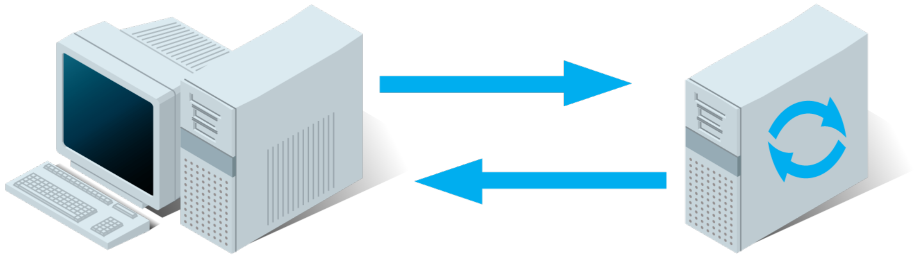
Server Push, WebSocket and JSF

The beginning

The original World Wide Web was designed for static content

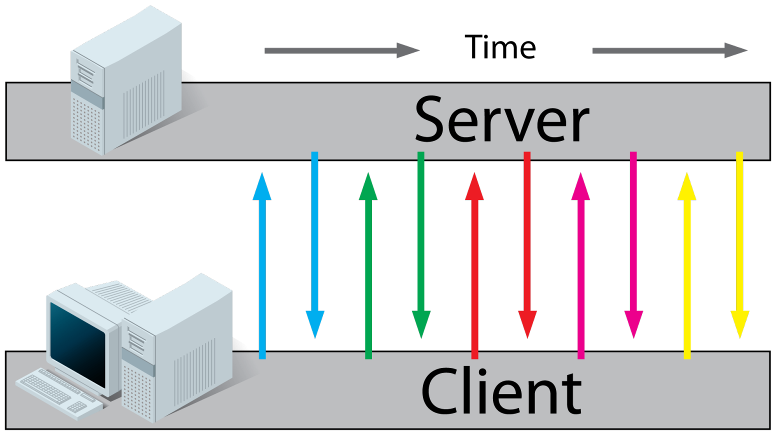


- A client requests a resource from the server via HTTP
- The server builds a response and sends it back
- Somewhere along the way the need for dynamic updates arose

Polling

In the beginning sites used the refresh meta tag

```
<meta http-equiv="refresh" content="5″ />
```



- Client requests the resource
- Server builds it and responds
- Extremely inefficient – creating massive amounts of traffic

Polling continued

Polling can also be achieved with JavaScript

```
setTimeout(function() {
    document.location.reload();
}, 5000);
```

- Same problems as with meta refreshing – generates way too much traffic and reloads the whole page
- Can we still use polling but somehow improve performance?

Polling continued

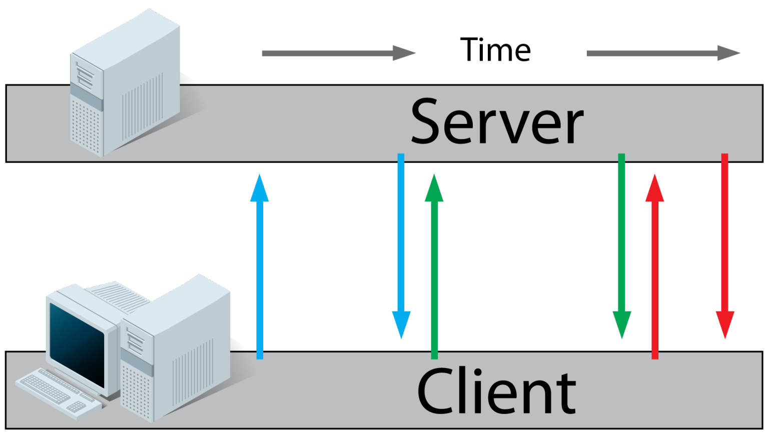
Originally introduced by Microsoft in 1999 (Explorer 5), we can use AJAX to improve performance

```
setInterval(function() {
    $('#mydiv').load(url);
}, 5000);
```

- Massively reduces traffic as we only have to updated part of the page
- Will still cause unnecessary page requests and traffic!
- Can we improve upon it?

