# EJOE

*Einführung in **E**nhanced **J**ava **O**bject **E**xchange*

*Version 1.3*



Michael Manske

EJOE Maintainer

netseeker@manskes.de

Inhaltsverzeichnis

# 1 Vorwort

Diese Einführung wurde anhand einer Vorabversion von EJOE 0.4.0 erstellt. Zum Zeitpunkt der Erstellung lag EJOE 0.3.9.2 vor.

Alle Codebeispiele und Konfigurationen wurden gegen diese Version getestet. Sie sind nicht bzw. nur begrenzt kompatibel zu älteren Versionen von EJOE.

Neuere Versionen von EJOE können Abweichungen enthalten, die mit Teilen dieser Einführung nicht kompatibel sind.

EJOE ist ein Open Source Projekt, dass vom Maintainer sowie wechselnden Kontributoren vorangetrieben wird. Für derartige Projekte, die nicht in die Infrastruktur großer Organisationen der Open Source Szene eingebunden sind, ist es leider sehr oft schwierig Informationen und Services für Integratoren und Anwender in einer professionellen Form zur Verfügung zu stellen. Dieser Mangel ist oft fehlender Koordination, fehlendem Verständnis für die Belange der Anwender sowie dem für viele Open Source Entwickler und -Maintainer typischen chronischem Zeitmangel aufgrund des ständigen Spagats zwischen Arbeit, Familie, Freunden und Ihren Projekten geschuldet. Dies soll nicht als Rechtfertigung für mangelnde Dokumentationen dienen, trägt jedoch vielleicht etwas zum Verständnis bei und kann gern als Aufruf zu aktiver(er) Mitarbeit und Unterstützung aufgefasst werden.

Diese Einführung dient dem besseren Verständnis darüber, was EJOE eigentlich ist, welche Möglichkeiten es bietet und wie man diese Möglichkeiten zum Einsatz bringt. Sie erhebt jedoch keinen Anspruch auf Vollständigkeit oder gar auf ein niedriges Einstiegslevel in der Art einer Reihe gewisser Programmierhandbücher, deren Titel mit den Worten *„für Dummies“* enden.

*Anregungen und auch die unvermeidliche Kritik zu dieser Einführung (und natürlich auch zu anderen Dokumentationen des EJOE Projektes wie bspw. der Projektseite) sind willkommen und ausdrücklich erbeten.*

# 2 Abstrakt

*„Simplicity is prerequisite for reliability„*
*Edsger W.Dijkstra*

Der Austausch von Daten zwischen Anwendungen ist dieser Tage ein essentieller Bestandteil der Anwendungsentwicklung. Netzwerkfähigkeit ist inzwischen kein Schlagwort mehr, sondern vielmehr eine zwingende Notwendigkeit für viele Anwendungen.

Java macht seit jeher die grundlegende Vernetzung von Anwendungen sehr einfach. Die Anwendung der von Java bereitgestellten Bordmittel - insbesondere in laufzeitkritischen Szenarien sowie beim Versand und Empfang beliebiger Anwendungsobjekte - wird jedoch weitestgehend den Entwicklern überlassen. Zusammenhängende, vollständige Dokumentationen über die korrekte Anwendung der durch Java bereitgestellten Möglichkeiten sind in diesem Sektor oft Mangelware.

Darüber hinaus hat mit Java 1.4 unter anderem eine neue I/O-API, das sog. NIO-Package, in Java Einzug gehalten und bringt unter anderem neue Möglichkeiten und Herausforderungen für die Netzwerkkommunikation mit sich.

Die Implementierung zuverlässiger und skalierbarer Transportschichten ist trotz der von Java bereitgestellten Low-Level-Funktionalitäten leider immer noch mit hohem Aufwand für Lernprozess, Planung und Implementierung verbunden.

*Das Framework EJOE bietet eine einfach anwendbare, vereinheitlichende und skalierbare Architektur um bestehende und neue Anwendungen mit einer zuverlässigen Transportschicht auszustatten.*

# 3 Überblick

EJOE ist eine zusammenführende Implementierung der durch Java bereitgestellten Netzwerk-, Parallelisierungs- sowie Serialisierungsfunktionalitäten. Es vereint diese Funktionen in einem sogenannten Objektbroker (ORB)[1], welcher das allgemeine Request-Process-Response-Pattern[2] implementiert und jeweils eine fertige Client- bzw. Serverkomponente bereitstellt. EJOE fällt damit unter anderem im weitesten Sinne in jenen Bereich, der allgemein als Middleware bezeichnet wird.

EJOE kann zur Entwicklung von Client/Server-Anwendungen sowie zur Bereitstellung von Remote-Services in bereits bestehenden Anwendungen verwendet werden. Seit EJOE 0.3.9.1 bietet EJOE Features, welche im allgemeinen der Definition eines Remoting-Frameworks entsprechen.

EJOE bietet die folgenden Highlights:

- ✔ Open Source unter der Apache License Version 2
- ✔ betriebssicher und hochperformant
- ✔ einfach integrierbare Schnittstelle für Client und Server
- ✔ umfangreiche, leicht erweiterbare Unterstützung verschiedenster Serialisierungstechniken
- ✔ zuverlässiges, skalierbares, einstellbares Multithreadingverhalten
- ✔ Unterstützung sowohl für non-blocking IO (java.nio) als auch das streamorientierte blocking IO (java.io)
- ✔ Unterstützung für WSIF (Web Service Invocation Framework)
- ✔ Unterstützung des Crispy-Frameworks
- ✔ Remote Method Invocation via Remote Reflection
- ✔ Unterstützung für die Ausführung asynchroner Requests
- ✔ ohne J2EE einsetzbar
- ✔ InProcess-Modus (Kommunikation ohne Sockets innerhalb einer Instanz der Java Virtual Machine)

EJOE ist beispielsweise keine gute Wahl, wenn:

- ✗ ein anderes Transportprotokoll als das EJOE-Protokoll oder HTTP verwendet oder implementiert werden soll
- ✗ Die Funktionen eines ausgereiften Applicationservers oder Servletcontainers benötigt werden
- ✗ Datenstreaming beispielsweise für die unterbrechungsfreie Übertragung von Audio- oder Videodaten umgesetzt werden soll

1 Siehe ORBs - Object Request Broker
2 Siehe Single-transmission bilateral interaction patterns

## 3.1 Download

EJOE wird auf sourceforge.net gehostet. Die Projektdokumentation ist über http://ejoe.sourceforge.net/ erreichbar. Quellcode, kompilierte Releases, Tracker, Forum und Mailingliste sind auf der Projektseite http://sourceforge.net/projects/ejoe/ erreichbar. Unter http://sourceforge.net/project/showfiles.php?group_id=116355 Gelangt man direkt zu den Downloads. Jedes Release wird als GZIP-Archiv sowohl fertig kompiliert und mit den empfohlenen Mindestabhängigkeiten als auch als Quellcode angeboten.

Das Archiv der fertig kompilierten Version ist wie folgt aufgebaut:

```
ejoe_x.x.gz
|-- ejoe_x.x.tar
    |-- doc/
    |-- lib/
        |-- xpp3_min-1.1.3.x.x.jar
        |-- xstream-1.x.jar
    |-- ejoe_xx.jar
    |-- LICENSE.txt
    |-- XStream.LICENSE
    |-- README.txt
```

| Datei/Verzeichnis | Beschreibung | zur Laufzeit benötigt |
|---|---|---|
| doc | Enthält die komplette EJOE Homepage als HTML sowie die Javadoc | ✖ |
| lib<br>/xpp3_min-1.1.3.x.x.jar<br>/xstream-1.x.jar | Enthält die empfohlenen 3rd-Party Libraries. | (✓)<br>nur bei Einsatz des XStream-Adapters benötigt |
| ejoe_xx.jar | EJOE Framework | ✓ |
| LICENSE.txt | Kopie der Apache 2 Lizenz | ✖ |
| XStream.LICENSE | Kopie der von XStream verwendeten Lizenz | ✖ |
| README.txt | Hinweise zum EJOE Release | ✖ |

*Tabelle 3.1.*

# 4 Architektur

Im Hinblick auf einen möglichst hohen Grad an Einfachheit für die Integration des Frameworks bei gleichzeitiger Erweiterbarkeit, verfolgt EJOE einen Kompromiss bestehend aus einer soliden Architektur mit fest definierten Punkten für die Erweiterbarkeit.

EJOE besteht grundsätzlich aus:

- einem Server, welcher Multithreading und Netzwerkoperationen kapselt,
- eine Schnittstelle für die Verarbeitung bzw. Weiterleitung eingegangener Anfragen,
- Modulen für die Umwandlung von Objekten in über Netzwerk übertragbare Entitäten sowie
- einem eigenen Remote-Classloader
- einer Clientkomponente

Der Kern, de.netseeker.ejoe.EJServer, selbst ist nicht einfach erweiterbar, da er die Aufgabe hat, alle aufwändig zu implementierenden, schwierigen und komplexen Aufgaben in den Bereichen Parallelisierung und Netzwerkoperationen zu kapseln.

Er bietet jedoch eine vordefinierte Schnittstelle zur Veröffentlichung von externer Geschäftslogik. Diese Schnittstelle ist prinzipiell für die Verarbeitung bzw. Weiterleitung eingegangener Anfragen an die Geschäftslogik der umgebenden Anwendung zuständig. Sie stellt sowohl fertig einsetzbare Implementierungen, welche auf Reflection aufsetzen, als auch ein Basismodul, welches abgeleitet werden kann, um in den Verarbeitungsprozess von Anfragen einzugreifen, zur Verfügung.

Implementierungen dieser Schnittstelle werden *ServerHandler* genannt und im Abschnitt Was ist ein ServerHandler? näher erläutert.

EJOE bringt von Hause aus Module für die Umwandlung von Objekten in über Netzwerk übertragbare Entitäten mit. Diese Module sind für den Prozess der Serialisierung sowie Deserialisierung von Clientanfragen und Serverantworten verantwortlich. EJOE beinhaltet eine Vielzahl an vorgefertigten Modulen, welche auf die unterschiedlichsten Frameworks, Protokolle und Standards im Bereich Serialisierung zurückgreifen. EJOE kann in sehr einfacher Weise mit eigenen Modulen für die Serialisierung erweitert werden. Derartige Module werden *SerializeAdapter* genannt und unter SerializeAdapter – Warum Serialisierung so wichtig ist näher erläutert.

Die Clientkomponente, *de.netseeker.ejoe.EJClient*, ist nicht ohne weiteres erweiterbar, kann dafür jedoch sehr einfach integriert werden. Sie wird im Abschnitt Die Clientkomponente: EJClient näher erläutert.

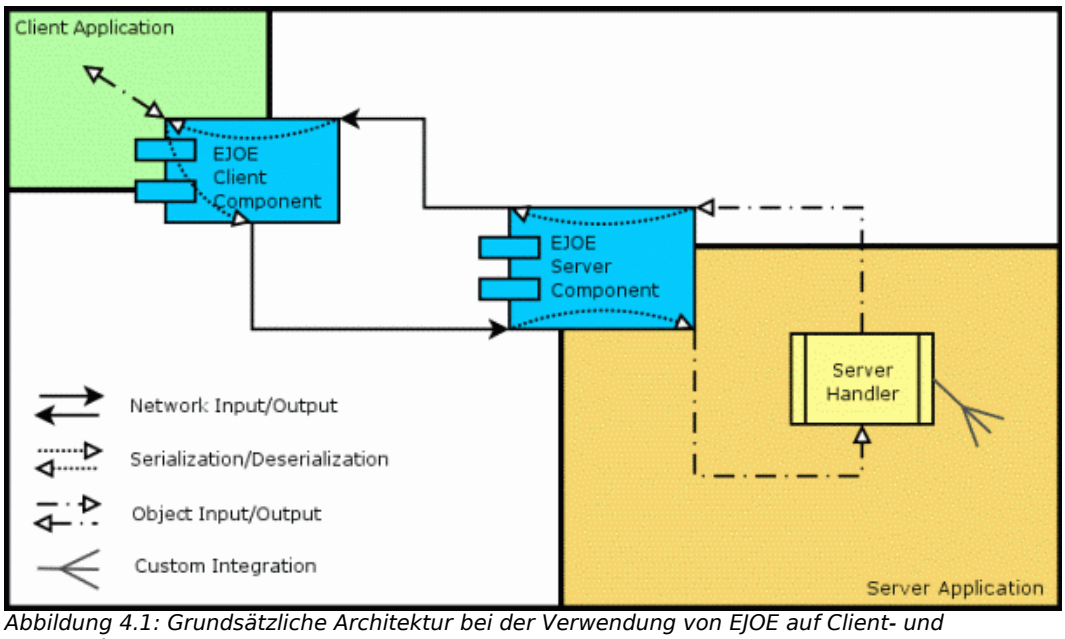



*Abbildung 4.1: Grundsätzliche Architektur bei der Verwendung von EJOE auf Client- und Serverseite*

## 4.1 Serverarchitektur

Die Serverkomponente, de.netseeker.ejoe.EJServer, verbindet mehrere Module für Parallelisierung, Netzwerkoperationen und Datenverarbeitung unter einem Dach:

- *ConnectionAcceptor*: Modul zum Annehmen und Aushandeln von Clientverbindungen
- *ConnectionProcessor*: Modul mit der Aufgabe angeforderte Operationen für eine Clientverbindung je nach Typ (Read/Write) für die Weiterverarbeitung durch einen entsprechenden Threadpool vorzubereiten und an diesen zu delegieren. Die Anzahl der verfügbaren Connection Processoren ist einstellbar, die Standardanzahl entspricht der auf dem jeweiligen System vorhandenen Prozessoren (CPUs) für eine ideale Parallelisierung über alle vorhandenen Prozessoren.
- *Threadpool für Read/Process Operationen*: Threadpool mit einer konfigurierbaren Anzahl von Workerthreads. Diese spezifischen Workerthreads haben folgende Aufgaben:
  - Daten über das Netzwerk zu empfangen
  - diese Daten mithilfe des vom Client angeforderten *SerializeAdapters* zu deserialisieren
  - den deserialisierten Request an einen *ServerHandler* zu delegieren
  - nach erfolgter Bearbeitung durch den *ServerHandler*, das Resultat an einen ConnectionProcessor zur Weiterverarbeitung zu delegieren

- *Threadpool für Write-Operationen*: Threadpool mit einer konfigurierbaren Anzahl von Workerthreads. Diese spezifischen Prozesse haben folgende Aufgaben:
  - Serialisierung der Serverantwort mithilfe des vom Client angeforderten Serialisierungsmechanismus
  - Versenden der Serverantwort über die bestehende Clientverbindung
- *SerializeAdapter*: Schnittstelle[3] und konkrete Implementierungen für die verschiedensten Mechanismen zur Serialisierung und Deserialisierung von Objekten. EJOE liefert bereits eine Vielzahl an einsetzbaren *SerializeAdaptern* mit, die das Spektrum von normaler Objektserialisierung über XML-basierte Mechanismen/Formate bis hin zu spezifischen Mechanismen/Protokollen wie JSON oder Hessian abbilden.
- *ServerHandler*: Schnittstelle und konkrete Implementierungen für die Verarbeitung von gelesenen, deserialisierten Clientanfragen durch externe Anwendungslogik.

de.netseeker.ejoe.EJServer bildet somit im eigentlichen Sinne vielmehr einen Container für Konfiguration, Start und Stop der beinhalteten Module ab, als selbst ein tatsächlicher Serverprozess im herkömmlichen Sinne zu sein.

