

Grundlagen der Programmierung von Servlets und JavaServerPages

(mit WebSphere 3.0 und Visual Age for Java 3.0, Enterprise)

Autor: Elias Volanakis

Stand: Mai 2000

Danksagung

Mein Dank gilt Prof. Dr. Wolfgang Rosenstiel, vom Arbeitsbereich Technische Informatik an der Universität Tübingen, für seine Anregungen und Korrekturvorschläge während der Erstellung dieses Handbuchs.

Tübingen, im Mai 2000

I. Inhaltsverzeichnis

I. Inhaltsverzeichnis.....	3
II. Abbildungsverzeichnis.....	7
1. IBM WebSphere Application Server kurz vorgestellt.....	8
1.1 Einführung.....	8
1.2 Was ist WebSphere?.....	8
1.3 Leistungsmerkmale der Version 3.0.....	9
1.3.1 Die Standard Edition.....	9
1.3.2 Die Advanced Edition.....	10
1.3.3 Die Enterprise Edition.....	10
1.4 Welche Plattformen werden unterstützt ?.....	10
1.5 Welche Server werden unterstützt ?.....	10
1.6 Gibt es „trial“ Versionen?.....	11
2. Servlets: Einführung.....	12
2.1 Was sind Servlets?.....	12
2.2 Das Java Servlet API.....	12
2.3 Was können HTTP Servlets ?.....	12
2.4 Nachteile von Servlets.....	13
2.5 Vorteile von Servlets.....	14
3. HTTP Servlets Programmieren.....	16
3.1 Kurzeinführung ins HTTP Protokoll.....	16
3.2 Das Request / Response Prinzip.....	16
3.3 HTTP Servlets erstellen.....	16
3.3.1 Die Methode init().....	17
3.3.2 Die Methode service(HttpServletRequest, HttpServletResponse).....	17
3.3.3 Die Methode doGet(HttpServletRequest, HttpServletResponse).....	17
3.3.4 Die Methode doPost(HttpServletRequest, HttpServletResponse).....	18
3.3.5 Die Methode destroy().....	18
3.3.6 Das Objekt HttpServletRequest.....	18
3.3.7 Das Objekt HttpServletResponse.....	19
3.4 Wie werden HTTP Servlets aufgerufen?.....	19
3.4.1 Über eine URL.....	19
3.4.2 Über ein HTML-Formular.....	19
3.5 Der Lebenszyklus eines Servlets.....	19
4. JavaServerPages (JSPs).....	21
4.1 Einführung.....	21
4.2 JSPs erstellen: Die JSP Komponenten.....	21
4.2.1 Directives.....	22
4.2.1.1 Die page Directive.....	22
4.2.1.2 Die include Directive.....	22

4.2.2 Declarations.....	22
4.2.3 Scriptlets.....	23
4.2.4 Kommentare.....	23
4.2.5 Expressions.....	23
4.2.6 Implizite Objekte.....	24
4.3 JSPs erstellen: Die JSP Tags.....	24
4.3.1 Tags zur Bearbeitung von Beans.....	24
4.3.2. tsx:repeat.....	25
4.4 JSPs aufrufen.....	25
4.4.1 JSP über ihre URL aufrufen.....	25
4.4.2 JSP über ein HTML Formular aufrufen.....	25
4.4.3 JSP aus einer JSP aufrufen.....	26
5. Datenbankbindung mit Servlets.....	27
5.1 JDBC.....	27
5.2 JDBC-Treiber.....	27
5.3 Das Package java.sql.....	28
5.3.1 Die Klasse DriverManager.....	29
5.3.2 Das Interface Connection.....	29
5.3.3 Das Interface Statement.....	30
5.3.4 Das Interface PreparedStatement.....	30
5.3.5 Das Interface ResultSet.....	31
5.4 WebSphere Datenquellen.....	31
5.4.1 Was ist Verbindungs-Pooling?.....	32
5.4.2 Erzeugen einer Datenquelle.....	32
5.4.3 Benutzen einer Datenquelle.....	33
5.6 Data Access Beans.....	34
6. Zustandserhaltung.....	36
6.1 Das HttpSession Objekt.....	36
Anhang A: Quellen und Links.....	38
A.1 Quellen.....	38
A.2 Links.....	38
A.3 Weiterführende Literatur.....	38
Anhang B: Quellennachweis der Bilder.....	40
Anhang C: Attribute der JSP Directive „page“.....	41
Anhang D: Zuordnung SQL-Datentypen zu Java-Datentypen.....	42
Anhang E: Programmcode Beispiele.....	43
E.1 Codebeispiele für Servlets.....	43
E.1.1 Das HelloWorld Servlet.....	43
E.1.2 Das Sample1Servlet.....	44
E.1.3 HTTP POST verarbeiten (Formular auswerten).....	45
E.2 Codebeispiele für JSPs.....	48