Long polling

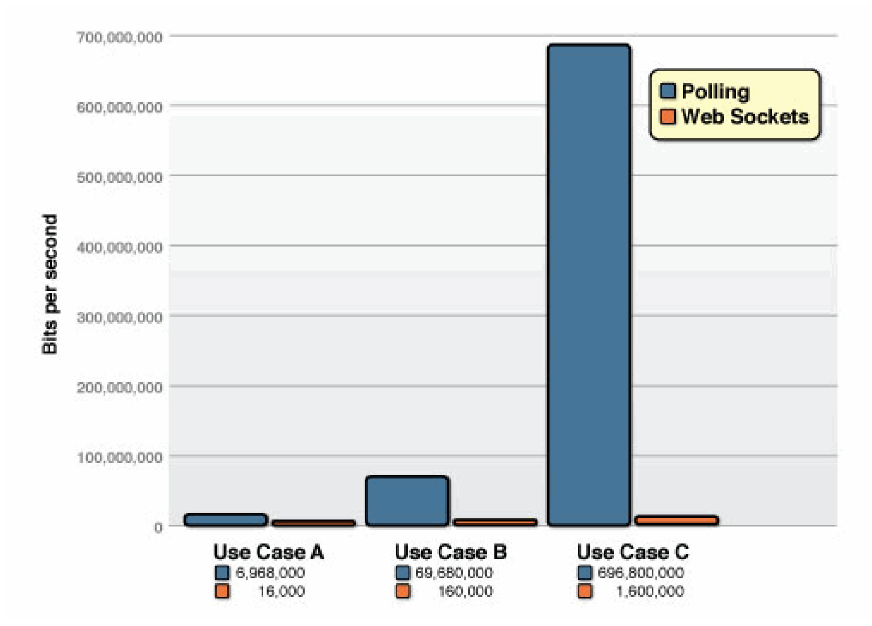
An alternative method is to rely on delayed server responses - commonly known as long polling



- Server holds the connection open by delaying the response
- The response is sent as soon as it is available
- This works quite well, so what’s the downside?

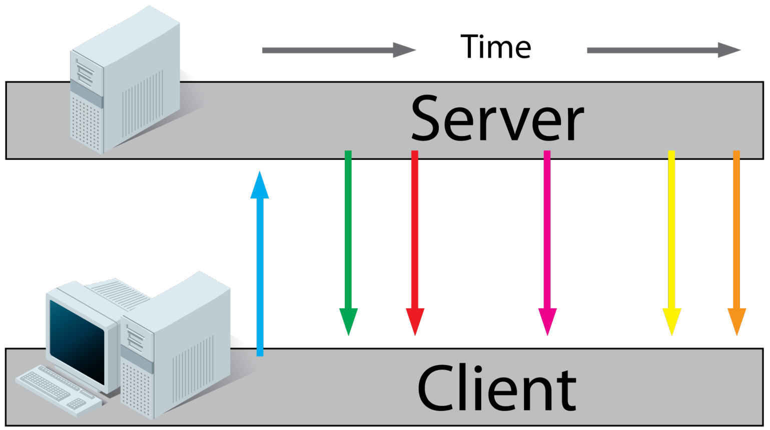
Long polling continued

Compared to web sockets, long polling still stresses bandwidth, memory and CPU consumption of the server



Server Sent Events

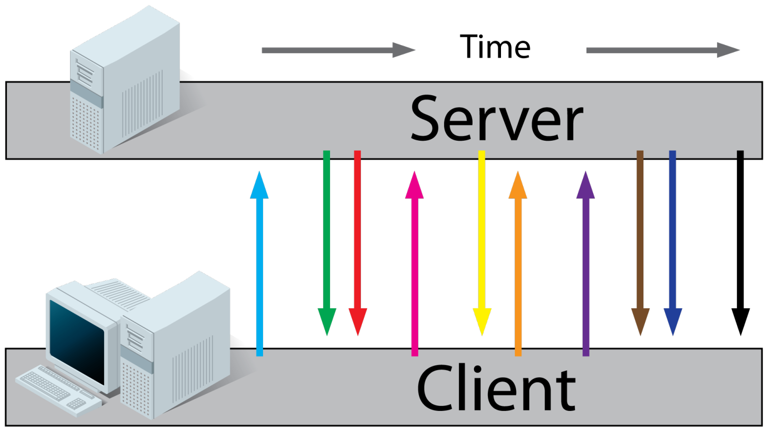
First HTML5-standardized implementation for notifications and dynamic updates