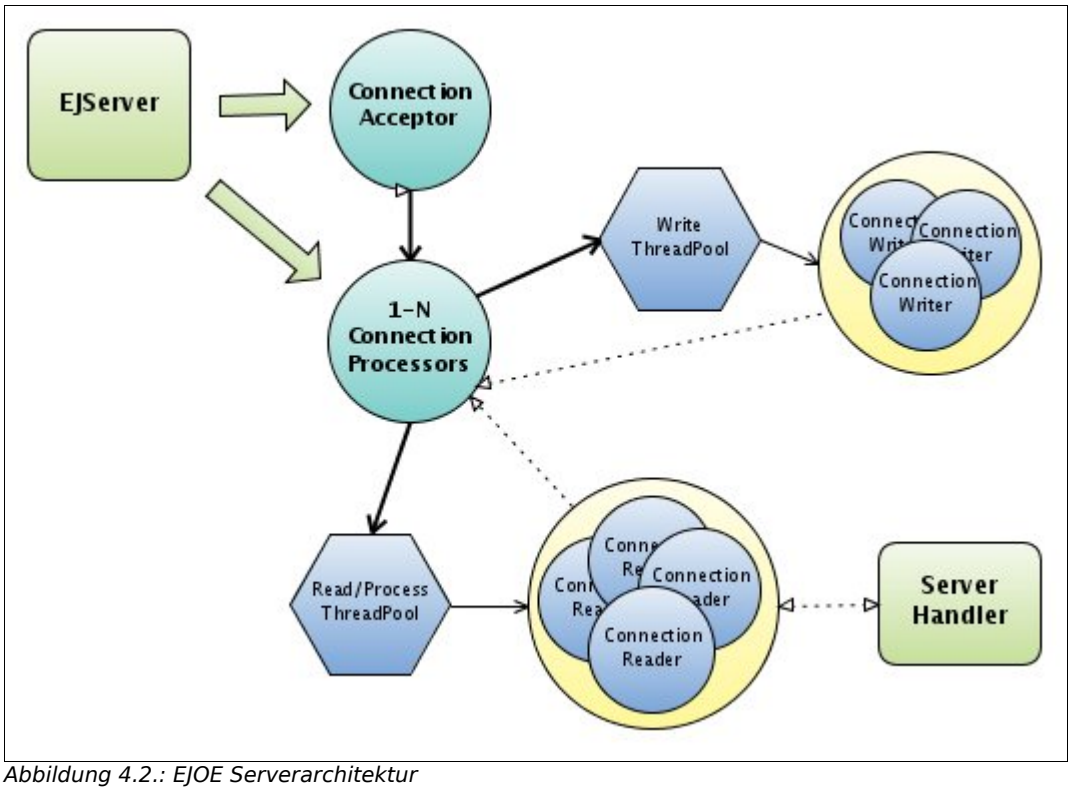



*Abbildung 4.2.: EJOE Serverarchitektur*

3 Schnittstelle bezeichnet hier eine definierte API für Erweiterungen

# 5 Hinweise zur Integration

> *„The most important single aspect of software development is to be clear about what you are trying to build.„*
>
> *Bjarne Stroustrup*

Für eine erfolgreiche Integration von EJOE in die eigene Anwendung sind in der Regel mindestens folgende Schritte notwendig:

- **Ermittlung**
  - der zu erwartenden Netzwerkinfrastruktur
  - der zu erwartenden durchschnittlichen und maximalen Anfragelast
  - der verfügbaren Ressourcen (Arbeitsspeicher, vorhandene Prozess-auslastung, freie Dateihandles etc.) auf dem Zielsystem (Server)
- **Definition** bzw. **Auswahl** (bei vorhandener Anwendungslogik), der via EJOE übers Netzwerk verfügbar zu machenden Komponenten und Funktionen
- **Entscheidung**, ob ein eigener ServerHandler notwendig wird oder ob ein ServerHandler aus der Familie der RemotingHandler (Remote Method Invocation durch serverseitiges Reflection) benutzt werden soll
- **Analyse** der zu erwartenden, über Netzwerk zu übertragenden Datentypen
- Bei der zu erwartenden Verwendung von komplexen Objekten, **Analyse** der in den ausgewählten Komponenten/Funktionen evtl. eingesetzten Binding-Frameworks wie beispielsweise JAXB oder Castor-XML
- **Auswahl** eines geeigneten SerializeHandler unter Berücksichtigung der erwarteten Datentypen sowie evtl. bereits eingesetzten Binding-Frameworks
- evtl. **Entscheidung** über Implementierung eines eigenen SerializeHandlers um Objekte aus Binding-Framework XY ideal übertragen zu können
  - EJOE kann in der empfohlenen Standardkonfiguration automatisch fast alle Objektarten übertragen!
- **Integration** EJServer in die Serveranwendung
- **Integration** EJClient in die Clientanwendung(en)
- Evtl. **Beschaffung** geeigneter Hardware bei geringen, vorhandenen Systemressourcen und erwarteter, hoher Anzahl von Clientanfragen. Der EJOE-Server ist von Natur aus prozessorintensiv, da ihm zum einen eine Multiprozess-Architektur zugrunde liegt und zum anderen Serialisierungsvorgänge in der Regel sehr prozessorintensive Vorgänge sind.

## 5.1 Get a taste: EJServer mit passendem EJClient

Ablauf:

- Erzeugen einer EJServer-Instanz
- Konfigurieren des Servers
- Starten des Servers
- Erzeugen einer EJClient-Instanz
- Durchführen einer Client-Anfrage am EJServer

```
import de.netseeker.ejoe.EJServer;
...
{
      ...
      EJServer server = new EJServer( new MyServerHandler() );
      server.start();
      ...
}
```

*nach zwei Codezeilen läuft der EJOE-Server*

```
import de.netseeker.ejoe.EJServer;
...
{
      //benutze RemoteReflection, binde an das Netzwerkinterface für die
      //Adresse 192.168.1.21 und warte auf Port 80 auf Verbindungen
      EJServer server = new EJServer( new MyServerHandler(), „192.168.1.21“, 80 );
      //Non-blocking IO oder streamorientiertes, blockendes I/O?
      server.enableNonBlockingIO( true|false );
      //GZIP Kompression für Netzwerkkommunikation
      server.enableCompression( true|false );
      //Erlaube persistente Clientverbindungen?
      server.enablePersistentConnections( true|false );
      //Machen Firewallkonfigurationen es evtl. notwendig, dass einige oder
      //alle Clients Requestdaten im HTTP-Protokoll übertragen müssen?
      server.enableHttpPackaging( true|false );
      //Wie viele Read/Process- und Write-Workerthreads benötigen wir?
      //Wie hoch ist die erwartete Anfragelast?
      //Wie sieht es mit den verfügbaren Hardwareressourcen aus?
      server.setMaxReadProcessors( Anzahl von Read/Process-Workerthreads );
      server.setMaxWriteProcessors( Anzahl von Write-Workerthreads );
      //starte den Server
      server.start();
      ...
}
```

*programmatisch die Konfiguration des EJOE-Server anpassen*

```
import de.netseeker.ejoe.EJClient;
...
{
      MyDataRequestBean bean = ...;
      EJClient ejClient = new EJClient( „192.168.1.21“, 80 );
      //Single Request oder wird der Client wiederverwendet und soll
      //nicht jedesmal eine neue Verbindung öffnen/schließen?
      //persistente Verbindungen können später via EJClient#close() geschlossen werden
      ejClient.enablePersistentConnection(true|false);
      //senden wir Textdaten, bspw. XML, dann könnte sich Kompression
      //evtl. vorteilhaft auswirken
      ejClient.enableCompression(true|false);
      //Unterliegen wir bestimmen Firewallregeln, die lediglich HTTP-Verkehr gestatten?
      ejClient.enableHttpPackaging(true|false);

      MyResultType result = null;
      try
      {
             result = (MyResultType)ejClient.execute( bean );
      }
      catch(RemoteException re)
      {
             //EJServer did return an Error
      }
      catch(IOException e)
      {
             //connection or serialization issues maybe?
      }
      ...
}
```

*Erzeugen eines EJClient und Durchführen einer Client-Anfrage*

# 6 Die Serverkomponente: EJServer

Der EJServer ist ein vollständiger TCP/IP basierter Netzwerkserver, der entweder zum Aufbau von Serveranwendungen oder zur Integration in bestehende Anwendungen genutzt werden kann. Er bietet Unterstützung für:

- ✔ Java non-blocking IO (java.nio)
- ✔ Java stream-oriented IO (java.io)
- ✔ persistente als auch nicht-persistente Verbindungen
- ✔ Kompression von Netzwerkdaten mittels GZip
- ✔ skalierbares Multithreading
- ✔ Nutzung aller Prozessorkerne auf Multiprozessorsystemen
- ✔ HTTP Protokoll (1.0 partiell, 1.1 partiell und ausschließlich lesend)
- ✔ partielles Lesen/Schreiben ohne Prozessblockierung bei großen Datenmengen

- ✔ automatische Überwachung der IO-Prozesse mit Reaktivierung/Neustart „gestorbener“ Prozesse
- ✔ dateibasierte als auch programmatischer Konfiguration

## 6.1 Erzeugen der Serverinstanz

Ein Server ist immer eine Instanz der Klasse de.netseeker.ejoe.EJServer. Die Instanz muss durch Aufruf eines der angebotenen Konstruktoren erzeugt werden:

::de.netseeker.ejoe

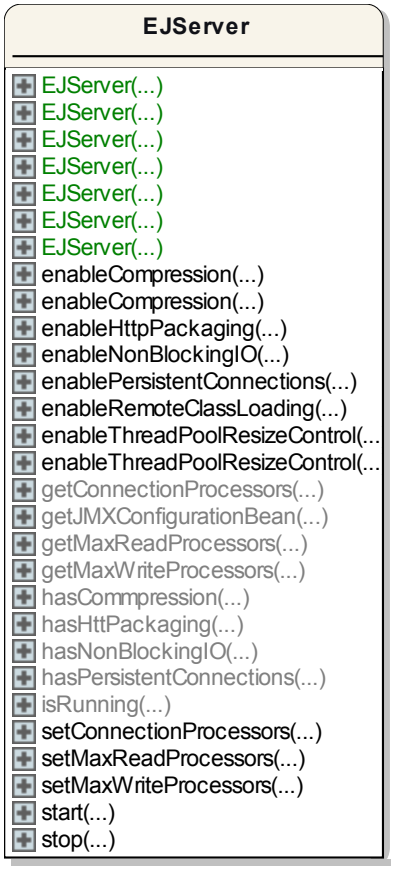



*Abbildung 6.1: de.netseeker.ejoe.EJServer*

- `EJServer()`: Erzeugt eine neue Instanz von EJServer, vorkonfiguriert mit den Einstellungen aus der Datei „ejserver.properties“, welche auf dem Classpath verfügbar sein muss
- `EJServer(Properties properties)`: Erzeugt eine neue Instanz von EJServer, vorkonfiguriert mit den übergebenen Einstellungen. Dieser Konstruktor kann benutzt werden, wenn die Einstellungen für EJServer bereit aus einer anderen Konfigurationsdatei geladen wurden.
- `EJServer(String pathToConfigFile)`: Erzeugt eine neue Instanz von EJServer mit den Einstellungen aus der übergebenen Konfigurationsdatei.
- `EJServer(ServerHandler handler)`: Erzeugt eine neue Instanz von EJServer mit Standardeinstellungen und dem übergebenen ServerHandler.
- `EJServer(ServerHandler handler, int port)`: Erzeugt eine neue Instanz von EJServer mit Standardeinstellungen und dem übergebenen ServerHandler. Es wird anstelle des Standardports (12577) am übergebenen Port auf eingehende Verbindungen gewartet.
- `EJServer(ServerHandler handler, String bindAddr)`: Erzeugt eine neue Instanz von EJServer mit Standardeinstellungen und dem übergebenen ServerHandler. Es wird das Netzwerkinterface für die übergebene Bind-Adresse benutzt, zum Beispiel „127.0.0.1“ oder „81.209.148.146“. Damit kann man verhindern, dass EJServer auf allen verfügbaren Netzwerkinterfaces auf eingehende Verbindungen wartet.
- `EJServer(ServerHandler handler, String bindAddr, int port)`: Erzeugt eine neue Instanz von EJServer mit Standardeinstellungen und dem übergebenen ServerHandler. Es wird das Netzwerkinterface für die übergebene Bind-Adresse benutzt, zum Beispiel „127.0.0.1“ oder „81.209.148.146“. Damit kann man verhindern, dass EJServer auf allen verfügbaren Netzwerkinterfaces auf eingehende Verbinden wartet. Es wird anstelle des Standardports (12577) am übergebenen Port auf eingehende Verbindungen gewartet.

## 6.2 Anpassen der Prozesslimits

EJServer verwendet zwei unterschiedliche Threadpools für die serverseitigen Aufgaben:

- Lesen und Bearbeiten von Clientanfragen
- Senden von Serverantworten

Wenn nicht abweichend angegeben startet EJServer mit

- vier Prozessen zum Lesen und Bearbeiten von Clientanfragen
- zwei Prozessen zum Senden von Serverantworten

Insgesamt lediglich sechs Prozesse für einen Netzwerkserver mag sich im ersten Moment wenig anhören, ist jedoch oftmals eine mehr als ausreichende Anzahl von Prozessen. Welche Werte jedoch den optimalen Durchsatz und die ideale Hardwareauslastung bewirken, ist von Hardware zu Hardware und Plattform zu Plattform unterschiedlich.

Grundsätzlich gilt, nach den Erkenntnissen diverser Testszenarien, folgende Faustregel:

*Ein Verhältnis von 2/1 zwischen Prozessen zum Lesen und Bearbeiten von Clientanfragen und Prozessen zum Senden von Serverantworten bietet durchschnittlich das ausgeglichenste Serververhalten.*

Der jeweilige Threadpool kann über die Methode `setMaxReadProcessors` bzw. `setMaxWriteProcessors` in EJServer beeinflusst werden:

```
import de.netseeker.ejoe.EJServer;
...
{
      EJServer server = new EJServer( ... );
      //Wie hoch ist die erwartete Anfragelast?
      //Wie sieht es mit den verfügbaren Hardwareressourcen aus?
      //Wie viele Read/Process-Workerthreads benötigen wir?
      server.setMaxReadProcessors( Anzahl von Read/Process-Workerthreads );
      //Wie viele Write-Workerthreads benötigen wir?
      server.setMaxWriteProcessors( Anzahl von Write-Workerthreads );
      //starte den Server
      server.start();
      ...
}
```

*programmatisch die Konfiguration des Threadpools anpassen*

## 6.3 Non-Blocking IO oder streamorientiertes I/O (java.nio vs. java.io)

Seit Java 1.4 stehen Entwicklern neue Möglichkeiten zur Umsetzung von Netzwerkfunktionen zur Verfügung. So kann weitaus direkter als früher mit den Netzwerkfähigkeiten des Betriebssystem gearbeitet werden, um insbesondere Serversysteme wesentlich skalierbarer zu gestalten. Mehr Einfluss heißt aber auch mehr Arbeit für Entwickler: Die neuen Funktionen wollen korrekt eingesetzt und bedient werden, was sich je nach Zielstellung hinsichtlich Latenz und Skalierbarkeit mehr und mehr komplex gestaltet.

EJOE implementiert sowohl die bisherigen, streamorientierten als auch die neu hinzugekommenen Netzwerkfunktionen in zuverlässiger und skalierbarer Art und Weise.