E.2.1 Ein sehr einfaches JSP.....	48
E.2.2 Formular auslesen durch eine JSP.....	48
E.2.3 Zusammenarbeit von Servlet und JSP mit Bean.....	49
E.3 JDBC Beispiele.....	52
E.3.1 Programm mit JDBC Aufruf.....	52
E.3.2 Servlet mit JDBC Aufruf.....	53
E.3.2.1 Konfigurationsvariablen in externe Dateien verlagern.....	55
E.3.3 Servlet mit Datenquellen Aufruf.....	56
E.4 Servlet mit HttpSession Objekt.....	58
Übungen – Allgemeine Vorbereitungen.....	61
1. Übung: Seitenzugriffszähler.....	62
1.1. Aufgabenstellung.....	62
1.2 Hinweise.....	62
1.3 Ausführen.....	62
1.4 Optionale Aufgaben.....	63
1.4.1 Cache Abschalten.....	63
1.4.2 Start-Zeit anzeigen.....	63
2. Übung: Formular auslesen.....	64
2.1 Vorbereitung.....	64
2.2 Aufgabenstellung.....	64
2.3 Hinweise.....	64
2.4 Ausführen - Testen.....	65
2.5 Optionale Aufgaben.....	65
3. Übung: JSP-Counter.....	66
3.1 Aufgabenstellung.....	66
3.2 Hinweise.....	66
4. Übung: Datenanzeige mit JDBC.....	67
4.1 Vorbereitung.....	67
4.2 Aufgabenstellung.....	67
4.3 Die Datenbank.....	68
5. Übung: Servlet mit JDBC Datenbankzugriff.....	69
5.1 Aufgabenstellung.....	69
5.2 Hinweise.....	69
5.3 Optionale Aufgabe.....	69
6. Übung: HttpSession.....	70
6.1 Aufgabenstellung.....	70
6.2 Hinweise.....	70
L. Lösungen.....	71
Lösung der Übung 1.....	71
Schritt 1: Erstellen des Servlet Objektes.....	71
Schritt 2: Instanzvariablen in die Klasse schreiben.....	72
Schritt 3: Die Methode doGet(...) erzeugen.....	72
Lösung der Übung 2.....	74
Code des HTML-Formulars.....	74
Schritt 1: Erstellung der Klasse.....	75
Schritt 2: Methode doPost(...) Programmieren.....	75
Lösung der Übung 3.....	77
Lösung der Übung 4.....	78
Lösung der Übung 5.....	79
Lösung der Übung 6.....	81

II. Abbildungsverzeichnis

Abbildung 1: WebSphere Advanced Edition - Ausführungsumgebung.....	9
Abbildung 2: Modell – das Ausführen von Servlets.....	14
Abbildung 3: Das Request / Response Prinzip (Grobdarstellung).16	
Abbildung 4: Ein HTTP Servlet, das GET und POST Requests verarbeitet.....	17
Abbildung 5: Der Lebenszyklus eines Servlets.....	20
Abbildung 6: Erstellung eines Servlets aus einer JSP (beim 1. Aufruf).....	21
Abbildung 7: Zusammenarbeit von Servlet und JSP.....	21
Abbildung 8: Übersicht JDBC.....	28
Abbildung 9: Oracle JDBC-Treiber in WebSphere anmelden.....	33
Abbildung 10: Datenquelle erstellen.....	33
Abbildung 11: HTML Formular.....	47
Abbildung 12: Antwort des Servlets FormServlet.....	47
Abbildung 13: Antwort von simple.jsp.....	48
Abbildung 14: Antwort von beanDemo.jsp.....	51
Abbildung 15: Auslagerung von Zeichenketten.....	56
Abbildung 16: Ausgabe von CallCounterServlet.....	58

1. IBM WebSphere Application Server kurz vorgestellt

1.1 Einführung

Dynamisch erzeugte Webseiten gewinnen immer mehr an Bedeutung. Sei es um die Ausgabe einer Datenbankabfrage anzuzeigen, eine Bestellung zu tätigen oder eine bestehende Anwendung an das Internet anzubinden.

Einige Technologien zur dynamischen Erstellung von Webseiten sind:

- CGI Skripte
- Skript-Sprachen (z.B. Net.Data von IBM, ActiveServerPages (ASP) von Microsoft, ColdFusion von Allaire).
- Server Plugins für bestimmte Webserver (z.B.: NSAPI von Netscape und ISAPI von Microsoft)
- **Servlets** und **JavaServerPages** (JSPs).