- The client regitsers as an event listener via the *EventSource API*
- Server sends an event packet when new data is available
- One way communication
- Limited connectivity

WebSocket

Finalized in 2011, the WebSocket protocol is also part of the HTML5 standard



- Got usable and widely adopted at the end of 2013.
- It took just over 20 years, but we can *finally* do proper dynamic page updates
- Two-way communication with great connectivity

WebSocket and JavaEE

The Atmosphere Framework – Push functionality for the enterprise

- The most popular asynchronous application development framework for enterprise Java.
- Provides everything required to build massive scalable and real time applications
- Fully configurable and clusterable
- Essential before Java Server Faces 2.3
- Falls back from WebSocket on failure

Pushing from Java Server Faces

Full support for WebSocket since **Java Server Faces 2.3** *(Java EE 8)* and **Servlet 3.1** *(Java EE 7)*

- Based on the implementation of **`o:socket`** from OmniFaces.
- JSF component frameworks no longer have to rely on custom implementations or Atmosphere
- Extremely easy to use

Pushing from JSF continued

Preparing the view

- We use the standard JSF component **`f:websocket`** to define the behaviour of the websocket channel
- *Attribute: **channel**, Name of the websocket channel*
- *Attribute: **scope**, Can define either “application”, “session” or “view”*
- *Attribute: **user**, Used to target a specific user*
- *Attributes: **onopen / onclose / onmessage**, allows us to call JavaScript functions during each event*
- *Can be combined with an* **f:ajax** *tag*

Pushing from JSF continued

Preparing the server

- Using CDI inside a bean or a EJB service we can ***@Push @Inject*** a PushContext instance on the server-side
- With *pushContext.send(Object)*, we can send messages to the clients listening on a specific channel
- With *pushContext.send(Object, ...users)*, we can target users on that channel

First example:

A simple message board

What do we need?

What do we need to make a simple message board in JSF with WebSocket support?

- Just plain JSF without any component framework is enough for a simple demonstration
- We need a model where the submitted messages can be stored
- A display of all the submitted messages
- A button and an input field for submitting new messages

On the server

Let’s define our server side

```
@Named @ViewScoped public class
MsgBackingBean implements Serializable {
    @EJB private MessageService msgService;
    @Inject @Push private PushContext incoming;

    @Getter @Setter
    private String enteredMessage;

    public List<String> getMessages() {
        return msgService.messages;
    }

    public void onSendMessage() {
        MsgService.add(enteredMessage);
        incoming.send(“new-message”);
    }
}
```

```
@Data @Singleton
public class MessageService {
    private List<String> messages;

    @PostConstruct
    private void init() {
        messages = new ArrayList<>();
    }

    public void add(String message) {
        messages.add(message);
    }
}
```

Some considerations

- Why are we using a *@ViewScoped* bean ?
- Why is the service seperated into an EJB? What benefits will that give us?
- Why does *@PostConstruct* even exist and why do we need it?

On the client

Let’s define our server side

```
<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>Message board</title>
    </h:head>
    <h:body>
        <h:form>
            <h:inputText value="#{msgBackingBean.enteredMessage}"/>
            <h:commandButton action="#{msgBackingBean.onSendMessage}"/>
            <f:websocket channel="incoming">
                <f:ajax event="new-message" process=”@form” render="@form"/>
            </f:websocket>
            <ui:repeat value="#{msgBackingBean.messages}" var="msg">
                <h:outputText value="#{msg}" />
            </ui:repeat>
        </h:form>
    </h:body>
</html>
```

One more thing to consider

- What do the **render** and **process** attributes on the ajax tag actually do ?