Standardeinstellung in EJOE ist die Benutzung der neuen I/O Funktionen. Mit Ihnen kann bei richtiger Anwendung höhere Latenz, bessere Auslastung unter Multiprozessorsystemen sowie die Bearbeitung hoher Lastverhältnisse mit einer vergleichsweise geringen Anzahl gleichzeitiger Prozesse umgesetzt werden. Merkmale, die je nach Szenario, nur schwierig mit den im Vergleich wenig beeinflussbaren, streamorientierten Netzwerkfunktionen des java.io-Packages zu erreichen waren.

Eine der besonderen Schwierigkeiten des neuen non-blocking I/O liegt allerdings im Speicherverbrauch. Während es mit streamorientierten I/O relativ problemlos möglich ist, von großen Datenmengen jeweils nur Teile im Speicher zu halten, gestaltet sich dies mit non-blocking I/O schwieriger, da wie der Name schon sagt, die Übertragung nicht aufgrund des Nachladens weiterer Datenteile einfach blockiert werden darf.

Mit Unterstützung partieller Lese-/Schreibvorgänge kommt EJOE hier auch unter Verwendung des neuen API dem Speicherverbrauch des streamorientierten I/O sehr nahe.

Sollte trotzdem aus bestimmten Gründen der Einsatz von streamorientierten I/O nahe liegen, kann dies problemlos im Server aktiviert werden:

```
EJServer server = new EJServer(...);
server.enableNonBlockingIO(false);
```

*Deaktivieren des non-blocking I/O*

➔ Die Clientkomponente reagiert dynamisch auf die jeweilige Servereinstellung und bedarf keiner Angaben durch den Entwickler über den verwendeten I/O-Typ

## 6.4 HTTP Unterstützung

EJServer bietet seit Version 0.3.9.1 die Möglichkeit Anfragen, welche im HTTP-Protokoll 1.0[4] und 1.1[5], übermittelt werden, zu verarbeiten.

Beim Betrachten der HTTP Unterstützung fällt folgendes auf: EJOE ist kein HTTP-Server und auch kein HTTP-Client. Es kann in der Kommunikation zwischen EJClient und EJServer lediglich das HTTP-Protokoll verwendet werden. Für die Kommunikation alternativer HTTP-Clients mit EJServer siehe Kommunikation mit einem alternativen HTTP-Client).

➔ EJServer implementiert die beiden Protokollversionen lediglich partiell.

HTTP-Unterstützung ist standardmäßig nicht aktiviert und muss bei Bedarf explizit aktiviert werden:

```
EJServer server = new EJServer( ... );
server.enableHttpPackaging( true );
```

*Aktivieren der HTTP-Unterstützung*

Die HTTP-Unterstützung ist dabei als duales Protokoll zu verstehen. EJServer bietet die Unterstützung zusätzlich zum EJOE-eigenen Protokoll an und ist auch bei aktivierter HTTP-Unterstützung weiterhin in der Lage, Anfragen im EJOE-Protokoll zu verarbeiten.

Wurde sowohl in EJClient als auch EJServer HTTP aktiviert, benutzen diese untereinander immer HTTP/1.0. EJServer bietet jedoch auch die Möglichkeit HTTP-Anfragen von nicht-EJOE-Clients zu verarbeiten. In einem solchen Szenario empfiehlt es sich für eine erweiterte HTTP/1.1 Konformität persistente Verbindungen zu aktivieren:

```
server.enablePersistentConnections( true );
```

*Aktivierung persistenter Verbindungen*

4 http://tools.ietf.org/html/rfc1945
5 http://tools.ietf.org/html/rfc2616

## 6.5 Konfiguration des Servers mittels Konfigurationsdatei

EJServer kann anstelle des programmatischen Ansatzes zur Konfiguration auch mittels einer Konfigurationsdatei eingerichtet werden.

Beim Aufruf des Standardkonstruktors EJServer() sucht EJServer auf dem Classpath nach der Datei *ejserver.properties*. Wird eine solche Datei gefunden, konfiguriert sich der Server selbst entsprechend der in der Konfigurationsdatei angegebenen Einstellungen.

Alternativ zum Standardkonstruktor können auch die Konstruktoren EJServer(java.lang.String pathToConfigFile) bzw. EJServer(java.util.Properties properties) verwendet werden um eine bereits vorhandene, mit abweichendem Dateinamen versehene Konfigurationsdatei zu verwendenden.

## 6.6 Referenz ejserver.properties

| Einstellung | Beschreibung | Mögliche Werte | Standard-wert | Pflicht |
|---|---|---|---|---|
| ejoe.port | TCP/IP Port auf dem EJServer auf Client-verbindungen wartet | 1-65535 | 12577 | × |
| ejoe.interface | DNS-Bezeichner oder IP-Adresse des zu benutzenden Netz-werkinterfaces | gültige IPv4 oder IPv6 Adresse | 127.0.0.1 | × |
| ejoe.serverHandler | Vollständiger Klassen-name inkl. Package-path für den Server-Handler, mit dem Clientanfragen bearbeitet werden | package. ClassName einer Instanz von de.netseeker. ejoe.handler. Server Handler | - | ✓ |
| ejoe.enableNIO | Gibt an, ob non-blocking I/O oder streamorientiertes I/O benutzt werden soll | true\|false | true | × |
| ejoe.maxReadProcessors | Anzahl von Worker-prozessen für das Lesen und Bearbeiten von Clientanfragen | >=1 | 4 | × |
| ejoe.maxWriteProcessors | Anzahl von Worker-prozessen für das Senden von Server-antworten | >=1 | 2 | × |
| ejoe.threadPoolResizeControl | Gibt an, ob die auto-matische Über- | true\|false | false | × |

| Einstellung | Beschreibung | Mögliche Werte | Standard-wert | Pflicht |
|---|---|---|---|---|
| | wachung für abgestürzte Workerprozesse verwendet werden soll | | | |
| ejoe.compression | Gibt an, ob der Server komprimierte Clientanfragen unterstützt | true\|false | false | × |
| ejoe.persistentConnections | Gibt an, ob persistente Clientverbindungen unterstützt werden sollen | true\|false | true | × |
| ejoe.httpTunneling | Gibt an, ob Anfragen im HTTP Protokoll unterstützt werden | true\|false | false | × |
| ejoe.remoteClassloader | Gibt an, ob Classloaderanfragen von Clients beantwortet werden | true\|false | false | × |

## 6.7 Was ist ein ServerHandler?

*ServerHandler wurden im Abschnitt 4.1 Serverarchitektur bereits kurz angesprochen. Daher wissen wir, dass es sich um eine Schnittstelle als auch konkrete Implementierungen für die Verarbeitung von gelesenen, deserialisierten Clientanfragen durch externe Anwendungslogik handelt.*

Ein ServerHandler hat die Aufgabe Client-Anfragen zur Weiterbehandlung und ggf. Beantwortung an externe Anwendungslogik zu delegieren. Er ist somit de facto **die Schnittstelle zwischen EJOE und der/den Anwendungen**, welche über EJOE RemoteServices zur Verfügung stellen.

Es existieren drei Arten von ServerHandlern in EJOE:

1) RemotingHandler: bieten die Möglichkeit Client-Anfragen per Reflection zu verarbeiten
   - de.netseeker.ejoe.handler.DefaultRemotingHandler: Standard Remoting-Handler, entspricht der Verwendung von normaler Reflection mit einigen Geschwindigkeitsgewinnen durch Caching von Klassen und Methoden.
   - de.netseeker.ejoe.handler.AssistedRemotingHandler: Erweiterter Remoting-Handler, welcher auf Basis von generierten und gecachten Proxyklassen für Reflection-Aufrufe arbeitet. Durch die Generierung der Proxyklassen dauert der erste Aufruf einer Methode länger als mit dem DefaultRemotingHandler,

alle folgenden Aufrufe sind jedoch schneller. Für die Proxygenerierung wird javassist eingesetzt.

➔ Der AssistedRemotingHandler ist nicht mit Java < 1.5.0 einsetzbar!

2) Eigene ServerHandler: ServerHandler, erstellt von – aus Sicht von EJOE – Fremdanwendungen, welche das vordefinierte Interface de.netseeker.ejoe.handler.ServerHandler implementieren
   - de.netseeker.ejoe.handler.MultiObjectHandler: Abstrakter Server-Handler, welcher für einfache Multi-Objekt-Anfragen erweitert werden kann. Dieser Handler gibt anstelle des in EJOE für ServerHandler üblichen, als java.lang.Object behandelten, Parameters eine java.util.Map vor.
   - de.netseeker.ejoe.handler.ClassHandler: Spezieller, EJOE-interner Server-Handler, welcher vom EJOE-eigenen Remote-Classloader angesprochen wird.

Bei der Integration von EJOE in die eigene Umgebung stellt sich also eine entscheidende Frage:

*Einen ServerHandler aus der Familie der RemotingHandler benutzen oder eigenen ServerHandler implementieren?*

## 6.8 Der eigene ServerHandler

*Wenn Reflection zu langsam, unsicher oder schlichtweg unpassend ist, weil beispielsweise zwischen Empfang einer Client-Anfrage durch EJOE und der Verarbeitung durch die externe Geschäftslogik noch Zwischenschritte (Prüfungen, zusätzliche Logik etc.) durchzuführen sind, dann bietet sich die Implementierung eines eigenen ServerHandlers an.*

EJOE stellt an eigene ServerHandler lediglich zwei Anforderungen:

1. Implementieren des Interfaces de.netseeker.ejoe.handler.ServerHandler
2. ServerHandler müssen threadsafe implementiert werden

Für den zweiten Punkt müssen zur erfolgreichen Umsetzung folgende Hinweise beachtet werden:

EJOE behandelt ServerHandler de facto als eine Art Singletons. Ein ServerHandler kann in EJOE in genau einer Instanz existieren. Bei der Implementierung eines Server-Handlers sollte auf generelle Synchronisation zum Erreichen der Threadsicherheit verzichtet werden, da dies EJOEs Multithreading nachhaltig beeinträchtigen würde.

```
package de.netseeker.ejoe.handler;

/**
 * Simple interface defining the entry point for all server handlers.
 * Server handlers are the working horses which the caller must implement.
 * Server handlers take the transported input objects send by the client
 * application and return none, one or more return values.
 *
 * @author netseeker aka Michael Manske
 * @since 0.3.0
 */
public interface ServerHandler
{
     /**
      * Handles a client request
      *
      * @param obj The input object transported by EJOE
      * @return null or a valid return value. If you want return custom datatypes,
      * eg. Your own beans and don‘t want (or not be able) to deploy the classes of
      * these datatypes on the client, you can turn on EJOEs remote classloading feature.
      */
      public Object handle( Object obj ) throws Exception;
}
```

*Das Interface de.netseeker.handler.ServerHandler*

### 6.8.1 Beispiel 1 – Der Echo-Server

*Ein Echo-Dienst, ist ein Dienst, der einfach alle eingehenden Daten wieder an den Client zurücksendet.*

Der Einfachheit halber beschränken wir den in diesem Beispiel anvisierten Echo-Server auf character-basierende Nachrichten:

```
import de.netseeker.ejoe.handler.ServerHandler;

public class EchoHandler implements ServerHandler
{
	private static final Logger logger = Logger.getLogger( EchoHandler.class.getName() );

	public Object handle( Object obj ) throws Exception
	{
		//the cast is just build in to ensure that we‘ve got a String
		//and not another object type
		String msg = (String)obj;
		logger.log( Level.INFO, „Received client message: „ + msg);
		return msg;
	}
}
```

*Der EchoHandler*

Start einer EJOE-Serverinstanz mit dem erstellten *EchoHandler* auf dem Port 9999:

```
import de.netseeker.ejoe.EJServer;

public class EchoServer
{
	private static final Logger logger = Logger.getLogger( EchoServer.class.getName() );

	public static void main( String[] args )
	{
		EJServer server = new EJServer( new EchoHandler(), 9999 );
		try
		{
			server.start();
		}
		catch (IOException e)
		{
			logger.log(Level.SEVERE, „Exception while starting server!“, e);
		}
	}
}
```

*Starten des Echo-Servers*

## 6.8.2 Beispiel 2 – Der PDF-Konvertierungsserver

**Szenario:**

Es wurde ein kommandozeilenbasiertes Tool zur Extraktion spezifischer Textdaten aus PDF-basierten Auftragsformularen entwickelt. Die extrahierten Textinformationen werden innerhalb der Vorstufe der Auftragsabwicklung für die Weitergabe an das Auftragsverwaltungssystem sowie die Auftragsarchivierung benötigt.

Bisher mussten das Auftragserfassungsprogramm sowie dieses Tool lediglich auf einer sehr geringen Anzahl von Arbeitsplätzen installiert werden. Der Aufwand für Installation und Update des Tools sowie der benötigten Java Runtime war problemlos verwaltbar. Aufgrund stark gestiegenen Auftragsaufkommens wird diese Funktionalität nun für eine große Anzahl von Arbeitsplätzen sowie den Cluster von Anwendungsservern, auf welchem das firmeneigene Portal, Kundenservices sowie die internetbasierten Aussendienstanwendungen gehostet werden, benötigt.

**Ziel:**

Erweiterung des kommandozeilenorientierten Programmes um Servereigenschaften damit alle betroffenen Arbeitsplätze sowie Anwendungsserver die Funktionalität des Programmes als Dienst ansprechen können und gleichzeitig die extrahierten Textdaten in eine zentrale Datenbank für die automatisierte Weiterverarbeitung abgelegt werden.

```
...
public class PdfExtractor
{
      public static void main( String[] args )
      {
             if( !pruefeParameter() )
             {
                    showHelp();
                    System.exit(0);
             }

             PdfExtractor extractor = new  PdfExtractor();
             String result = extractor.fuehreTextExtractionAus( args[0] );
             System.out.println( result );
      }

      public String  fuehreTextExtractionAus( String pathToPdfFile )
      {
                    ...
      }
      ...
}
```

*Aufbau des bisherigen Programms*

Während bisher die PDF-Dateien direkt zur Verfügung gestellt wurden, werden sie zukünftig als Datenströme über Netzwerk geliefert. Da PDF-Dateien als Binärdaten anzusehen sind, werden dafür Bytearrays verwendet werden. Darüberhinaus muss eine neue Schicht für die Datenbankinteraktion eingeführt werden um die extrahierten Texte in einer Datenbank abzuspeichern.

Diese Schicht wird hier nur abstrakt, beispielhaft behandelt, da Möglichkeiten zum Datenbankzugriff nicht Inhalt dieser Einführung sind.