Servlets sind spezielle, serverseitige Java Programme. „Serverseitig“ bedeutet, daß diese auf einen Server ausgeführt werden und nicht auf dem Client des Benutzers (meist ein Webbrowser). Hauptaufgabe ist das Erzeugen von Web-Inhalten und die Einbindung von externen Ressourcen wie Datenbanken.

JavaServerPages sind HTML Seiten mit eingebetteten Java Programmcode Stücken. Sie können die Anzeige der generierten Inhalte übernehmen, um so Präsentation und Programmlogik voneinander zu trennen.

Mehr Informationen zu Servlets und JSPs finden Sie in den folgenden Kapiteln.

1.2 Was ist WebSphere?

WebSphere ist ein Applikationsserver der Firma IBM, der die Funktionalität von Webservern erweitert. Er stellt eine Plattform für die Entwicklung von portablen, auf Java basierenden, Web Anwendungen zur Verfügung. Seine Hauptaufgabe ist das Ausführen von Servlets und JavaServerPages (JSPs).

Zu diesem Zweck bildet er eine logische Einheit mit einem Webserver. Wenn der Benutzer Servlets oder JSPs anfordert, leitet der Webserver diese Anforderung an den Applikationsserver weiter. WebSphere bearbeitet die Anforderung und sendet das Ergebnis zurück an den Webserver, der dieses zum Benutzer transportiert.

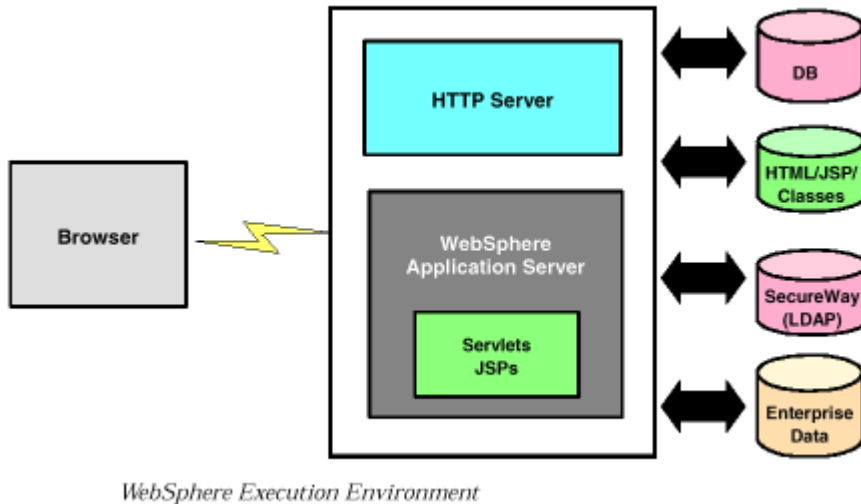


Abbildung 1: WebSphere Advanced Edition - Ausführungsumgebung

Der WebSphere Application Server (WAS) ist besonders für folgende Zielgruppen interessant:

- Unternehmen, die existierende Applikationen in das Web integrieren wollen.
- Unternehmen, die verteilte, unternehmensweite, Applikationen erstellen wollen.
- Unternehmen, welche Ihre Präsenz im Web, durch Benutzung der neuesten Technologien, attraktiver gestalten wollen.

1.3 Leistungsmerkmale der Version 3.0

Es gibt den WebSphere Application Server v3.0 in drei Ausführungen (Editions): Standard, Advanced und Enterprise. Im folgenden werden die wichtigsten Leistungsmerkmale der verschiedenen Ausführungen vorgestellt.

1.3.1 Die Standard Edition

Dies ist die kostengünstigste Ausführung und bietet u.a. folgende Leistungsmerkmale:

- Unterstützung des Java Servlet API v. 2.1
- Unterstützung der JavaServerPages Spezifikation in den Versionen 0.91 und 1.0
- Datenbankanbindung über JDBC für die Datenbanken IBM DB2 Universal Database, Oracle und Microsoft SQLServer. Zur Verbesserung der Leistung werden die Datenbankverbindungen in sog. „Pools“ verwaltet.
- XML Tools, zum lesen/parsen und transformieren von XML Daten.
- Administrationskonsole, um benutzerfreundliche Administration zu ermöglichen.
- Protokollierungs- und Analyse-Tools, zur Inhalts- und Nutzungsanalyse der Web-Site.