Second example:

A chat service

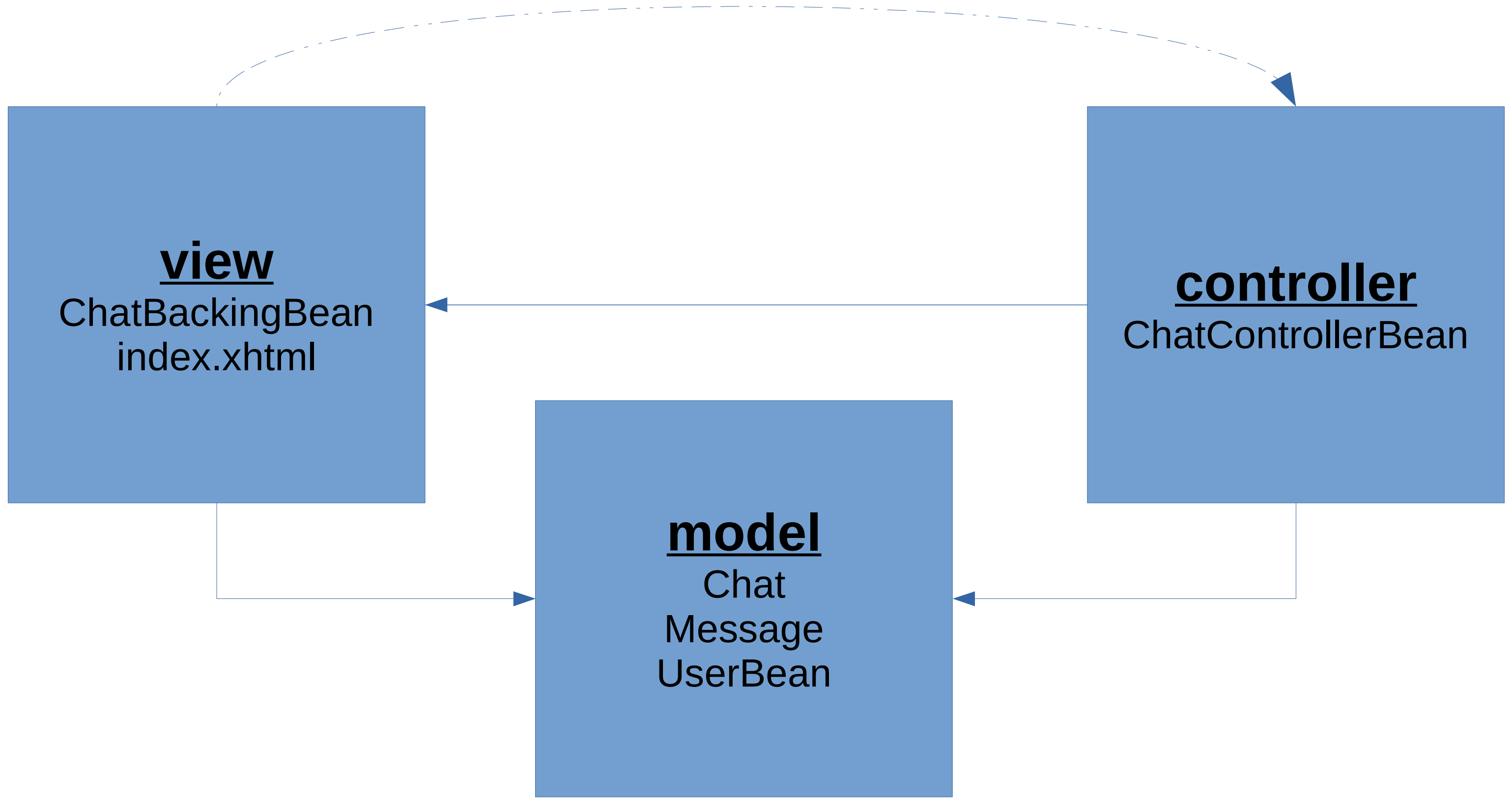
Let’s get a little more fancy!

We can demonstrate JSF and get a little more fancy by using some external component libraries

- PrimeFaces
- PrimeFaces Extensions
- The rest is vanilla JSF and Java EE!

Lets do something we can run and discuss during the lecture!

Our design


view
ChatBackingBean
index.xhtml
controller
ChatControllerBean
model
Chat
Message
UserBean

Visit the application while we code!

You can view the running instance of the application on

http://88.131.213.111:8080