Auslagerung der Textextraktion sowie Datenbankinteraktion in einen ServerHandler:

```
...
import de.netseeker.ejoe.handler.ServerHandler;

public class PdfConvertHandler implements ServerHandler
{
	private  DAOContext context;

	//DAContext ist rein fiktiv und stellt eine Schnittstelle zu einem fiktiven
	//Persistenzlayer dar, mit welchem Daten in eine Datenbank gespeichert bzw.
	//gelesen werden können
	public PdfConvertHandler( DAContext context )
	{
		this.context = context;
	}

	public Object handle( Object obj ) throws Exception
	{
		//wir gehen von binaeren PDF-Inhalten aus
		byte[] pdfData = (byte[])obj;
		PdfExtractor extractor = new PdfExtractor();
		//Der alte PDF-Parser wurde für die  Verarbeitung von Datenströmen vorbereitet
		String result = extractor.FuehreTextExtractionAus(  new ByteArrayInputStream(
		                                                        pdfData ) );
		//der extrahierte Text muss in eine zentrale DB gespeichert werden
		//die Implementierung des DB-Zugriffs wird in diesem Beispiel außen vor gelassen
		Transaction t = this.context.startTransaction();
		try
		{
			this.context.storeOrder( result, t );
			this.context.commitTransation( t );
		}
		catch( Exception e )
		{
			this.context.rollbackTransaction( t );
			//der Client sollte über den Fehler informiert werden
			//EJOE wird eine entsprechende RemoteException an den Client übergeben
			throw( e );
		}

		return result;
	}
}
```

*möglicher Aufbau des ServerHandlers für die PDF-Konvertierung und -speicherung*

Start einer EJOE-Serverinstanz mit dem erstellten *PdfConvertHandler*:

```
...
import de.netseeker.ejoe.EJServer;

public class PdfConvertServer
{
      private static final Logger logger = Logger.getLogger( PdfConvertServer.class.getName() );

      public static void main( String[] args )
      {
             DAContext daContext = ...;
             ...
             EJServer server = new EJServer( new PdfConvertHandler( daContext ), 9103 );
             //per default startet EJOE mit 4 Reader- und 2 Writerprozessen
             server.setMaxReadProcessors( 10 );
             server.setMaxWriteProcessors( 5 );
             try
             {
                    server.start();
             }
             catch (IOException e)
             {
                    logger.log(Level.SEVERE, „Exception while starting server!“, e);
             }
      }
}
```

*PDF-Konvertierungsserver starten*

## 6.9 Die RemotingHandler

Fertig einsetzbare RemotingHandler sind bereits in EJOE enthalten. Ihr Einsatz macht die Implementierung eines eigenen ServerHandlers überflüssig, entbindet jedoch nicht in allen Fällen von weiteren serverseitigen Anpassungen, da insbesondere in komplexen Serverarchitekturen notwendige Funktionalität oftmals nicht einfach ohne weiteres über Reflection bedient werden kann.

Die Benutzung der RemotingHandler erfordert zusätzliche Informationen über auf dem Server verfügbare Klassen und Methoden in der Clientimplementierung.

RemotingHandler benötigen eine erweiterte Parameterliste, da Sie im Gegensatz zu spezifischen ServerHandlern selbst kein Wissen über externe Anwendungslogik(en) besitzen. RemotingHandler benötigten im Detail Informationen über:

- die zu benutzende Entität
- die innerhalb dieser Entität anzusprechende Operation
- die für die anzusprechende Operation benötigen Argumente

Auf Java bzw. Reflection bezogen bedeutet dies, dass Packagepfad und Name der Zielklasse (=Entität), Methoden- oder Konstruktorname (=Operation) sowie ggf. die Argumente für die aufzurufende Methode bzw. den aufzurufenden Konstruktor benötigt werden.

Für Client-Anfragen stellt EJOE deshalb einen besonderen Anfragecontainer, den RemotingRequest, bereit. Der RemotingRequest dient sowohl als Container für oben genannte Daten als auch als Erkennungszeichen auf dem Server, dass es sich bei einer Anfrage um eine Anfrage des Typs „RemoteReflection“ handelt, welche ausschließlich von RemotingHandlern bearbeitet werden.

```
import de.netseeker.ejoe.EJClient;
...
{
      EJClient ejClient = new EJClient( „192.168.1.21“, 80 );

      List result = null;
      try
      {
            RemotingRequest request = new RemotingRequest( „com.who.AddressManagement“,
                                    „listAddressesByName“, new Object[]{ „Mustermann“ } );
            result = (List)ejClient.execute(request );
      }
      catch(RemoteException re)
      {
            //EJServer did return an Error
      }
      catch(IOException e)
      {
            //connection or serialization issues maybe?
      }
      ...
}
```

*beispielhafter RemotingRequest*

➔ Remoting-Anfragen müssen immer in einen RemotingRequest gekapselt werden!

### 6.9.1 Ohne die richtige Konfiguration geht nichts - Sicherheitsaspekte

RemotingHandler sind theoretisch von Natur aus eine gefährliche Sache: Clients könnten via RemotingRequest alle möglichen Funktionen des Servers anzapfen. EJOE steuert diesem Aspekt entgegen, indem es von vornherein erst einmal alle Reflectionaufrufe ablehnt, es sei denn, die Zielklasse ist in einer speziellen Konfigurationsdatei eingetragen.

EJOE sucht auf dem Classpath nach einer Datei mit dem Namen „ejoe-reflection-conf.properties“. Wird diese gefunden, werden Reflectionaufrufe für alle eingetragenen Klassen bzw. Packages gestattet. Wird allerdings keine solche Datei gefunden, wird die sehr restriktive Standardkonfiguration aus dem Pfad „META-INF/ ejoe-reflection-conf.properties“ verwendet.

```
##########################################################################################################
# This is the default configuration for remote reflection in EJOE.                                       #
# EJOE handles received reflection requests by clients VERY restrictive:                                 #
# Only packages/classes which are listed in this file                                                    #
# can be used for remote reflection.                                                                     #
# To overwrite the reflection settings, provide a customized                                             #
# ejoe-reflection-conf.properties on the classpath.                                                      #
##########################################################################################################
de.netseeker.ejoe.test.*
```

*„META-INF/ ejoe-reflection-conf.properties“*

Die rekursive Freigabe aller Klassen eines Packages sowie aller Sub-Packages und deren Klassen erfolgt durch Angabe des Stern-Symbols:

```
package.path.*
```

Die gezielte Freigabe einzelner Klassen erfolgt durch die vollständig qualifizierte Angabe von Packagepath und Klassennamen:

```
package.path.MyClass
```

### 6.9.2 DefaultRemotingHandler vs. AssistedRemoting-Handler

EJOE beinhaltet standardmäßig zwei konkrete Implementierungen für Remoting-Handler: de.netseeker.handler.DefaultRemotingHandler sowie de.netseeker.handler.AssistedRemotingHandler. Im wesentlichen erfüllen beide Handler die gleiche Aufgabe: Anfragen von Clients via Reflection an externe Geschäftslogik übergeben.

| | ***DefaulRemoting-Handler*** | ***AssistedRemoting-Handler*** |
|---|---|---|
| ***Reflection*** | ✓ | ✓ |
| ***Caching von Zielklassen*** | ✓ | ✓ |
| ***Caching von Zielmethoden*** | ✓ | ✓ |
| ***Generierung von Proxy-klassen für den beschleu-nigten, wiederholten Zugriff ohne Reflection*** | ✗ | ✓<br><br>pro Methode/Konstruktor wird eine gecachte Proxyklasse erzeugt |
| ***Benötigt weitere Ab-hängigkeiten*** | ✗ | ✓<br><br>es wird javassist benötigt |
| ***Mindestvoraussetzung JRE*** | 1.4 | 1.5 |

Einsatz des DefaultRemotingHandlers:

```
import de.netseeker.ejoe.EJServer;
import de.netseeker.ejoe.handler.DefaultRemotingHandler;
...
{
      ...
      EJServer server = new EJServer( new DefaultRemotingHandler() );
      server.start();
}
```

*Start EJServer mit DefaultRemotingHandler*

Einsatz des AssistedRemotingHandlers:

```
import de.netseeker.ejoe.EJServer;
import de.netseeker.ejoe.handler.AssistedRemotingHandler;
...
{
      ...
      EJServer server = new EJServer( new AssistedRemotingHandler() );
      server.start();
      ...
}
```

*Start EJServer mit AssitedRemotingHandler*

# 7 Die Clientkomponente: EJClient

EJClient ist ein TCP/IP basierter Netzwerkclient, welcher ausschließlich zur Kommunikation mit einem EJServer fähig ist.

Ebenso wie EJServer verfolgt auch EJClient den Ansatz, eine möglichst leichte Integration zu gewährleisten.

EJClient unterstützt:

- synchrone wie auch asynchrone Requests
- die Kapselung von Anfragen in das HTTP/1.0 Protokoll
- Kompression (wenn ebenfalls im angesprochenen EJServer aktiviert)
- dynamisches (Remote-)Classloading
- persistente Verbindungen (wenn ebenfalls im angesprochen EJServer aktiviert)
- frei wählbaren Verbindungstimeout
- dateibasierte als auch programmatische Konfiguration
- verschiedene (De-)Serialisierungsstrategien

Darüberhinaus ist EJClient selbst serialisierbar und kann bspw. in Containern vom Typ HttpSession[6] abgelegt werden.

6 siehe Java™ Servlet Specification 2.3

## 7.1 Erzeugen einer Clientinstanz

Ein Client ist immer eine Instanz der Klasse de.netseeker.ejoe.EJClient. Die Instanz muss durch Aufruf eines der angebotenen Konstruktoren erzeugt werden:

::de.netseeker.ejoe


*Abbildung 7.1: de.netseeker.ejoe.EJClient*

- `EJClient()`: Erzeugt eine neue Instanz von EJClient, vorkonfiguriert mit den Einstellungen aus der Datei „ejoe.properties“, welche auf dem Classpath verfügbar sein muss
- `EJClient(Properties properties)`: Erzeugt eine neue Instanz von EJClient, vorkonfiguriert mit den übergebenen Einstellungen. Dieser Konstruktor kann benutzt werden, wenn die Einstellungen für EJClient bereits aus einer anderen Konfigurationsdatei geladen wurden.
- `EJClient(String pathToConfigFile)`: Erzeugt eine neue Instanz von EJClient mit den Einstellungen aus der übergebenen Konfigurationsdatei.
- `EJClient(String host, int port)`: Erzeugt eine neue Instanz von EJClient, welche mit dem EJServer auf der Netzwerkadresse „host:port“ kommunizieren kann. Host kann dabei eine IP-Adresse oder ein DNS-Name sein.
- `EJClient(String host, int port, SerializeAdapter adapter)`: Erzeugt eine neue Instanz von EJClient, welche mit dem EJServer auf der Netzwerkadresse „host:port“ kommunizieren kann. Host kann dabei eine IP-Adresse oder ein DNS-Name sein. Für die Serialisierung der Anfragedaten wird der übergebene SerializeAdapter benutzt.
- `EJClient(String host, int port, SerializeAdapter adapter, boolean isPersistent, boolean isHttp, boolean useCompression)`: Erzeugt eine neue Instanz von EJClient, welche mit dem EJServer auf der Netzwerkadresse *„host:port“* kommunizieren kann. Host kann dabei eine IP-Adresse oder ein DNS-Name sein. Für die Serialisierung der Anfragedaten wird der übergebene SerializeAdapter benutzt. Je nachdem, ob *„isPersistent“* den Wert true oder false aufweist, wird eine persistente Verbindung zum EJServer benutzt oder nicht. Je nachdem, ob *„isHttp“* den Wert true oder false aufweist, werden Requests in HTTP-Post-Anfragen verpackt oder nicht. Je nachdem, ob *„useCompression“* den Wert true oder false aufweist, werden Requests komprimiert oder nicht.

## 7.2 Mit dem Server kommunizieren

Um mit dem Server zu kommunizieren, wird ein Objekt, der Request, an EJClient übergeben. EJClient führt die Abfrage am Server durch und liefert die Antwort des Servers zurück.

Für die Durchführung derartiger Requests stellt EJClient die Methode `execute(Object request)` bereit.

Der Request kann von jedem nicht-primitiven Typ sein und muss die vom jeweilig gewählten SerializeAdapter evtl. gestellten Anforderungen erfüllen.

Kommunikationsfehler werden als java.io.IOException, Fehler bei der serverseitigen Verarbeitung des Requests als java.rmi.RemoteException ausgelöst.

```
import de.netseeker.ejoe.EJClient;
...

Integer myIntegerRequest = new Integer(1);
String myStringRequest = „Hallo EJServer“;
Map myMapRequest = new HashMap();
SomeBean myBeanRequest = new SomeBean();
...
EClient client = new EJClient(...);
try
{
      client.execute( myIntegerRequest );
      client.execute( myStringRequest );
      client.execute( myMapRequest );
      client.execute( myBeanRequest );
      ...
}
catch(RemoteException re)
{
      //EJServer did return an Error
}
catch(IOException e)
{
      //connection or serialization issues maybe?
}
```

*Ausführen verschiedener Requests*

➔ Handelt es sich bei dem verwendeten EJServer um eine Instanz mit einem Remoting-Handler, muss der Request wie in Die RemotingHandler beschrieben in einen RemotingRequest eingebettet oder wie nachfolgend beschrieben mit Remoting über dynamische Proxies gearbeitet werden.

## 7.3 Remoting über dynamische Proxies

Alternativ zur unter Die RemotingHandler beschriebenen Methode, können auch dynamische Proxies verwendet werden um für den Programmierer leichter verwendbare, transparente Remote-Aufrufe zu erreichen.

Der Vorteil dieser Vorgehensweise liegt vorrangig darin, mit Remote-Methoden auf die gleiche Art und Weise umgehen zu können wie mit lokalen Methodenaufrufen. Darüber hinaus sind dynamische Proxies aber auch ein durchaus passabler Weg um eine Zwischenschicht einzuführen, welche die konkret verwendete Remote-Technologie von der eigenen Anwendung löst und somit einfacher austauschbar macht.

➔ Die Grundlagen der dynamischen Proxies werden bspw. im Artikel Java Dynamic Proxies: One Step from Aspect-oriented Programming auf DevX.com hervorragend erklärt.

EJOE beinhaltet einen Proxygenerator, welcher über Bereitstellung

- des Klassennamens der serverseitigen Remoteklasse
- eines Interfaces für die aufzurufenden Methodensignaturen
- einer EJClient-Instanz

einen dynamischen Proxy für die Remote-Aufrufe erzeugt. Der Proxygenerator verbirgt sich in der Klasse `de.netseeker.ejoe.RemotingService` und ist in der Lage über entsprechende Factory-Methoden dynamische Proxies für synchrone oder asynchrone Requests zu erzeugen.

```
import de.netseeker.ejoe.EJClient;
import de.netseeker.ejoe.RemotingService;
import de.netseeker.ejoe.test.service.ISimpleTypes;
import de.netseeker.ejoe.test.service.SimpleTypes;
...
EJClient client = new EJClient( "127.0.0.1", 12577 );
//SimpleTypes implementiert ISimpleTypes und stellt u.a. eine Implementation
//der Signatur #getString(String) bereit
ISimpleTypes service = (ISimpleTypes) RemotingService.createService(
                                                        SimpleTypes.class.getName(),
                                                        ISimpleTypes.class,
                                                        client );
String str = service.getString( "abcd" );
...
```

*Benutzung des Proxygenerators de.netseeker.ejoe.RemotingService*

## 7.4 Asynchrone Requests

Asynchrone Requests sind eine vereinfachte Form des Observer-Patterns und unterscheiden sich von normalen Anfragen hinsichtlich der verwendeten Art und Weise, wie und wann der Aufrufer eine Serverantwort von EJClient erhält. Bei normalen Anfragen wird die Methode `EJClient#execute()` aufgerufen und der Aufrufer muss solange warten, bis die Anfrage beendet ist. Bei asynchronen Requests startet der Aufrufer lediglich eine Anfrage, kann aber sofort, ohne dass die Anfrage beantwortet wurde, weiterarbeiten. Der Aufrufer wird später über das Eintreffen der Serverantwort oder ggf. einen Fehler informiert.

Für asynchrone Anfragen stellt EJClient die Methode `executeAsync` bereit.

```
public long executeAsync(java.lang.Object obj, EJAsyncCallback callback)
```

Diese Methode erwartet äquivalent zu `EJClient#execute()` ein Argument vom Typ java.lang.Object – den Request. Zusätzlich kommt mit einer Callbackinstanz vom Typ de.netseeker.ejoe.EJAsyncCallback ein weiteres Argument hinzu. Die Callbackinstanz erhält Benachrichtigungen, sobald der Server die Anfrage beantwortet hat oder ein Fehler aufgetreten ist.

