Websockets

Get the web up to speed!

Van statische HTML pagina's, via dynamisch gegenereerde pagina's, naar AJAX en vervolgens naar Single Page Applications (SPA): dit is in vogelvlucht de historie van web applicaties. Daarbij valt op dat er steeds meer view logica naar de cliënt verschuift en dat de pagina's steeds interactiever worden. Maar om ten volle gebruik te kunnen maken van deze evolutie, zal er een oplossing moeten worden gevonden voor het probleem dat het HTTP-protocol niet erg geschikt is voor zeer interactieve applicaties. Websockets bieden een oplossing voor dit probleem. Het doel van dit artikel is te laten zien hoe websockets op het Java platform gebruikt kunnen worden met het uitkomen van Java EE 7. Daarvoor wordt eerst getoond wat het nut is van websockets om daarna de overstap te maken naar het Java platform.

Toegevoegde waarde van websockets

Het HTTP-protocol is oorspronkelijk ontworpen voor het opvragen van resources. Dit opvragen gebeurde dan vanuit de browser en de server gaf het bijbehorende resultaat terug. Geen wonder dus dat het HTTP-protocol message-georienteerd is en dat het initiatief steeds bij de client ligt. Tussen de client en server worden steeds kleine berichtjes verstuurd. Hoewel deze berichtjes klein zijn is er toch behoorlijk veel overhead (HTTP-headers) aanwezig. Voor veel applicaties is dit meestal voldoende maar voor zeer interactieve applicaties, waarbij veel berichtjes verstuurd worden, kan dit een probleem zijn. Een ander probleem is dat bij deze communicatie alleen de client het initiatief kan nemen. Als er data bijgewerkt moet worden op het scherm moet de client actie ondernemen om te 'pollen' of er nieuwe data beschikbaar gekomen is. In het verleden zijn er allerlei 'oplossingen' bedacht om ook de server data te kunnen laten 'pushen'. Deze oplossingen waren in veel gevallen nog steeds op het HTTP-protocol gebaseerd. Daarom is er gezocht of er naast het HTTP-protocol ook volgens een ander protocol gecommuniceerd kan worden. Daarvoor is een nieuw protocol bedacht: websockets. Communiceren met websockets maakt geheel nieuwe web-gebaseerde toepassingen mogelijk op gebied van games, realtime applicaties, social media en samenwerking. Daar waar het HTTP-protocol message-gebaseerd is, zijn web-sockets stream-gebaseerd. Om websockets wat beter te begrijpen zoomen we wat dieper in op de interne werking van websockets.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Listing 1: Handshake request (bron: W3C web-socket specificatie)