- Maschinelle Übersetzung; sie ermöglicht dem Benutzer, Webseiten, die ursprünglich in einer anderen Sprache erstellt wurden, in einer von Ihm ausgewählten Standardsprache anzuzeigen.

1.3.2 Die Advanced Edition

Diese Ausführung erweitert die „Standard“ Edition u.a. um folgende Aspekte:

- Unterstützung von Enterprise Java, nach der Enterprise JavaBeans (EJB¹) Spezifikation v. 1.0.
- Skalierbarkeit und Auslastungsverwaltung, durch Verteilung der Last auf mehrere Maschinen.
- Verbesserte Steuerungs- und Verwaltungselemente für die Sicherheit, u.a. auch Authentifizierung über den LDAP-Service (Lightweight Directory Access Protocol).
- CORBA Unterstützung inkl. Unterstützung des Internet Inter-ORB Protokolls (IIOP) und der Remote Method Invocation (RMI) über IIOP.

1.3.3 Die Enterprise Edition

Schwerpunkt der „Enterprise“ Ausführung ist die Bereitstellung einer robusten Lösung für die Integration von Webapplikationen in Unternehmensumgebungen. Es wird besonderer Wert auf Skalierbarkeit, Performanz, Verfügbarkeit und die Einbindung bestehender Ressourcen gelegt. Es wird Transaktionsverarbeitung, unter Benutzung der Produkte CICS oder IBM Encina, unterstützt.

1.4 Welche Plattformen werden unterstützt ?

Der WebSphere Application Server unterstützt die Plattformen AIX, Linux, NetWare, OS/2, OS/390, OS/400, Solaris und Windows NT.

Welche Ausführungen in welcher Version für das jeweilige Betriebssystem verfügbar sind, können Sie auf den Webseiten der Firma IBM, unter folgender URL erfahren:

<http://www-4.ibm.com/software/webservers/appserv/>

1.5 Welche Server werden unterstützt ?

Der WebSphere Application Server arbeitet mit folgenden Servern zusammen:

- Apache Server, Version 1.3.6 für IBM AIX, Sun Solaris und Microsoft Windows NT
- Netscape Enterprise Server, Version 3.51 und Version 3.60 für AIX, Solaris und Windows NT
- Microsoft Information Server Version 4.0 für Windows NT
- Lotus Domino™ Application Server Release 5 für Windows NT, Sun Solaris und AIX

¹ EJB: Enterprise JavaBeans sind serverseitige Komponenten, die in einem Container zum Einsatz kommen. Es gibt Entity-Beans und Session-Beans. Die Entity Beans sind persistente Geschäftsobjekte, die in einer Datenbank gespeichert werden. Session Beans sind Geschäftsobjekte, die kurze UseCases oder Workflows repräsentieren.

- Domino Go Webserver Release 4.6.2.5 und 4.6.2.6 für AIX, Solaris und Windows NT
- IBM HTTP Server V1.3.6 für AIX, Solaris und Windows NT

1.6 Gibt es „trial“ Versionen?

Es ist möglich, kostenlose, zeitlich beschränkt lauffähige, trial Versionen der Ausführungen „Standard“ und „Advanced“ des WebSphere Application Servers zu beziehen. Mehr Informationen dazu, auch über die einzelnen Ausführungen und deren Systemanforderungen, finden Sie unter der folgenden URL:

<http://www-4.ibm.com/software/webservers/appserv/>.

2. Servlets: Einführung

2.1 Was sind Servlets?

Servlets sind **spezielle serverseitige Java Programme**, deren Zweck die Erweiterung der Funktionalität eines Servers ist.

"Serverseitig" bedeutet, daß die Programme, in einer Java Virtual Machine (JVM), auf einem Server laufen und nicht auf dem Client, der sie aufgerufen hat. Der Client (Webbrowser) braucht Java nicht zu unterstützen.

Geladen und ausgeführt werden Servlets also von einem Applikationsserver (z.B. IBM WebSphere) oder von einem Webserver welcher Servlets unterstützt. So wie Java-Applets auf einem Webbrowser ausgeführt werden, um dessen Funktionalität zu erweitern, erweitern Servlets die Funktionalität eines Servers.

Servlets sind spezielle Java Programme, weil sie durch das **Java Servlet API** definiert werden. Dieses stellt eine Schnittstelle zwischen dem Server und dem Servlet dar. Servlets sind deshalb prinzipiell nicht abhängig von einem bestimmten Server. Sie können auf jedem Server laufen, der dieses API unterstützt.