Als direkten Rückgabewert liefert `executeAsync()` lediglich einen eindeutigen Identifikator für die gestartete Anfrage.

```
package de.netseeker.ejoe;

import java.io.IOException;

/**
 * A callback interface, which is able to process events of
 * asynchronous client requests.
 * @author netseeker
 * @since 0.3.9.1
 */
public interface EJAsyncCallback
{
    /**
     * This method will be called by EJClient whenever the related asynchronous
     * client request was successfully processed.
     * @param ident the unique identification number of the related asynchronous request
     * @param response the server response, can be null if no response was received
     */
    public void onRequestProcessed( long ident, Object response );
    /**
     * This method will be called by EJClient whenever an error occured
     * while processing the related asynchronous client request.
     * @param ident the unique identification number of the related asynchronous request
     * @param e the exception, which either has occured either in EJClient or was
```

```
     * received by EJClient from EJServer
     */
    public void onErrorOccured( long ident, IOException e);
}
```

*Das Interface de.netseeker.ejoe.EJAsyncCallback*

- ➔ EJClient startet pro asynchronem Request einen neuen Prozess. Werden parallel mehrerer asynchrone Requests in einem EJClient gestartet, so werden diese sequentiell abgearbeitet. Für echte Parallelität von Requests muss ein EJClient pro Request verwendet werden.
- ➔ In der Praxis sind sequentiell abgearbeitete, asynchrone Requests mit einem EJClient aufgrund des geringeren Ressourcenverbrauchs oftmals schneller verarbeitet als bei der Benutzung vieler paralleler EJClients.

## 7.5 Clienteinstellungen

| Einstellung | Beschreibung | Aufruf |
|---|---|---|
| Connection Timeout | Zeitspanne, die der Client auf Serverantworten wartet | `setConnectionTimeout(int)` |
| Persistente Verbindung | Bei einer persistenten Verbindung bleibt die Verbindung zwischen Client und Server zwischen den Anfragen geöffnet. Bei deaktiverter persistenter Verbindung wird die Verbindung nach Erhalt der Serverantwort wieder geschlossen und muss bei der nächsten Anfrage wieder aufgebaut werden, was oftmals mehr Zeit in Anspruch nimmt als die Anfrage selbst. | `enablePersistentConnection( boolean )` |
| HTTP Unterstützung | Kapselung von Anfragen in HTTP-Post-Requests.<br>Wenn Proxy- oder Firewallregeln eine direkte Kommunikation zwischen Client und Server nicht zulassen, ist es unter Umständen möglich diese Regeln durch Nutzung des HTTP Protokolls zu unterlaufen, da HTTP-Traffic oftmals gestattet ist | `enableHttpPackaging(boolean)` |
| Kompression | Bei aktivierter Kompression werden Anfragen mittels GZIP komprimiert | `enableCompression(boolean)`<br>oder<br>`enableCompression(int compressionLevel)` |

| Einstellung | Beschreibung | Aufruf |
|---|---|---|
| Dynamisches Classloading | Dynamisches Classloading erlaubt das Nachladen unbekannter Klassen vom Server ähnlich wie bei RMI | `enableRemoteClassloading()` |
| Verwendete (De-)Serialisierungs-strategie | | `setAdapterStrategy(int adapterStrategy)` |

## 7.6 Referenz ejoe.properties

| Einstellung | Beschreibung | Mögliche Werte | Standardwert | Pflicht |
|---|---|---|---|---|
| ejoe.adapter | Vollständiger Klassenname (package.ClassName) Serialize-Adapter, mit welchem Anfragen serialisiert und Serverantworten deserialisiert werden | Alle Klassen, die das Interface `de.netseeker.eoe.adaptr.SerializeAdapter` implementieren | `de.netseeker.ejoe.adapter.XStreamAdapter` bzw. `de.netseeker.ejoe.adapter.ObjectStreamAdapter` falls Xstream nicht gefunden wurde | ✓ |
| ejoe.host | DNS-Bezeichner oder IP-Adresse des zu anzusprechenden EJServers | gültige IPv4 oder IPv6 Adresse oder DNS-Name | localhost | ✗ |
| ejoe.port | Port auf dem der Server auf Clientanfragen wartet | 1 - 65535 | 12577 | ✗ |
| ejoe.connectionTimeout | Verbindungstimeout in Millisekunden | 0 - ? | 10000 | ✗ |
| ejoe.compression | Gibt an, ob der Server komprimierte Clientanfragen unterstützt | true\|false | false | ✗ |
| ejoe.persistentConnection | Gibt an, ob eine persistente Verbindung zum Server aufgebaut werden soll | true\|false | true | ✗ |
| ejoe.httpTunneling | Gibt an, ob Anfragen im HTTP Protokoll unterstützt werden | true\|false | false | ✗ |
| ejoe.remoteClassloader | Gibt an, ob unbekannte Klassen vom Server nachgeladen werden dürfen | true\|false | false | ✗ |

# 8 SerializeAdapter – Warum Serialisierung so wichtig ist

*Unter Serialisierung versteht man eine sequenzielle Abbildung von Objekten auf eine externe Darstellungsform. Das Umwandeln von Objekten in ein Format, welches es ermöglicht, diese in einer Nachricht an einen Empfänger zu schicken nennt man Marshalling. Den Prozess der Wiederherstellung der Objekte aus der Nachricht nennt man Unmarshalling.*

*Wenn bei EJOE also die Begriffe Serialisierung und Deserialisierung verwendet werden, ist konkret der Prozess des Marshalling bzw. Unmarshalling gemeint.*

Objekte, selbst die einfachsten Objekte, sind komplexe Strukturen.

Um Objekte aus der in Java zur Laufzeit erzeugten Struktur auf bzw. über ein Medium zu transportieren, müssen diese Objekte in ein speicherbares bzw. übertragbares Format transformiert werden.

Java stellt dafür die Mechanismen der Objektserialisierung sowie Long Term Persistence für Java Beans bereit. Beide stellen strenge Anforderungen an die zu transformierenden Objekte, die oftmals entweder nicht erfüllt werden können oder mit Laufzeiteinbußen durch zusätzliche Konvertierungen der in den Objekten beinhalteten Werte während der Objektverwendung zur Laufzeit erkauft werden müssen. Beispielhaft mag hier angeführt werden, dass sogar viele Containerklassen innerhalb der Java Runtime die gestellten Anforderungen nicht erfüllen.

Desweiteren fehlt den beiden genannten Mechanismen fast vollständig jegliche Unterstützung für moderne Formen der Datenmodellierung und des Datenbindings. Letzeres hat insbesondere in den letzten Jahren stark an Bedeutung gewonnen.

EJOE hat unter anderem zum Ziel möglichst viele Arten der Serialisierung zu unterstützen um Anwendungen zusätzlichen Implementierungsaufwand zu ersparen. Anwendungen sollen in die Lage versetzt werden, möglichst alle Arten von Objekten, eigene sowie Standardtypen aus der Java Runtime, über Netzwerk austauschen zu können ohne dafür an den Objekten Veränderungen vornehmen zu müssen.

An dieser Stelle kommen die sogenannten SerializeAdapter ins Spiel: Sie sind die Schnittstelle zwischen EJOE und verschiedensten Mechanismen und Frameworks zur Serialisierung.

## 8.1 Welche SerializeAdapter bringt EJOE mit?

| Adapter | Mechanismus/ Framework | Unterstützte Objekttypen | Daten-format | Lesen/ Schreiben |
|---|---|---|---|---|
| BetwixtAdapter | Betwixt Library | Beans gemäß der Java Beans Convention sowie gemischte Inhalte für die ein entsprechendes Mappingfile erstellt wird | XML | ✓/✓ |
| CastorAdapter | The Castor Project | Alle Objekte für die ein entsprechender Deskriptor vorliegt oder die per (Standard-)Reflection transformiert werden können | XML | ✓/✓ |
| HessianAdapter | Caucho Hessian | Unbekannt | Binär | ✓/✓ |
| JavaBeansXml-Adapter | Long Term Persistence for Java Beans | Alle Objekte, die der Java Beans Convention[7] entsprechen | XML | ✓/✓ |
| JavolutionAdapter | Javolution | Alle Objekte für die ein entsprechendes XML Mapping vorhanden ist | XML | ✓/✓ |
| Jaxb2Adapter | JAXB[8] | Alle Objekte, für die ein entsprechendes JAXB-Binding erzeugt wurde | XML | ✓/✓ |
| JsonLibAdapter | Json-lib | java.lang.String, java.lang.Character, char, java.lang.Number, byte, short, int, long, float, double, java.lang.Boolean, boolean, net.sf.json. JSONFunction, net.sf.json.JSONArray (object, string, number, boolean, function), net.sf.json.JSONObject | JSON | ✓/✓ |
| JsonToolsAdapter | Json Tools | All kind of POJOs | JSON | ✓/✓ |
| MyJsonAdapter | myJSON | unbekannt | JSON | ✓/✓ |
| JSXAdapter[9] | Java Serialization To XML | Laut Hersteller werden alle Objekte unterstützt | XML | ✓/✓ |
| SkaringaAdapter | http://skaringa.sourceforge.net/ | Unbekannt, laut Hersteller: „Skaringa works with all Plain Old Java Objects (POJOs), it is not limited to special cases...“ | XML | ✓/✓ |

7 Siehe bspw. Java in a Nutshell Third Edition Chapter 6: Java Beans
8 Siehe auch http://de.wikipedia.org/wiki/JAXB
9 JSX ist ein kommerzielles Projekt, für das Lizenzgebühren zu entrichten sind

| ***Adapter*** | ***Mechanismus/ Framework*** | ***Unterstützte Objekttypen*** | ***Daten-format*** | ***Lesen/ Schreiben*** |
|---|---|---|---|---|
| ObjectStream-Adapter[10] | Object Serialization | Objekte mit Standard-konstruktur, welche die Schnittstelle java.io.Serializeable implementieren | Binär | ✓/✓ |
| SojoAdapter | Simplify your Old Java Objects | unbekannt | XML | ✓/✓ |
| UTF8String-Adapter | | java.lang.String | UTF-8 enko-dierter String | ✓/✓ |
| XStreamAdapter[11] | XStream | Alle Objekte für die ein entsprechender spezieller Konverter vorliegt oder ein generallisierter Konverter benutzt werden kann | XML | ✓/✓ |
| XStreamJson-Adapter | XStream | Theoretisch wie XStreamAdapter, in der Praxis treten bei diversen Datentypen noch Schwierigkeiten auf | JSON | ✗/✓ |
| XmlBeansAdapter | XMLBeans | XMLBeans | XML | ✓/✓ |
| | | | | |

10 EJOE Standardadapter, falls XStream nicht verfügbar ist
11 EJOE Standardadapter, falls XStream verfügbar ist

## 8.2 Den zu verwendenden SerializeAdapter selbst bestimmen

Die Clientkompontente EJClient bringt spezielle Konstruktoren für die Auswahl eines speziellen SerializeAdapters mit:

```
public EJClient(String host, int port, final SerializeAdapter adapter)
...

public EJClient(String host, int port, final SerializeAdapter adapter,
                      boolean isPersistent, boolean isHttp,
                      oolean useCompression, boolean isDirect)
...

public EJClient(String host, int port, int classLoaderPort,
                      final SerializeAdapter adapter)
...
```

*Konstruktoren für die Auswahl eines SerializeAdapters*

➔ Pro Client kann nur ein SerializeAdapter benutzt werden, eine spätere Änderung zur Laufzeit ist nicht möglich.

Wird kein Adapter gesetzt, benutzt EJClient automatisch den Standardadapter:

- XStreamAdapter, falls XStream verfügbar ist
- ansonsten ObjectStreamAdapter

## 8.3 Serialisierungsstrategien

*Eine Serialisierungsstrategie entscheidet darüber, in welchem Umfang EJServer und EJClient Serialisierungsaufgaben übernehmen sollen.*

Auch wenn einer der Grundgedanken hinter EJOE der einfache, unkomplizierte Austausch von Objekten über Netzwerk ist, bietet EJOE dennoch Eingriffsmöglichkeiten um neben der Art der Serialisierung (siehe SerializeAdapter) auch Einfluss auf den Umfang der Serialisierung nehmen zu können.

EJOE, sowohl EJClient als auch EJServer, versendet Daten immer in der sogenannten serialisierten Form, welche durch den jeweiligen SerializeAdapter bestimmt wird. Unter gewissen Umständen kann es sinnvoll sein entweder komplett auf die Verwendung der durch EJOE automatisch, intern erzeugten serialisierten Form zu verzichten oder aber anstelle deserialisierter Serverantworten in Objektform Zugriff auf die serialisierte Form zur Weiterverarbeitung zu erhalten.

Arbeiten bspw. sowohl ServerHandler als auch Clientanwendung mit Bytearrays und/oder Objekten vom Typ java.nio.ByteBuffer, dann ist es unnötig diese Daten in eine transportable Form zu bringen, da sowohl Bytearrays als auch ByteBuffer bereits transportabel sind.

Evtl. ist aber auch wünschenswert lediglich die Serverantworten von EJServer mittels eines XML-basierten SerializeAdapters in XML-Form zu serialisieren, an den EJClient zu übermitteln und dann in der serialisierten XML-Form an den Aufrufer zurückzugeben, da die XML-Form via XSLT oder anderer auf XML aufsetzender Mechanismen direkt weiterverarbeitet werden kann.

### 8.3.1 Standardserialisierung

Bei der Standardserialisierung werden alle Objekte serialisiert bevor sie

- als Request an den Server versendet werden
- als Response an den Client gesendet werden

und deserialisiert bevor sie

- an den ServerHandler übergeben werden
- an den Aufrufer von EJClient zurückgeliefert werden.

Sowohl Clientanwendung als auch ServerHandler bemerken von den Serialisierungs- und Deserialisierungsvorgängen nichts. Sie haben keinen Zugriff auf die serialisierte Form eines Objekts, weder auf serialisierte Request- noch auf serialisierte Responsedaten.

➔ Wird keine Serialisierungsstrategie gesetzt, ist automatisch die Standard-serialisierung aktiv.

