebsocket/WebsocketHome.html>
- **Supporto Spring a WebSocket**
  - <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html>