Prof. Dr. Th. Letschert

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

NVP – Nebenläufige und Verteilte Programme

Aufgabenblatt 9

Aufgabe 1

Ab Java 7 enthält das package *java.nio.file* das Interface *WatchService* für systemabhängige Dienste zur Beobachtung von Änderungen im Dateisystem. Das Tutorial *Watching a Directory for Changes*[1] erläutert deren Verwendung.

Das Vorgehen ist dabei grob folgendes: Man wartet auf ein Ereignis, z.B. die Erzeugung einer Datei in einem bestimmten Verzeichnis und reagiert dann auf dieses Ereignis mit einer bestimmten Aktion. Ein auf das Notwendigste reduziertes Beispiel ist:

```
import java.nio.file.{FileSystems, Path, Paths}
import java.nio.file.StandardWatchEventKinds.ENTRY_CREATE

object FileWatch_App extends App {

  val dirS = scala.io.StdIn.readLine()
  val dirPath = Paths.get(dirS)
  val watcher = FileSystems.getDefault().newWatchService()
  val key = dirPath.register(watcher, ENTRY_CREATE)

  val keyWatched = watcher.take()
  val events = keyWatched.pollEvents()

  if (!events.isEmpty()) {
    val event = events.get(0)
    println(s"observerd creation of ${event.context}")
  }
}
```

Hier wird auf das Erzeugen einer neuen Datei in einem einzugebenden Verzeichnis gewartet und deren Name dann ausgegeben.

Machen Sie Sache asynchron und passen Sie den Stil Ihrer Lösung an Futures an. Ihre Lösung könnte dann etwa so zu nutzen sein:

```
object FileWatch_App extends App {
  val dirS = scala.io.StdIn.readLine()

  val fileCreation = WaitForFile(dirS)

  fileCreation onComplete {
    case Success(path) => println(s"there is a new file $path")
    case Failure(t) => println(s"failed because of : $t")
  }

  Thread.sleep(100000)
}
```

[1] http://docs.oracle.com/javase/tutorial/essential/io/notification.html

Hinweis: Verwenden Sie ein *Promise* zur Erzeugung des *Futures*.

Aufgabe 2

HTTP In seiner Konzeption ist HTTP ein reines *Request–Response*–Protokoll. Der Client fordert eine Webseite an, der Server liefert sie. Fertig, oder das Gleiche noch einmal.

Ajax Mit AJAX kam ca. 2005 eine Technologie auf, mit der das Neu-Laden ganzer Seiten durch inkrementelle Updates vermieden werden kann. Das Grundprinzip der HTTP-Kommunikation wurde damit aber nicht verändert, das Prinzip *Request–Response* blieb: der Client fordert an, der Server liefert. Nun neuerdings nicht eine ganze Seite, sondern nur neue Inhalte, die der Client in seine Seite, den DOM, einbauen kann. Das macht *Single-Page*-Anwendungen möglich. Ein erster kleiner Schritt in Richtung Applikationen mit "Web-Frontend".

HTTP mit seiner asymmetrischen *Request–Response* Kommunikation zwischen Frontend (View) und Backend (Model und Controller) ist bei jeder Anwendung mit einer halbwegs anspruchsvollen Benutzerinteraktion völlig unangemessen. Um trotzdem Applikationen mit Web-Frontend realisieren zu können, wurden etliche technologischen Krücken eingeführt, auf die wir hier nicht weiter eingehen wollen, mit denen aber insgesamt auf einem symmetrischen verbindungsorientierten Protokoll, TCP, ein zustandsloses Request-Response-Protokoll, HTTP, realisieren wurde und auf diesem dann mit allerlei Tricks und Kniffen die Zustandsorientierung und symmetrische Kommunikation wieder hinzu gemurkst wurde, die HTTP vorher eliminiert hatte.

Ca. 2010 hatte man dann endlich ein Einsehen und akzeptierte, dass HTTP nicht unantastbar ist und man tatsächlich offen und ohne Tricks verbindungsorientiert sein darf.

Websockets Eine symmetrische Kommunikation zwischen Frontend und Backend ist in vielen Fällen eine unumgängliche Notwendigkeit und mit den *Websockets* wurde eine Technologie eingeführt., die dies unterstützt. Grob gesagt ist eine Websocket-Verbindung eine TCP-Verbindung, die mit HTTP aufgebaut wird. Die Spezifikation des Websocket-Protokolls findet sich in RFC-6455[2].