Die Standardserialisierung wird durch die Konstante **ADAPTER_STRATEGY_DEFAULT** in der Klasse de.netseeker.ejoe.EJConstants abgebildet.

```
import de.netseeker.ejoe.EJClient;
import de.netseeker.ejoe.EJConstants;
...
      EJClient client = new EJClient(...);
      client.setAdapterStrategy( EJConstants. ADAPTER_STRATEGY_DEFAULT );
...
```

*manuelles Setzen der Standardserialisierung*

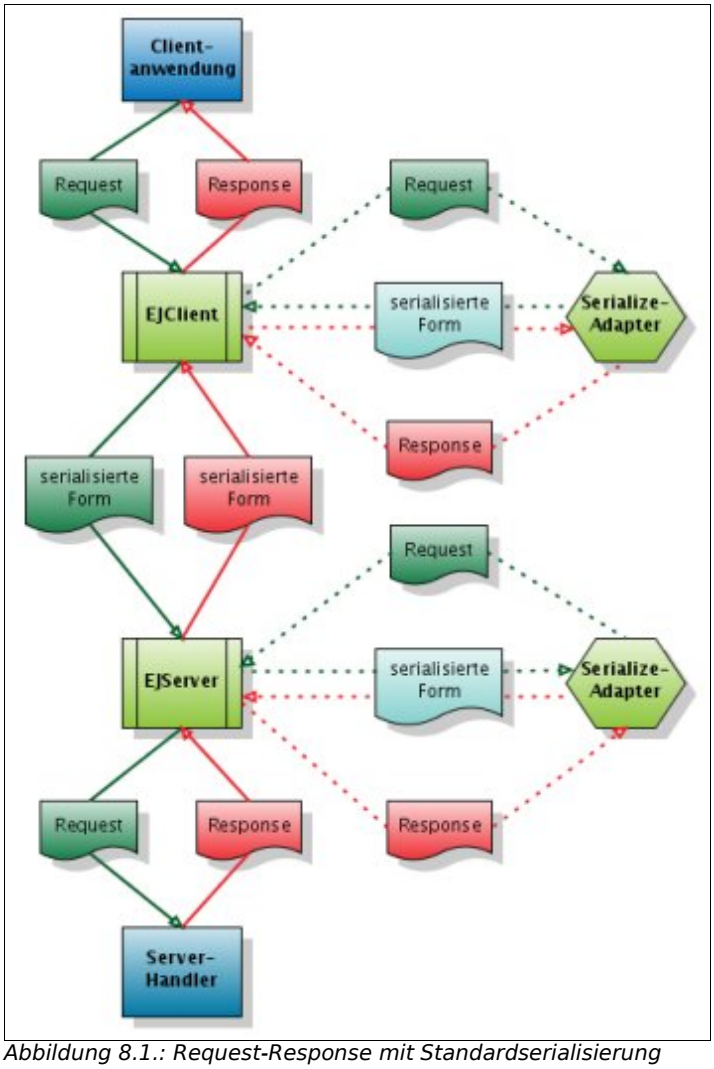



*Abbildung 8.1.: Request-Response mit Standardserialisierung*

### 8.3.2 gemischte Serialisierung

EJClient bietet die Möglichkeit, sich anstelle deserialisierter Objekte die serialisierte Form einer Serverantwort zurückliefern zu lassen. Dies kann insbesondere dann sinnvoll sein, wenn

- die Serverantwort ohnehin in serialisierter Form auf dem Client abgespeichert werden soll, zum Beispiel in einer Datenbank oder in einer Datei
- die Serverantwort mit einem Postprozessor[12] wie zum Beispiel einem XSLT-Prozessor[13] nachbearbeitet bzw. weiterverarbeitet werden soll, der nicht mit deserialisierten Objekt sondern mit der Ausgabeform des verwendeten SerializeHandlers umgehen kann.

12 siehe bspw. Wikipedia http://de.wikipedia.org/wiki/Postprozessor
13 siehe bspw. Wikipedia http://de.wikipedia.org/wiki/Xslt

Die gemischte Serialisierungsstrategie wird durch die Konstante **ADAPTER_STRATEGY_MIXED** in der Klasse de.netseeker.ejoe.EJConstants abgebildet.

```
import de.netseeker.ejoe.EJClient;
import de.netseeker.ejoe.EJConstants;
...
      EJClient client = new EJClient(...);
      client.setAdapterStrategy( EJConstants. ADAPTER_STRATEGY_MIXED );
...
```

*Aktivierung der gemischten Serialisierungsstrategie*

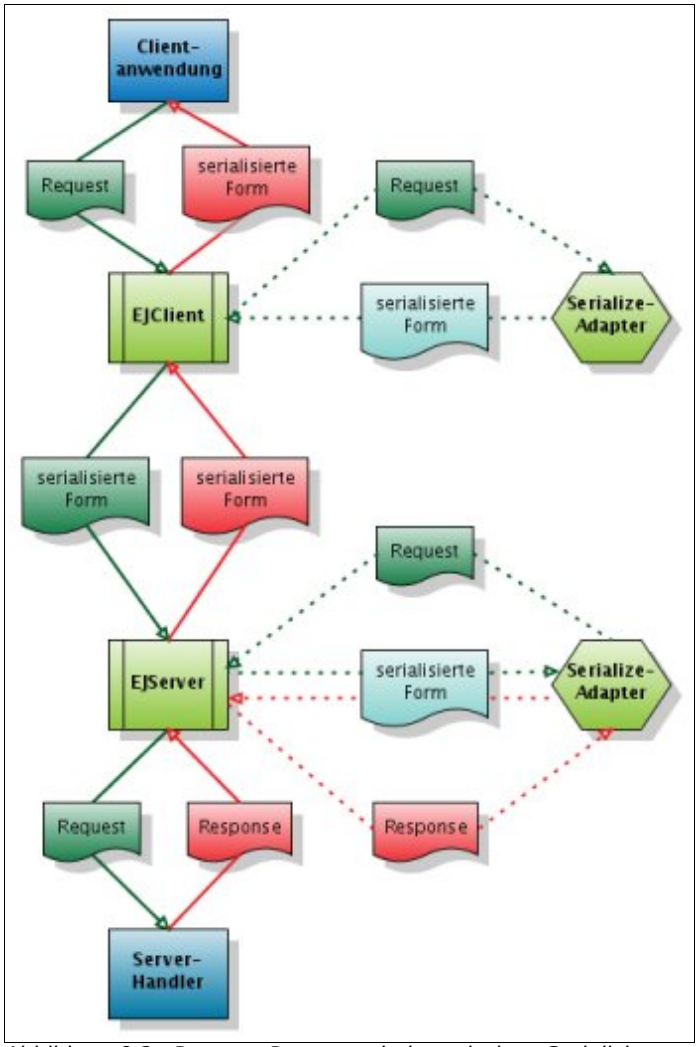



*Abbildung 8.2.: Request-Response bei gemischter Serialisierung*

### 8.3.3 direkter Austausch von Objekten vom Typ java.nio.ByteBuffer

In Fällen, in denen sowohl ServerHandler als auch Clientanwendung Request- und Responsedaten als java.nio.ByteBuffer verarbeiten können, ist es möglich die Serialisierung komplett abzuschalten, da Objekte vom Typ java.nio.ByteBuffer bereits ohne weitere Serialisierung über Netzwerk transportabel sind. Es wird keine serialisierte Form für den Transport dieser Objekte benötigt.

Die direkte Serialisierungsstrategie wird durch die Konstante **ADAPTER_STRATEGY_DIRECT** in der Klasse de.netseeker.ejoe.EJConstants abgebildet.

```
import de.netseeker.ejoe.EJClient;
import de.netseeker.ejoe.EJConstants;
...
      EJClient client = new EJClient(...);
      client.setAdapterStrategy( EJConstants. ADAPTER_STRATEGY_DIRECT );
...
```

*Deaktivierung der Serialisierung*

## *8.4 Adapterzauberei – wie der Server den richtigen Adapter auswählt*

Adapter werden grundsätzlich immer vom Client angefordert. Der Client muss bei Anfragen also eine Information über den gewünschten SerializeAdapter mitliefern. Während des initialen Handshakes zwischen Client und Server, teilt der Cient dem Server den Klassenname des gewünschten Adapter mit, ab dann kann der Server serialisierte Daten des Clients deserialisieren und verwenden.

Der Server erfragt die zu verwendende Instanz eines Adapters über seinen Klassennamen bei der AdapterFactory. Ist noch keine Instanz für den gewünschten Adapter vorhanden, erzeugt die Factory eine neue Instanz und speichert diese für nachfolgende Anfragen. Ein Adapter liegt innerhalb von EJOE somit immer nur in exakt einer Instanz vor und muss über einen Standardkonstruktor initialisierbar sein.

Für spezielle Adapter, welche nicht mit diesem Mechanismus in einen einsatzbereiten Zustand versetzt werden können, bietet die AdapterFactory entsprechende Registrierungsfunktionen an. (siehe Was tun, falls ein Adapter zusätzliche Konfiguration benötigt? )

## 8.5 Adapter aktivieren und deaktivieren

*Unter Umständen kann es notwendig werden bestimmte Adapter serverseitig zu verbieten. Ein solches Verbot bedeutet, dass alle Client-Anfragen für diesen Adaptertyp mit einer entsprechenden Fehlermeldung beantwortet werden. Clients, welche den entsprechenden Adapter verwenden, sind dann nicht mehr in der Lage Daten mit dem Server auszutauschen.*

EJOE verwaltet Adapter in der Konfigurationsdatei *ejoe-adapter-conf.properties*. Zum Auffinden dieser Datei wird zuerst der Classpath abgesucht und - falls im Classpath eine entsprechende Datei gefunden wird – diese geladen. Es werden alle SerializeAdapter zugelassen, die in der Datei aufgeführt sind und den Wert `true` zugewiesen haben.

Anschließend lädt EJOE die Datei *META-INF/ejoe-adapter-conf.properties* und führt einen Abgleich zwischen beiden Konfigurationen durch. Ist ein SerializeAdapter in beiden Dateien genannt, wird immer der Eintrag aus *ejoe-adapter-conf.properties* aus dem Classpath übernommen. SerializeAdapter, welche ausschließlich in *META-INF/ejoe-adapter-conf.properties* mit dem Wert `true` aufgeführt sind, werden zusätzlich geladen.

Um einen der in *META-INF/ejoe-adapter-conf.properties* eingetragenen Serialize-Adapter zu deaktivieren, muss die Datei *ejoe-adapter-conf.properties* auf dem Classpath verfügbar gemacht werden und für den betroffenen Adapter den Wert `false` zuweisen.

```
#############################################################################################
# This is the default adapter configuration for EJOE                                        #
# EJOE handles adapters requested by clients VERY restrictive:                              #
# Only adapter which are listed in this file and are set to enabled (true)                  #
# will be used.                                                                             #
# To overwrite the adapter settings provide a customized                                    #
# ejoe-adapter-conf.properties on the classpath.                                            #
#############################################################################################

#package.class=true|false
de.netseeker.ejoe.adapter.BetwixtAdapter=true
de.netseeker.ejoe.adapter.CastorAdapter=true
de.netseeker.ejoe.adapter.HessianAdapter=true
de.netseeker.ejoe.adapter.JavaBeansXmlAdapter=true
de.netseeker.ejoe.adapter.JavolutionAdapter=true
de.netseeker.ejoe.adapter.Jaxb2Adapter=true
de.netseeker.ejoe.adapter.ObjectStreamAdapter=true
de.netseeker.ejoe.adapter.UTF8StringAdapter=true
de.netseeker.ejoe.adapter.XStreamAdapter=true
...
```

*META-INF/ejoe-adapter-conf.properties*

## 8.6 Was tun, falls ein Adapter zusätzliche Konfiguration benötigt?

Für den Fall, dass ein Adapter wie bspw. der CastorAdapter, weitere Angaben für einen sinnvollen Einsatz benötigt, kann die sog. AdapterFactory benutzt werden.

Die AdapterFactory wird von EJOE benutzt um Adapter zur Verwendung zu registrieren, zu cachen und/oder über Synonyme erreichbar zu machen.

EJOE erzeugt für alle eingetragenen Adapter in *ejoe-adapter-conf.properties* durch Aufruf des Standardkonstruktors eine Instanz und registriert diese automatisch unter dem verwendeten Klassennamen an der AdapterFactory.

So wird bspw. auch für den CastorAdapter eine derartige Instanz erzeugt, welche jedoch mangels Angabe einer für Castor typischen Mappingdatei oftmals nicht sinnvoll nutzbar ist.

Das Überschreiben der Standardinstanz eines Adapters erfolgt mittels Aufruf der Methode *registerAdapter(SerializeAdapter)*. Ein derartiger Aufruf registriert eine Adapterinstanz unter Ihrem Klassennamen erstmalig oder erneut bei EJOE. Für Clientanfragen, welche den benutzten Adapter anfordern, wird ab dann die manuell registrierte Instanz benutzt.

```
//registriert einen mittels Mappingdatei konfigurierten CastorAdapter anstelle
//des standardmäßig geladenen, unkonfigurierten CastorAdapters
AdapterFactory.registerAdapter( new CastorAdapter( "castor-mapping.xml" ) );
```

*Registrierung eines Adapters*

## 8.7 Eigene SerializeAdapter

EJOE bringt zwar von Hause aus bereits eine Vielzahl vorgefertigter Adapter mit, es kann jedoch notwendig sein einen eigenen Adapter zu implementieren um einen ansonsten (noch) nicht für EJOE nutzbaren Serialisierungsmechanismus zu unterstützen.

EJOE versucht die Integration neuer Adapter so einfach wie möglich zu gestalten und stellt lediglich zwei, in bestimmten Fällen drei, Anforderungen an einen Adapter. SerializeAdapter müssen:

- das Interface de.netseeker.ejoe.adapter.SerializeAdapter implementieren,
- serverseitig in die Datei *ejoe-adapter-conf.properties* eingetragen werden,
- falls notwendig mithilfe der AdapterFactory (de.netseeker.ejoe.adapter.AdapterFactory) registriert werden.

```
public interface SerializeAdapter extends Serializable
{
    /**
     * Deserializes an object out of an given InputStream
     * ...
     */
    public Object read( InputStream in ) throws Exception;

    /**
     * Serializes an object into an output stream
     * ...
     */
    public void write( Object obj, OutputStream out ) throws Exception;

    /**
     * Signals a change of the ContextClassLoader of the current thread to the adapter.
     * This method will be called when the EJOE client is started with support for remote
     * classloading.
     * ...
     */
    public void handleClassLoaderChange( ClassLoader classLoader );

    /**
     * Signals if this adapter has problems with EJOE's UncloseableInputStreams and
     * requires a really closed Stream (which EJOE prevents to ensure that an adapter
     * can not close the underlying socket unintentionally). If the adapter requires closed
     * streams, EJOE will append a custom EOF signature at the end of the stream and
     * ...
     */
    public boolean requiresCustomEOFHandling();
    /**
     * Indicates that the adapter uses a StreamBuffer mechanism and doesn't require
     * a buffered input stream
     * ...
     */
    public boolean isSelfBuffered();

    /**
     *...
     */
    public String getContentType();
}
```

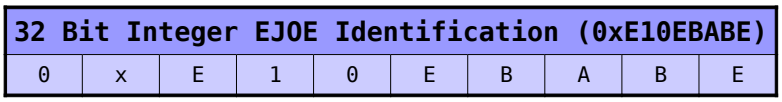
*Das Interface SerializeAdapter*

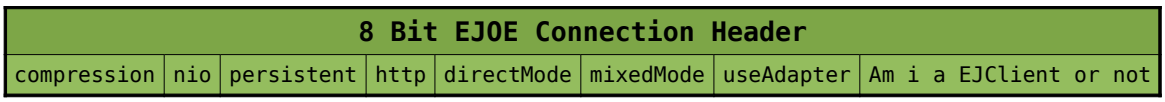
➔ EJOE bietet zusätzlich die Möglichkeit, die abstrakte Basisklasse de.netseeker.ejoe.adapter.BaseAdapter abzuleiten, welche Standardimplementierungen für einige der durch das Interface erzwungenen Methoden bereitstellt.