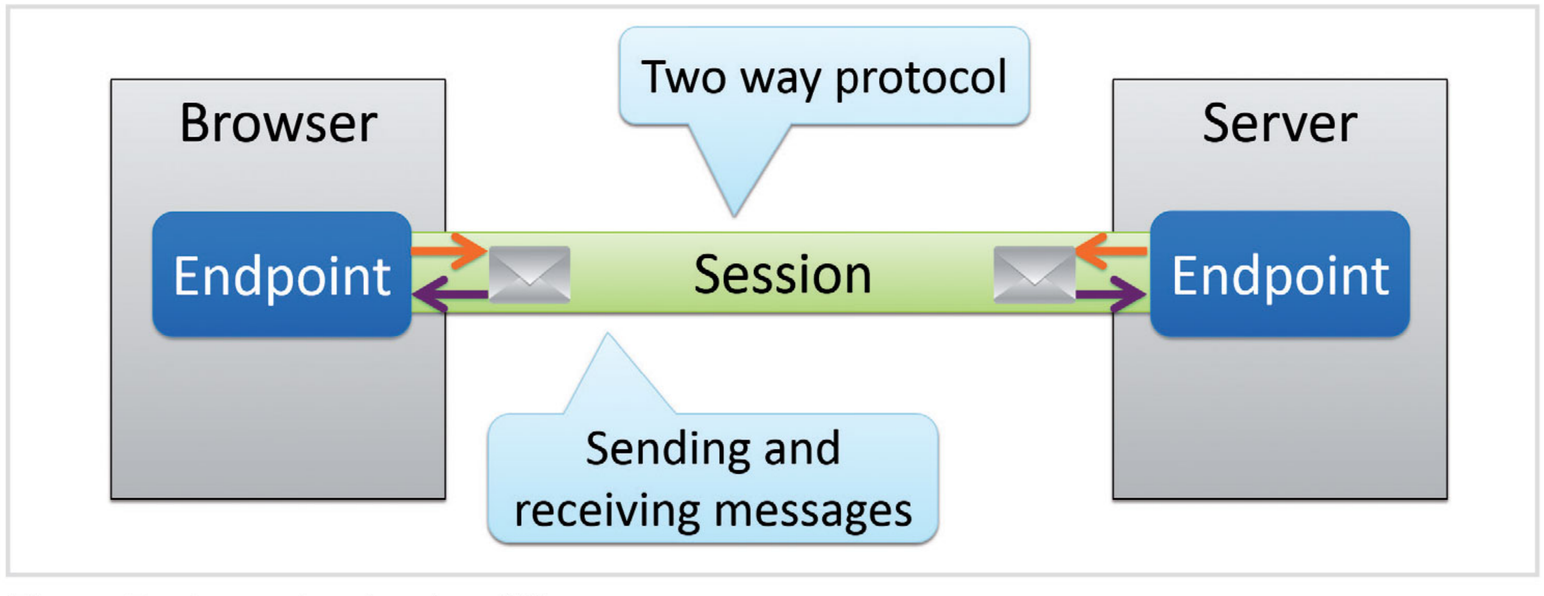
```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Listing 2: Handshake response (bron: W3C web-socket specificatie)

Werking van websockets

Eén van de grote voordelen van websockets is dat de communicatie via dezelfde poorten verloopt als het HTTP-protocol. Dat heeft tot gevolg dat infrastructuur componenten zoals routers minder vergaand aangepast hoeven te worden. Maar hoe is het dan duidelijk of communicatie van het type http of van het type websocket is? Elke websocket sessie start vanuit de client met een HTTP-request. In dit bericht vraagt de client aan de server om verder te mogen communiceren via websockets (**listing 1**). Indien dit door de server geaccepteerd wordt stuurt deze een bericht terug (**listing 2**).Dit proces wordt de handshake genoemd. Na het response bericht van de server zal elke verdere communicatie niet meer volgens http verlopen maar volgens het websocket protocol. Communiceren volgens het websocket protocol wil zeggen dat er simultaan

Teun Hakvoort is fulltime trainer en Lead developer bij Info Support en geeft training in Java technologie (JavaSE, Java EE, JSF, Spring) en 'browser' technologie zoals GWT, JavaScript en AngularJS.



Figuur 1 Begrippen uit websockets API

tweerichtingsverkeer plaats kan vinden tussen de client en de server. Bijkomend voordeel is dat een HTTP-request minimaal zo'n 40 bytes vereist door de headers die steeds meegestuurd worden. Vergelijk dit met de minimale 2 bytes die nodig is om volgens websockets te communiceren. Websockets moeten ondersteund worden door zowel de client (bijvoorbeeld de browser) als door de server. Wat de browsers betreft, ondersteunen de meeste moderne HTML5 browsers het protocol. Wat de server betreft hangt dit af van de achterliggende techniek. In Java kan in ieder geval al gebruik gemaakt worden van deze technologie.

Websockets in Java

Sinds het uitkomen van JSR 356 is er een standaard specificatie voor het gebruik van websockets op het Java platform. Deze specificatie is sinds maart 2013 final geworden en is onderdeel van Java EE 7. Dit betekent dat deze technologie op Java EE 7 applicatieservers (vooralsnog alleen Glassfish 4), Tomcat 8 (RC), Jetty 9 al te gebruiken is. De reference implementatie is geleverd door het project Tyrus. In de API spelen een aantal begrippen een belangrijke rol. Deze zijn schematisch weergegeven in **figuur 1**. Bij websocket communicatie is er sprake van een endpoints: een endpoint is een kant van de communicatie en zowel server als client hebben dus een endpoint. De communicatie verloopt via een connection waarover messages verstuurd en ontvangen worden via het websocket protocol. De term session wordt gebruikt om de communicatie tussen het ene endpoint en het andere endpoint aan te geven. Vanuit JavaScript kan eenvoudig met behulp van een JavaScript object een websocket verbinding opgezet worden (**listing 3**) naar een bepaald endpoint. Wordt in HTTP altijd het schema http(s) in de url gebruikt, voor websockets is dit ws(s). Met een simpel Java object kan er een server implementatie gemaakt worden. **Listing 4** laat zien hoe met een paar annotaties verteld wordt welke methode aangeroepen moet worden bij een binnenkomend bericht. Deze annotaties komen uit de websockets-API en zijn ondergebracht in de package javax.net.websocket. Boven de klasse SayHelloServer is aangegeven dat het een WebSocket endpoint betreft en onder welke URL deze te benaderen is. Met annotaties kan ingehaakt worden op de lifecycle van websockets. Zo zijn de @OnOpen en @OnClose annotaties beschikbaar om in te haken op het openen en sluiten van een WebSocket en is er een @OnError voor foutafhandeling. De API is flexibel opgezet. Het gebruik van annotaties is namelijk niet verplicht. Er kan ook gekozen worden om gebruik te maken van overerving of het implementeren van interfaces. Listing 4 is vrijwel het meest eenvoudige voorbeeld van websockets. Hoe kan de server berichtjes sturen naar de client? Daarom duiken we wat dieper de websocket-API in.

Initiatief vanuit de server

Om berichtjes naar alle verbonden clients te kunnen sturen moet er een lijst bijgehouden worden van Session-objecten. Het Session-

▸▸