Ein Websocket-Client oder -Server kann in jeder Programmiersprache, die Socket-Kommunikation in irgendeiner Art unterstützt, geschrieben werden. Hilfreich sind dabei Bibliotheken, die Funktionalität von RFC-6455 abdecken. Noch besser ist es, wenn die Schnittstelle dieser Bibliotheken normiert ist, so dass eine Implementierung problemlos gegen andere ausgetauscht werden kann.

Für jede relevante Programmiersprache gibt es Implementierungen des Websocket-Protokolls. Für Java wird in JSR-356[3] eine API für eine RFC-6455-Implementierung spezifiziert. Für diese API-Spezifikation gibt es diverse Implementierungen. Leider ist keine davon Bestandteil von Java-SE. Nur Java-EE enthält im Package `javax.websocket` eine Implementierung der Websocket-API. Das ist nicht erstaunlich, denn JSR-356 spezifiziert explizit eine Servertechnologie, d.h. es wird vorausgesetzt, dass eine Java-EE-Server zur Verfügung steht, um die Funktionalität umzusetzen.

Websocket-Client (HTML / JavaScript) Für HTML/JavaScript wird durch die API-Spezifikation *The WebSocket API*[4] des W3C ein Zugriff auf Dienste entsprechend RFC-6455 geregelt. Browser-Implementierungen realisieren diese API-Spezifiktion. Deren exakte Umsetzung findet man in der Dokumentation des Browsers. Siehe beispielsweise die Dokumentation von Mozilla.[5]

Der HTML-Client einer Echo-Applikation als Demo ist:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>WebSockets Demo</title>
</head>
<body>
```

[2] https://tools.ietf.org/html/rfc6455
[3] http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html
[4] https://www.w3.org/TR/2012/CR-websockets-20120920/
[5] https://developer.mozilla.org/de/docs/WebSockets

```
<h1>WebSockets Demo</h1>
<div id="status">Connecting...</div>
<ul id="messages"></ul>
<form id="message-form" action="#" method="post">
   <textarea id="message" placeholder="Write your message here..."
       required></textarea>
   <button type="submit">Send Message</button>
   <button type="button" id="close">Close Connection</button>
</form>
<script type="text/javascript">
   window.onload = function() {
      var form = document.getElementById('message-form');
      var messageField = document.getElementById('message');
      var messagesList = document.getElementById('messages');
      var socketStatus = document.getElementById('status');
      var closeBtn = document.getElementById('close');

      var socket = new WebSocket('ws://127.0.0.1:4711/websockets/echo');

      socket.onopen = function(event) {
         socketStatus.innerHTML = 'Connected to: ' + event.currentTarget.url;
      };
      socket.onmessage = function(event) {
         var message = event.data;
         messagesList.innerHTML += '<li class="received">Received: ' + message +
             '</li>';
      };
      socket.onclose = function(event) {
         socketStatus.innerHTML = 'Disconnected from WebSocket.';
      };
      socket.onerror = function(error) {
         console.log('WebSocket Error: ' + error);
      };

      form.onsubmit = function(e) {
         var message = messageField.value;
         socket.send(message);
         messagesList.innerHTML += '<li class="sent">Sent: ' + message + '</li>';
         messageField.value = '';
         return false;
      };
      closeBtn.onclick = function(e) {
         socket.close();
         return false;
      };
   };
</script>
</body>
</html>
```

Ein Websocket-Client muss natürlich nicht zwingend ein Browser sein. Für Java-Clients existiert mit JSR-356 eine API-Spezifikation, die wie *The WebSocket API* eine Schnittstelle zu Kommuniationsdiensten entsprechend RFC-6455 beschreibt. Eine Implementierung dieser API ist ebenfalls Bestandteil von Java-EE.

Websocket-Server (Java / Scala) Die Referenz-Implementierung von JSR-356 wird von Tyrus.[6] bereitgestellt. Die notwendigen Jar-Dateien werden über Maven zur Verfügung gestellt.[7] Die Informationen hier beruhen im Wesentlichen auf der Dokumentation Tyrus[8]

[6] https://tyrus.java.net
[7] https://mvnrepository.com/artifact/javax.websocket
[8] https://tyrus.java.net/documentation/latest/index/ siehe auch https://abhirockzz.gitbooks.io/java-websocket-api-handbook/content/

In einem SBT-Projekt löst man die Abhängigkeit in `build.sbt` mit