# 9 Anhang

## 9.1 Das EJOE-Protokoll

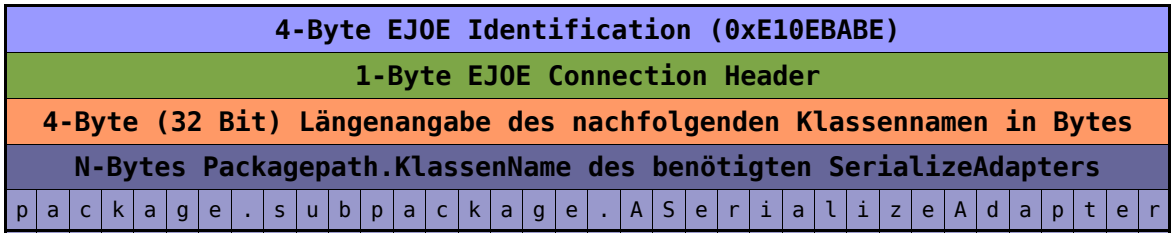
**Besondere Container:**



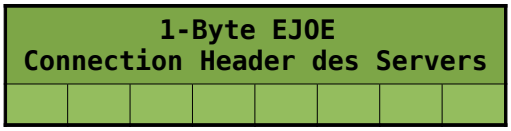




**Handshake:**

Der Client sendet einen Handshake in der Form:

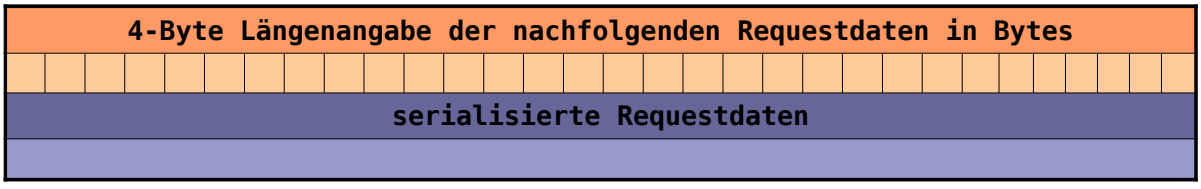



Der Server beantwortet den Handshake mit 1-Byte EJOE ConnectionHeader:

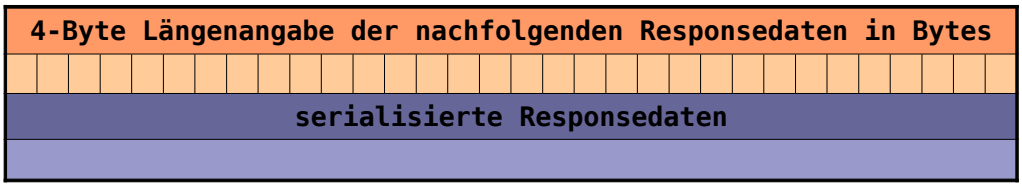



**Request/Response:**

Der Client sendet einen Request in der Form:




Der Server antwortet mit gleichartig aufgebauter Response:



## 9.2 EJOE HTTP-Protokoll Details

### Handshake

Client sendet Handshake:

```
HEAD /11010011/de/netseeker/ejoe/adapter/XStreamAdapter HTTP/1.0
Host: 127.0.0.1:19272
User-Agent: EJClient/0.3.9.1
Content-Type: text/xml
Connection: close
Content-Length: 0
```

Server anwortet mit 1-Byte EJOE ConnectionHeader:

```
HTTP/1.0 200 OK
Server: EJServer/0.3.9.1
Date: So, 19 Nov 2006 02:11:18 CET
Content-Type: application/octet-stream
Connection: close
Content-Length: 1

?
```

### Request/Response

Client sendet Request:

```
POST /01010011/de/netseeker/ejoe/adapter/XStreamAdapter HTTP/1.0
Host: 127.0.0.1:15622
User-Agent: EJClient/0.3.9.1
Content-Type: text/xml
Connection: close
Content-Length: 1014

<map><entry><string>KEY3</string><big-decimal>102</big-
decimal></entry><entry><string>KEY1</string><strin
g>Hello</string></entry><entry><string>KEY2</string><int>101</int></entry><entry><string>KEY4</st
ring><de
.netseeker.ejoe.test.ObjectBean><theByte>101</theByte><theShort>102</theShort><theInt>103</theInt
><theFlo
at>104.0</theFloat><theString>Hallo</theString><theDate>2006-11-19 02:11:13.359
CET</theDate><theArray><s
tring>string1</string><string>string2</string><string>string3</string></theArray><theList><byte>1
01</byte
><short>102</short><int>103</int><date
reference="../../theDate"/></theList><theMap><entry><string>theDat
e</string><date
reference="../../../theDate"/></entry><entry><string>theInteger</string><short>102</short
></entry><entry><string>theDecimal</string><byte>101</byte></entry><entry><string>theDouble</stri
ng><int>
103</int></entry></theMap><theSet><date
reference="../../theDate"/><short>102</short><int>103</int><byte>
101</byte></theSet></de.netseeker.ejoe.test.ObjectBean></entry></map>
```

Server sendet Response:

```
HTTP/1.0 200 OK
Server: EJServer/0.3.9.1
Date: So, 19 Nov 2006 02:11:19 CET
Content-Type: text/xml
Connection: close
Content-Length: 1014

<map><entry><string>KEY3</string><big-decimal>102</big-
decimal></entry><entry><string>KEY1</string><strin
g>Hello</string></entry><entry><string>KEY2</string><int>101</int></entry><entry><string>KEY4</st
ring><de
.netseeker.ejoe.test.ObjectBean><theByte>101</theByte><theShort>102</theShort><theInt>103</theInt
><theFlo
at>104.0</theFloat><theString>Hallo</theString><theDate>2006-11-19 02:11:13.359
CET</theDate><theArray><s
tring>string1</string><string>string2</string><string>string3</string></theArray><theList><byte>1
01</byte
><short>102</short><int>103</int><date
reference="../../theDate"/></theList><theMap><entry><string>theDat
e</string><date
reference="../../../theDate"/></entry><entry><string>theInteger</string><short>102</short
></entry><entry><string>theDecimal</string><byte>101</byte></entry><entry><string>theDouble</stri
ng><int>
103</int></entry></theMap><theSet><date
reference="../../theDate"/><short>102</short><int>103</int><byte>
101</byte></theSet></de.netseeker.ejoe.test.ObjectBean></entry></map>
```

### 9.2.1 Kommunikation mit einem alternativen HTTP-Client

Bei der Kommunikation mit einem alternativen HTTP-Client ergibt sich folgende Schwierigkeit:

*Der Client hat keine Kenntnis über den EJOE spezifischen HTTP-HEAD Handshake und schickt anstelle des Handshakes sofort einen Datenrequest.*

Um damit umgehen zu können stellt EJServer an Nicht-EJOE-Clients folgende Anforderungen:

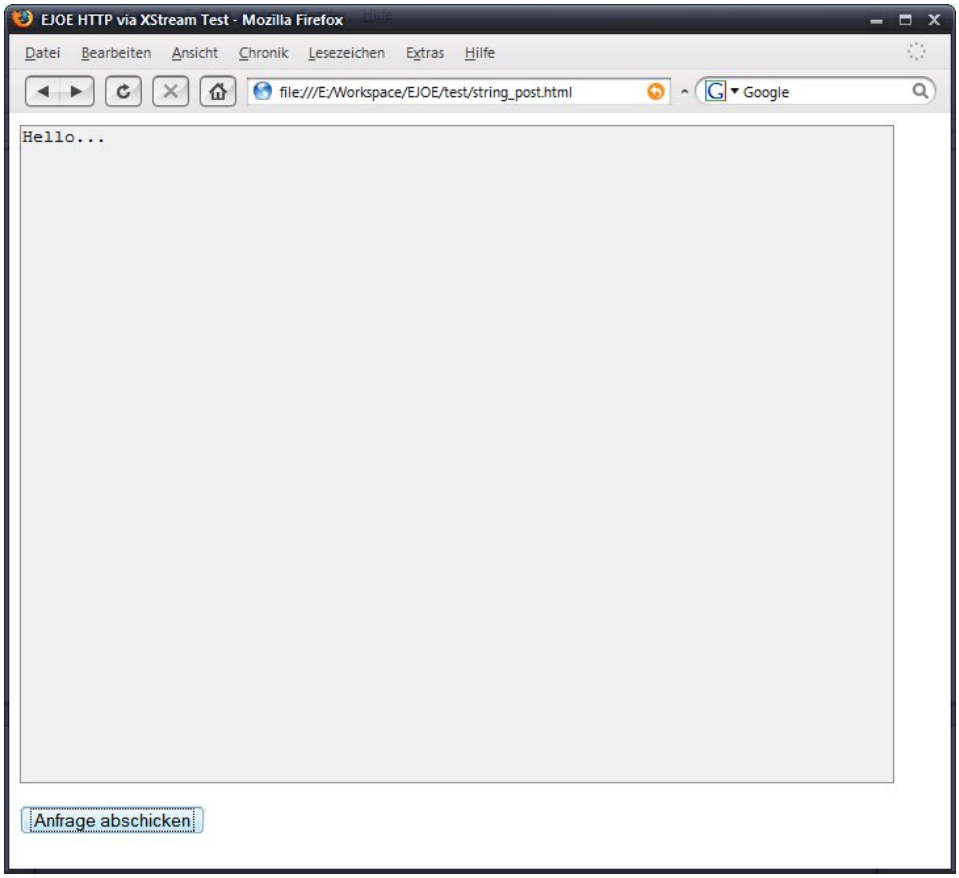
- die URI muss immer mit einem „/“ sowie nachfolgend den acht Bit des EJOE-ConnectionHeaders beginnen, die Bitwerte werden als Folge von acht Zahlen, jeweils 0|1 dargestellt
- falls nicht der Standard-SerializeAdapter benutzt werden soll, muss die URI nach dem ConnectionHeader mit einem „/“ sowie nachfolgend dem vollständigen Klassennamen des zu verwendenden SerializeAdapters fortgesetzt werden.
- In der URI müssen alle Punkte „.“ durch Slashes „/“ ersetzt sein.

➔ HTTP-Clients **können** Daten für den Server in eine POST-Variable mit Namen ***ejdata*** verpacken. Dies ist in der Regel insbesondere bei Requests aus HTML-Formularen notwendig.

EJOE-Request aus einem HTML-Formular:

```
<html>
  <head>
    <title>EJOE HTTP via XStream Test</title>
  </head>
  <body>
  <!-- |compression|nio|persistent|http|directMode|mixedMode|useAdapter|am I a EJClient| -->
  <form action="http://localhost/01010010/de/netseeker/ejoe/adapter/UTF8StringAdapter"
   method="post">
      <textarea name="ejdata" cols="80" rows="30" >Hello...</textarea><p/>
      <input type="submit">
  </form>
  </body>
</html>
```

Das Formular im Browser:



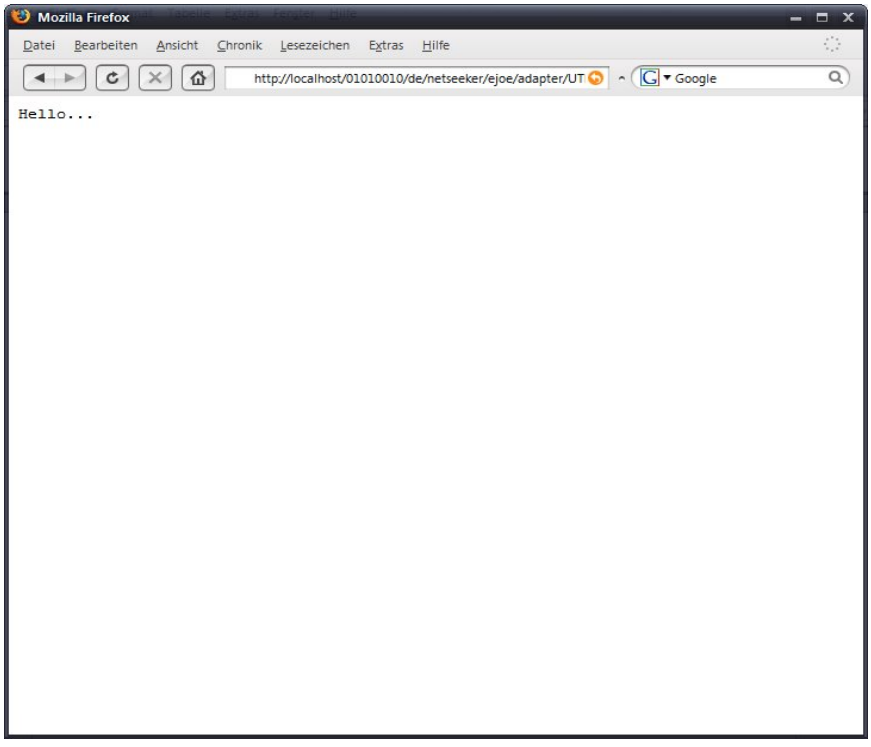
Der aus dem Formular abgeschickte HTTP-Request:

```
POST /01010010/de/netseeker/ejoe/adapter/UTF8StringAdapter HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.8.1) Gecko/20061010 Firefox/2.0
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 30
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

ejdata=Hello...
```

EJServer erkennt die Anfrage, ordnet den Browser als Nicht-EJOE-Client ein und reagiert wie viele andere HTTP-Server:

```
HTTP/1.0 200 OK
Server: EJServer/0.3.9.1
Date: So, 19 Nov 2006 02:41:54 CET
Content-Type: text/plain
Content-Encoding: x-gzip
Connection: keep-alive
Content-Length: 28

????????????????????????????
```

Die ServerResponse im Browser:



### 9.2.2 Ausblick HTTP-Unterstützung

Selbst die lediglich partiell vorhandene Unterstützung der beiden HTTP-Protokoll-versionen erlaubt bereits die Nutzung alternativer Clients mit einem EJServer.

Mit Benutzung eines Character-, XML- oder JSON-basierten SerializeHandlers lassen sich EJServer-Funktionen jetzt auch via Aufruf aus HTML-Formularen sowie bei Verwendung von Ajax[14] nutzen.

Zum Zeitpunkt der Erstellung dieser Einführung ist die HTTP-Unterstützung in EJOE noch neu und muss sich erst noch im Einsatz beweisen.

## *9.3 Crispy Extension*

CRISPY = **C**ommunication per **R**emote **I**nvocation for different kinds of **S**ervices via **P**rox**Y**s

Laut Homepage http://crispy.sourceforge.net ist Crispy eine API mit der Intention einen einheitlichen und einfachen Einstieg zu einer großen Anzahl von Transporttechnologien wie beispielsweise RMI, JAX-RPC und XML-RPC dazustellen.

Crispy geht damit mit dem Ansatz abstrahierter, auf einheitliche Art und Weise aufrufbarer Remote-Services einher.

Im Gegensatz zu WSIF (**W**eb **S**ervice **I**nvocation **F**ramework) wählt Crispy einen eher programmatischen Ansatz ohne die Programmiersprache Java hin zu ergänzenden Beschreibungstechnologien und Formaten zu verlassen.

Crispy ist sehr einfach integrierbar und kommt mit einer umfangreichen Dokumentation auf der Homepage http://crispy.sourceforge.net einher.

EJOE unterstützt Crispy mittels eines dynamischen Proxies[15]. EJOE wird dabei als Server- und Transporttechnologie verstanden, auf die mittels Crispy ebenso wie auf beispielsweise RMI oder JBoss Remoting zugegriffen werden kann.

➔ Die Crispy-Erweiterung setzt die Verwendung eines RemotingHandlers im EJServer voraus!