```
var ws = new WebSocket("ws://localhost:8080/app/hello");
ws.onopen = function()  {
    ws.send('NLJUG');
};
ws.onmessage = function (evt) {
   console.write(evt.data); //print 'Hello NLJUG'
};
```

Listing: 3 Open Websocket verbinding vanuit JavaScript

```
import javax.net.websocket.*;

@WebSocketEndpoint("/hello")
public class SayHelloServer {

  @WebSocketMessage
  public String sayHello(String name) {
    return "Hello " + name;
  }
}
```

Listing: 4 Hello world voorbeeld in Java

object representeert de verbinding tussen twee endpoints en kan ten allen tijde als argument meegegeven laten worden. Dit lijstje met sessions mag als attribuut in de JFallChatService klasse (uit **listing 5**) opgeslagen worden omdat een containerproces één instantie van een WebSocketEndpoint aanmaakt (net zoals standaard bij Servlets). Listing 5 toont een deel van de implementatie van een eenvoudige chat service die bezoekers van JFall onderlinge ervaringen laat uitwisselen. Berichtjes die vanaf clients naar deze chats toegestuurd worden, worden direct doorgestuurd naar alle andere verbonden clients. Elke websocket-sessie wordt bij het openen middels de onOpen(…) methode opgeslagen in een lijstje met openstaande sessies. Dit lijstje wordt in de onMessage(…) methode gebruikt om het binnenkomend bericht naar alle andere endpoints toe te sturen. Dit bericht is in het codevoorbeeld van het type String. Dit kan ook binaire data zijn door gebruik te maken van de methode sendBinary(ByteBuffer data). Een Java desktop applicatie die wil communiceren met een WebSocket kan een serializable Java object serializeren naar een byte array en dit versturen. In listing 5 wordt er synchroon een bericht naar alle andere clients gestuurd. Door een kleine aanpassing kan dit ook asynchroon gebeuren: othersession.getAsyncRemote(). sendString(msg). In bovenstaande voorbeelden was de url van het endpoint steeds statisch. Net als bij JAX-RS is het ook mogelijk om placeholders hierin op te nemen. Bijvoorbeeld de url ws://localhost:8080/jfall/2013 bevat verschillende variabelen. De ServerEndpoint annotatie kan placeholders bevatten: @ServerEndpoint("/jfall/{year}"), die vervolgens in de onMessage(…) methode als argumenten binnenkomen: public void open(…, @PathParam("year") String year).

Client API

Toen de eerste versie van de JAX-RS specificatie uitkwam, bleek al snel dat deze één groot nadeel had: namelijk het ontbreken van een client-API. Bij websockets is deze direct in de 1.0 versie toegevoegd. Werd in codevoorbeeld 3 een websocket vanuit JavaScript aangeroepen, in Java is dit ook mogelijk. Dit werkt client-side bijna op precies dezelfde manier als serverside. Wederom wordt de keus gegeven om met annotaties of overerving te werken. Codevoorbeeld 6 (nu met overerving) laat zien hoe de client zich registreert als message handler en elk binnenkomend bericht beantwoordt. De klasse JFallChatServiceClient wordt geïnstantieerd door de websocketcontainer. Codevoorbeeld 7 laat zien hoe vanaf de client verbinding gemaakt wordt met de remote websocket.