```
libraryDependencies += "javax.websocket" % "javax.websocket-api" % "1.1"
libraryDependencies += "org.glassfish.tyrus" % "tyrus-server" % "1.13.1"
libraryDependencies += "org.glassfish.tyrus" % "tyrus-container-grizzly-server" %
    "1.13.1"
```

Man sieht an diesen Abhängigkeiten, dass es sich um Java-EE handelt: die Verwendung der Websockets auf der Serverseite setzt einen Java-EE-Server voraus. Hier wird konsequenterweise mit *Glassfish*[9] die Referenz-Implementierung eines Java-EE-Servers verwendet. Der Server wird mit diesem Gehilfen im Rücken damit recht einfach:

```
import java.io.BufferedReader
import java.io.InputStreamReader
import org.glassfish.tyrus.server.Server
import javax.websocket.server.ServerEndpoint
import javax.websocket.{Endpoint, EndpointConfig, Session, OnMessage, OnOpen, OnClose,
    OnError}

@ServerEndpoint(value = "/echo")
class EchoEndpoint {

  @OnMessage
  def onMessage(message: String, session: Session): String = {
    println("received message " + message)
    message
  }
  @OnOpen
  def onOpen(session: Session, config: EndpointConfig): Unit = {
    println("new Session " + session.getId)
  }
  @OnError
  def onError(t: Throwable): Unit = {
    t.printStackTrace()
  }
  @OnClose
  def onClose(session: Session): Unit = {
    println("session " + session.getId + " is closed")
  }
}

object WebSocketServer {
  def main(args: Array[String]): Unit = {
    val server = new Server("localhost", 4711, "/websockets", null, classOf[EchoEndpoint])
    try {
      server.start()
      val reader = new BufferedReader(new InputStreamReader(System.in))
      System.out.print("Please press a key to stop the server.")
      reader.readLine
    } catch {
      case e: Exception =>
        throw new RuntimeException(e)
    } finally server.stop()
  }
}
```

Der Glassfish ist ein Java-EE-Server. Er hat beispielsweise die Aufgabe per *Dependency Injection* die Annotationen (z.B. `@ServerEndpoint`) zu verarbeiten. Ein Java-EE-Server ist ein Webserver und *Container*: Ein Applikation wird in den Container geworfen, der sie dann untersucht, mit seinen Diensten verknüpft und ausführt. Es gibt inzwischen viele

[9] https://glassfish.java.net

alternative Lösungen. *Jetty*[10] und *Vert.x* [11] sind bekannte “leichtgewichtigere” Alternativen.

Asynchrone Berechnung im Server Der Server kann natürlich auch asynchrone Berechnungen starten. Damit die Antwort an den richtigen Partner gesendet werden kann, muss die zu der Anfrage gehörende *Session* festgehalten werden:

```
import ...
import scala.concurrent.{ExecutionContext, Future}
import ExecutionContext.Implicits.global)

@ServerEndpoint(value = "/echo")
class EchoEndpoint {

  @OnMessage
  def onMessage(message: String, session: Session): String = {
    println("received message " + message)
    val endpoint = session.getBasicRemote
    Future {
       Thread.sleep(1000)
       endpoint.sendText("Result after some pondering: " + message) }
    "please wait ... "
  }
  ...
}
```

Die Websocket-API bietet die Möglichkeit Session-spezifische Daten in einer Map zu speichern. Beispielsweise können die Aufrufe eines bestimmten Clients wie folgt gezählt werden[12]:

```
import ...

@ServerEndpoint(value = "/echo")
class EchoEndpoint {

  @OnMessage
  def onMessage(message: String, session: Session): String = {
    println("received message " + message)
    val endpoint = session.getBasicRemote
    val userProperties = session.getUserProperties
    val nrOpt = userProperties.asScala.get("nr")
    var nr: Integer = 0
    nrOpt match {
      case Some(x) =>
        nr = x.asInstanceOf[Integer]
      case None =>
    }
    nr = nr + 1
    userProperties.asScala.put("nr", nr)
    Future { Thread.sleep(1000); endpoint.sendText("Echo Nr " + nr.toString) }
    "please wait ... "
  }

  ...

}
```

[10] http://www.eclipse.org/jetty/
[11] http://vertx.io
[12] Für Details siehe die API-Dokumentation http://docs.oracle.com/javaee/7/api/javax/websocket/Session.html

Aufgabe Modifizieren / erweitern Sie ihre Lösung von Aufgabe 1 von Blatt 7 derart, dass eine auf dem Server laufende Faktorisierung im (Html-) Client mit einem Knopf abgebrochen werden kann.

Verwenden sie Websockets zur Kommunikation und im Server nach Bedarf und Belieben Scala-Futures und / oder Promises.