14 **A**synchronous **J**avaScript **a**nd **X**ML, siehe bspw. http://de.wikipedia.org/wiki/Ajax_%28Programmierung%29

15 Siehe http://crispy.sourceforge.net/guide_developer.html

Die Benutzung eines EJClients über Crispy mittels der Klasse **de.netseeker.ejoe.ext.crispy.EJExecutor** gestaltet sich dabei sehr einfach:

```
import de.netseeker.ejoe.ext.crispy.EJExecutor;
import de.netseeker.ejoe.adapter.XStreamAdapter;
import net.sf.crispy.impl.ServiceManager;
...
{
  Properties prop = new Properties();
  prop.put(Property.EXECUTOR_CLASS, EJExecutor.class.getName());
  prop.put(EJExecutor.EJOE_SERIALIZATION_ADAPTER, XStreamAdapter.class.getName());
  prop.put(Property.REMOTE_URL_AND_PORT, „socket://localhost:12577“);
  prop.put(EJExecutor.EJOE_USE_PERSISTENT_CONNECTION, Boolean.toString(true));
  ServiceManager manager = new ServiceManager(prop);
  Echo e = (Echo) manager.createService(Echo.class);
  e.echo( „Hallo“ );
...
```

*Verwendung des EJExecutors anhand einer einfachen Echoimplementierung*

EJExecutor unterstützt die folgenden Einstellungen:

| ***Einstellung*** | ***Beschreibung*** | ***Mögliche Werte*** |
|---|---|---|
| **EJOE_SERIALIZATION_ADAPTER** | Von EJOE zu verwendender SerializeAdapter | „Package.ClassName“ einer Instanz von de.netseeker.ejoe. adapter.SerializeAdapter |
| **EJOE_USE_PERSISTENT_CONNECTION** | Entscheidet über die Verwendung einer persistenten Verbindung zum EJServer | „true“|“false“ |
| **EJOE_CONNECTION_TIMEOUT** | Setzt den Connection Timeout in Millisekunden | 1-? |
| **EJOE_USE_COMPRESSION** | Entscheidet über die Verwendung von GZIP bei der Übertragung von Daten | „true“|“false“ |
| **EJOE_USE_REMOTE_CLASSLOADER** | Entscheidet, ob EJClient unbekannte Klassen innerhalb der Serverantworten vom EJServer nachladen darf | „true“|“false“ |

*Tabelle 9.1.*

Weitere Details zum Einsatz von Crispy-Extensions können der allgemeinen Crispy-Dokumentation entnommen werden.

## 9.4 WSIF-Untersützung

WSIF = **W**eb **S**ervices **I**nvocation **F**ramework

Mit WSIF ist es unter anderem möglich nicht-serviceorientierte Architekturen über WSDL Beschreibungen servicefähig zu machen - vorausgesetzt es existieren die entsprechenden WSDL-Erweiterungen sowie eine entsprechende Providerimplementierung.

Weitere Details zu WSIF sind auf der Projektseite http://ws.apache.org/wsif/ abrufbar. Ein ausgezeichnetes deutschsprachiges Tutorial wurde im Oktober 2003 vom Java Magazin unter http://javamagazin.de/itr/online_artikel/psecom,id,424,nodeid,11.html veröffentlicht.

EJOE beinhaltet sowohl einen WSIF-Provider als auch WSDL-Erweiterungen um Operationen, welche über EJServer als Remote-Services zur Verfügung gestellt werden, mittels WSIF nutzbar zu machen.

### 9.4.1 EJOE WSDL Erweiterungen

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions .... />
      <!-- binding declns -->
      <binding name="EJOEBinding" type="...">
             <ejoe:binding />
             <format:typeMapping encoding="..." style="uri">
                    <format:typeMap typeName="qname" formatType="nmtoken" />*
             </format:typeMapping>
             <operation>*
                    <ejoe:operation
                           className="nmtoken"
                           methodName="nmtoken" parameterOrder="nmtoken"?
                           invocationType="reflection|custom" />
                    <input name="nmtoken"? />?
                    <output name="nmtoken"? />?
             </operation>
      </binding>

      <!-- service decln -->
      <service name="..." ...>
             <port name="..." binding="...">
                    <ejoe:address
                           adapterClass="nmtoken"
                           ejoeServerURL="socket|http://host:port" useCompression="true|false"
                           usePersistentConnection="true|false" timeout="long"/>
             </port>
      </service>
</definitions>
```

| ***Binding Element*** | ***Beschreibung*** |
|---|---|
| `ejoe:binding` | Gibt an, dass es sich beim Binding um ein EJOE-Binding handelt |
| `format:typeMapping` | Siehe http://ws.apache.org/wsif/providers/wsdl_extensions/formattypemapping_extension.html<br><br>Typemappings können der Vollständigkeit halber angegeben werden, werden von EJOE jedoch nicht benötigt! EJOE mappt ServerResponses nicht, wie bspw. Axis in via WSDL2Java generierte Objekte, sondern benutzt direkt die vom Server zurückgegebenen Typen. Diese müssen entweder im ClassPath des Clients verfügbar sein oder über den EJClassloader nachgeladen werden. |
| `ejoe:operation` | Dieses Element mappt eine abstrakte WSDL-Operation auf eine via EJServer remote verfügbar gemachte Operation.<br><br>Das Attribut `className` beinhaltet den vollständigen Klassennamen inklusive Packageangabe der remote im EJServer aufzurufenden Java-Klasse.<br><br>Das Attribut `methodName` entspricht dem Name der remote in der unter `className` angegebenen Klasse im EJServer aufzurufenden Methode bzw. des Konstruktors.<br><br>Das Attribut `parameterOrder` hat denselben Zweck – und überschreibt – die angegebene Parameter Reihenfolge in der übergeordneten, abstrakten WSDL-Operation. Es definiert die Reihenfolge der Input-Message-Parts für den Aufruf. Im Zusammenhang mit RemoteReflection entspricht die Reihenfolge der Argumente der Remote-Methode.<br><br>Das Attribut `invocationType` beschreibt die Art des verwendeten ServerHandlers, derzeit wird allerdings nur der Wert „reflection“ für RemotingHandler unterstützt. |
| `ejoe:address` | Dieses Element ist eine WSDL-Erweiterung innerhalb des Port-Elements und erlaubt die Angabe des zu verwendenden EJServers als Endpoint sowie die zu verwendende Client-Konfiguration.<br><br>Über das Attribut `adapterClass` kann der zu verwendende SerializeAdapter als Klassenname inklusive Packageangabe spezifiziert werden.<br><br>Das Attribut `useCompression` entscheidet über die Verwendung der GZIP-Kompression während der Client-Server-Kommunikation.<br><br>Das Attribut `usePersistentConnection` entscheidet über die Verwendung einer persistenten Verbindung zum EJServer. |

| Binding Element | Beschreibung |
|---|---|
| | Über das Attribut `timeout` kann der zu verwendende Connection Timeout in Millisekunden angegeben werden.<br><br>Schlussendlich wird über das Attribut `ejoeServerURL` die Verbindung zum EJServer beschrieben. Es handelt sich dabei um eine URI-Angabe in der Form `Protokoll://Host:Port`. Protokoll kann die Werte `socket` oder `http` annehmen. Die Angabe `http` aktiviert die Kapselung der Clientanfragen in das HTTP/1.0 Protokoll. |

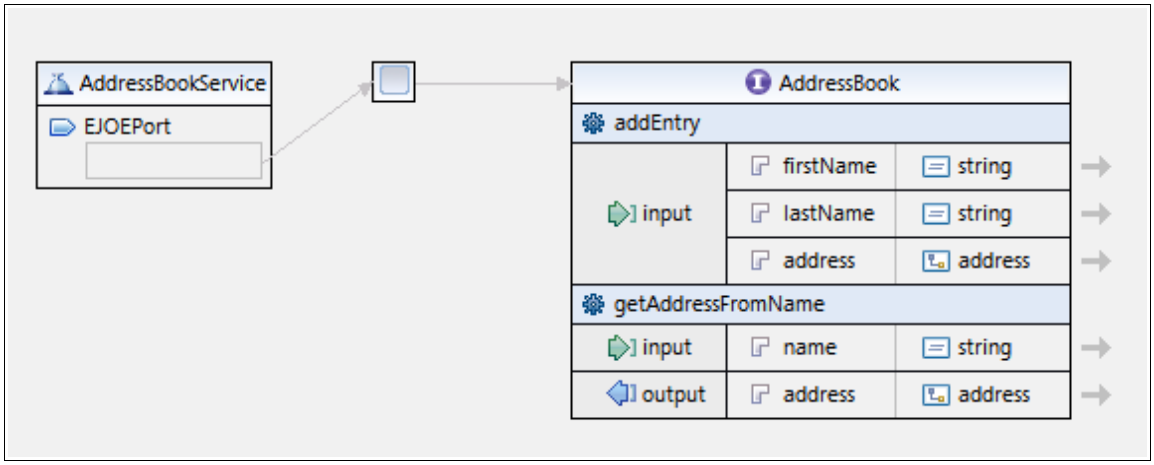
*Tabelle 9.2.*

### 9.4.2 Beispiel

```
<definitions targetNamespace="http://wsifservice.addressbook/"
       xmlns:tns="http://wsifservice.addressbook/"
       xmlns:typens="http://wsiftypes.addressbook.stub.client.ejoe/"
       xmlns:xsd="http://www.w3.org/1999/XMLSchema"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
       xmlns:ejoe="http://schemas.xmlsoap.org/wsdl/ejoe/"
       xmlns="http://schemas.xmlsoap.org/wsdl/">

       <!-- type defs -->
       <types>
              <xsd:schema
                     targetNamespace="http://wsiftypes.addressbook.stub.client.ejoe/"
                     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
                     <xsd:complexType name="phone">
                            <xsd:sequence>
                                   <xsd:element name="areaCode" type="xsd:int" />
                                   <xsd:element name="exchange" type="xsd:string" />
                                   <xsd:element name="number" type="xsd:string" />
                            </xsd:sequence>
                     </xsd:complexType>

                     <xsd:complexType name="address">
                            <xsd:sequence>
                                   <xsd:element name="streetNum" type="xsd:int" />
                                   <xsd:element name="streetName" type="xsd:string" />
                                   <xsd:element name="city" type="xsd:string" />
                                   <xsd:element name="state" type="xsd:string" />
                                   <xsd:element name="zip" type="xsd:int" />
                                   <xsd:element name="phoneNumber" type="typens:phone" />
                            </xsd:sequence>
                     </xsd:complexType>
              </xsd:schema>
       </types>

       <!-- message declns -->
       <message name="AddEntryFirstAndLastNamesRequestMessage">
              <part name="firstName" type="xsd:string" />
              <part name="lastName" type="xsd:string" />
              <part name="address" type="typens:address" />
       </message>
       <message name="GetAddressFromNameRequestMessage">
              <part name="name" type="xsd:string" />
       </message>
       <message name="GetAddressFromNameResponseMessage">
              <part name="address" type="typens:address" />
       </message>
```

```
	<!-- port type declns -->
	<portType name="AddressBook">
		<operation name="addEntry">
			<input name="AddEntryFirstAndLastNamesRequest"
				message="tns:AddEntryFirstAndLastNamesRequestMessage" />
		</operation>
		<operation name="getAddressFromName">
			<input name="GetAddressFromNameRequest"
				message="tns:GetAddressFromNameRequestMessage" />
			<output name="GetAddressFromNameResponse"
				message="tns:GetAddressFromNameResponseMessage" />
		</operation>
	</portType>
	<!-- binding declns -->
	<binding name="EJOEBinding" type="tns:AddressBook">
		<documentation>EJOEBinding</documentation>
		<ejoe:binding />
		<format:typeMapping encoding="Java" style="Java">
			<format:typeMap typeName="typens:address"

	formatType="de.netseeker.ejoe.test.client.stub.addressbook.wsiftypes.Address" />
			<format:typeMap typeName="xsd:string"
				formatType="java.lang.String" />
		</format:typeMapping>
		<operation name="addEntry">
			<ejoe:operation
				className="de.netseeker.ejoe.test.service.AddressBookImpl"
				methodName="addEntry" parameterOrder="firstName lastName address"
				invocationType="reflection" />
			<input name="AddEntryFirstAndLastNamesRequest" />
		</operation>
		<operation name="getAddressFromName">
			<ejoe:operation
				className="de.netseeker.ejoe.test.service.AddressBookImpl"
				methodName="getAddressFromName" parameterOrder="name"
				invocationType="reflection" returnPart="address" />
			<input name="GetAddressFromNameRequest" />
			<output name="GetAddressFromNameResponse" />
		</operation>
	</binding>
	<!-- service decln -->
	<service name="AddressBookService">
		<port name="EJOEPort" binding="tns:EJOEBinding">
			<ejoe:address
				adapterClass="de.netseeker.ejoe.adapter.XStreamAdapter"
				ejoeServerURL="socket://localhost:12577" useCompression="false"
				usePersistentConnection="true" timeout="500000"/>
		</port>
	</service>

</definitions>
```

*de.netseeker.ejoe.test.wsif.AddressBook.wsdl*



### Aufruf des Beispiels:

```
import org.apache.wsif.*;
import org.apache.wsif.base.*;
import de.netseeker.ejoe.ext.wsif.*;
import de.netseeker.ejoe.test.client.stub.addressbook.wsiftypes.*;
...
{
  //create a service factory
  WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
  //WSIF-Provider manuell setzen
  WSIFServiceImpl.setDynamicWSIFProvider("http://schemas.xmlsoap.org/wsdl/ejoe/",
                                         new WSIFDynamicProvider_EJOE() );
  //EJOE-Extension-Registry hinzufügen
  WSIFServiceImpl.addExtensionRegistry( new EJOEExtensionsRegistry() );

  try
  {
       WSIFService service = factory.getService( WSIFTest.class.getResource(
                                         "AddressBook.wsdl" ).toString(), null, null,
                                         "http://wsifservice.addressbook/", "AddressBook" );

       //get the port
       WSIFPort port = service.getPort();

      //create the operation
      WSIFOperation operation = port.createOperation( "addEntry",
                                                      "AddEntryFirstAndLastNamesRequest",
                                                      null );

      //create the input message associated with this operation
      WSIFMessage input = operation.createInputMessage()
      //populate the input message
      input.setObjectPart( "firstName", "John" );
      input.setObjectPart( "lastName", "Smith" );

      //create an address object to populate the input
      Address address = new Address();
      address.setStreetNum( 20 );
      address.setStreetName( "Peachtree Avenue" );
      address.setCity( "Atlanta" );
      address.setState( "GA" );
      address.setZip( 39892 );
      Phone phone = new Phone();
      phone.setAreaCode( 701 );
      phone.setExchange( "555" )
      phone.setNumber( "8721" );
      address.setPhoneNumber( phone );
      input.setObjectPart( "address", address );

       //do the invocation
       System.out.println( "Adding address for John Smith..." );
       operation.executeInputOnlyOperation( input );
  }
  catch ( WSIFException we )
  {
       System.out.println( "Got exception from WSIF, details:" );
       we.printStackTrace();
  }
}
...
```

*Aufruf der Operation „addEntry“*

```
...
{
  try
  {
       ...
       // create the operation
       WSIFOperation operation = port.createOperation( "getAddressFromName" );

       // create the input message associated with this operation
       WSIFMessage input = operation.createInputMessage();
       WSIFMessage output = operation.createOutputMessage();
       WSIFMessage fault = operation.createFaultMessage();

       // populate the input message
       input.setObjectPart( "name", "John Smith" );

       // do the invocation
       System.out.println( "Querying address for John Smith..." );

       if ( operation.executeRequestResponseOperation( input, output, fault ) )
       {
            // invocation succeeded
            // extract the address from the output message
            Address address = (Address) output.getObjectPart( "address" );

            System.out.println( "Service returned the following address:" );
            System.out.println( address.getStreetNum() + " " + address.getStreetName() + ", "
                      + address.getCity() + " " + address.getState() + " " + address.getZip()
                      + "; Phone: (" + address.getPhoneNumber().getAreaCode() + ") "
                      + address.getPhoneNumber().getExchange() + "-"
                      + address.getPhoneNumber().getNumber() );
            }
            else
            {
                System.out.println( "invocation failed, check fault message" );
            }
        }
        catch ( WSIFException we )
        {
            System.out.println( "Got exception from WSIF, details:" );
            we.printStackTrace();
        }
    }
...
```

*Aufruf der Operation „getAddressFromName“*