```
@WebSocketEndpoint(path="/jfall/chat")
public class JFallChatService {
  Set<Session> sessions = new HashSet<>(); //not threadsafe

  @WebSocketOpen
  public void onOpen(Session session) {
     sessions.add(session);
  }

  @WebSocketMessage
  public void onMessage(String msg, Session session) throws IOException
{
      for (Session othersession : sessions) {
        if (othersession != session)
          othersession.getBasicRemote().sendString(msg);
      }
  }
```

Codevoorbeeld 5: Gebruik van de cascade operator

```
@ClientEndpoint
public class JFallChatServiceClient extends Endpoint {

  public void onOpen (Session session, EndpointConfig config) {

   final RemoteEndpoint.Basic remote = session.getBasicRemote();
   session.addMessageHandler (new MessageHandler.Whole<String>() {
      public void onMessage(String msg) {
        System.out.println("From JFall: " msg);
        remote.sendString("Hello NLJUGGERS!"); //exception handling
ignored
      }
   });
  }
  ...
```

Codevoorbeeld 6: Gebruik van de client API

Volwassenheid

In de API is er sterk rekening mee gehouden dat de API zo flexibel mogelijk moet zijn. Met behulp van ServerEndpointConfig.Configurator object zijn er veel aspecten van het handshake proces in te stellen. In de codevoorbeelden 4 en 5 zijn de messages van het type String. Dit kunnen ook eigen typen zijn die met behulp van Decoders en Encoders ge(de)codeerd worden. Wederom wordt in de @ServerEndpoint annotatie informatie opgenomen met betrekking tot encoders. @ServerEndpoint(…, encoders = { JFallMessageEncoder.class }). Na het opgeven van de te gebruiken encoders kan de onMessage methode direct een message van het type JFallMessageEncoding binnen krijgen. De API werkt goed samen met belangrijke andere specificaties zoals CDI en EJB. Zo is het geen probleem om een EJB Timerbean periodiek een CDI event te laten versturen die vervolgens met behulp van @Observe door een websocket opgevangen wordt en op basis daarvan alle openstaande sessions (clients) bijwerkt worden met nieuwe data. Authenticatie vindt plaats wanneer de communicatie nog via het HTTP-protocol verloopt: bij de handshake. Daarom kan het beveiligingsmechanisme van websockets aansluiten op het

standaard beveiligingsmechanisme van servlets. Dat wil zeggen in web.xml kunnen security-constraints opgenomen worden die gecombineerd kunnen worden met een login-config. Tevens kan het gebruik van versleuteling afgedwongen worden met behulp van user-data-constraints.

Conclusie

Met websockets wordt door de steeds meer eisende applicaties in een behoefte voorzien. Gezien de steeds bredere ondersteuning op zowel client als server zal de interactiviteit van applicaties nog verder toenemen. Met de komst van de final release van JSR 356 heeft Java volwaardige beschikking over websockets ondersteuning. Voor een eerste versie van de specificatie is deze verrassend compleet. Zowel een server- als client API is beschikbaar en beide zijn flexibel en werken intuïtief. Ook de integratie met andere specificaties zoals CDI, EJB en Servlets is goed geïmplementeerd. Helaas zal het nog wel even op zich laten wachten voordat de grote spelers een Java EE 7 applicatieserver in de aanbieding hebben. Tot die tijd kan er al uitgebreid gewerkt worden met bijvoorbeeld Glassfish 4 – of met de reference-implementatie die ook standalone te gebruiken is. Kortom voldoende mogelijkheden om hiermee aan de slag te gaan! ■

```
import javax.websocket.*;
WebSocketContainer container = ContainerProvider.getWebSocketContainer();
String url = “ws://localhost:8080/nljug/jfall”;
container.conntectToServer(JFallChatServiceClient.class, new URI(url));
```

Codevoorbeeld 7: Connecten naar een web-socket