2.2 Das Java Servlet API

Um Servlets zu erstellen, werden entsprechende Klassen aus dem Java Servlet API erweitert. Das API besteht aus den folgenden zwei Paketen (Packages):

- **Package javax.servlet:** Es enthält Klassen für die Unterstützung von generischen, protokollunabhängigen Servlets. Solche Servlets können für jedes Protokoll benutzt werden, das nach dem Anforderung/Antwort (Request/Response) Prinzip arbeitet. Beispiele für solche Protokolle sind FTP², SMTP³ und POP⁴. Um eine generisches Servlet zu erstellen wird die Klasse `javax.servlet.GenericServlet` erweitert.
- **Package javax.servlet.http:** Es enthält Klassen für die Unterstützung von Servlets, welche das HTTP Protokoll benutzen. Diese spezialisierten Servlets können HTTP-Client Anforderungen empfangen und eine Antwort zurückgeben. Sie werden **HTTP Servlets** genannt. Ein HTTP Servlet wird durch Erweitern der Klasse `javax.servlet.http.HttpServlet` erstellt.

Der Applikationsserver WebSphere v3.0, unterstützt das Java Servlet API in der Version 2.1.

2.3 Was können HTTP Servlets ?

Die Hauptfunktion der HTTP Servlets ist das Generieren von Web-Inhalten, die Kommunikation mit Enterprise JavaBeans und die Einbindung von externen Ressourcen.

² FTP: File Transfer Protocol. Wird zum Transport von Dateien über TCP/IP Netzwerke benutzt.

³ SMTP: Simple Mail Transfer Protocol. Wird zum Transport von eMails benutzt.

⁴ POP: Post Office Protocol. Wird von Rechner ohne permanente Netzverbindung benutzt, um eMails von POP-Servern abzurufen.

Einige Beispielanwendungen für solche Servlets sind:

- Eine ganze HTML Seite, aufgrund einer Client-Anforderung, dynamisch erzeugen und zurückliefern.
- Ein Teil einer HTML Seite (ein HTML-Fragment) erzeugen, welches dann in eine bereits bestehende Seite eingefügt werden kann.
- Eine Benutzereingabe verarbeiten und die Ergebnisse an andere Servlets oder JavaServerPages schicken.
- Mit diversen Ressourcen kommunizieren und interagieren. Die können z.B. Datenbanken, weitere Servlets, Applets oder andere Java Applikationen sein.
- Filtern von Daten nach MIME-Typ und anschließende Bearbeitung der Daten, z.B. dynamische Grafikumwandlungen oder Veränderung von Textdateien.
- Verbindungen mit mehreren Klienten verwalten und Daten an diese verteilen. Zum Beispiel kann ein Chat-Server mit Servlets realisiert werden.

Zu den Unternehmen, die Servlets verwenden gehört das online Auktionshaus eBay (<http://www.ebay.de>), welches mit dieser Technologie die Web-Oberfläche für sein Versteigerungssystem implementiert hat.

2.4 Nachteile von Servlets

Höhere Anforderungen: Um mit Servlets zu arbeiten, ist ein Webserver / Applikationsserver notwendig, der das Java Servlet API unterstützt.

Performanz: Die Performanz von Servlets läßt sich nicht genau einstufen und genaue Zahlen stehen mir nicht zur Verfügung. Es existieren jedoch zwei offensichtliche Faktoren, die sich nachteilig auf diese auswirken:

1. Servlets werden in Java programmiert. Wie Ihnen bekannt ist, bestehen Java Programme aus Bytecode, welcher von einer JavaVirtualMachine (JVM) interpretiert wird. Die Interpretation des Bytecodes verögert die Abarbeitung.

Just-in-time Compiler können dies teilweise verbessern. Sie konvertieren Java Bytecode in Maschinencode, mit dem Ziel die Ausführungsgeschwindigkeit zu steigern. Ihr Einsatz macht aber nicht immer Sinn. Eine JVM mit JIT-Compilierung bringt dann Vorteile, wenn kleine Schleifen oft durchlaufen werden, so daß die zusätzliche Zeit für die JIT-Compilierung geringer ist, als der Durchlauf der Schleife ohne JIT. Wenn keine Schleifen vorhanden sind, wird der JIT Compiler u.U. Code übersetzen, der nur einmal ausgeführt werden wird, und der zusätzliche Aufwand wird sich u.U. nicht lohnen. Es kann also von Vorteil sein, die Servlets ohne und mit JIT auszuführen, wenn diese Möglichkeit besteht, um herauszufinden, welche Alternative besser ist.

2. Servlets werden in der Regel nur einmal instanziiert. Es existiert also genau ein Objekt von jedem Servlet. Mehrere Threads (eins pro Client-Anfrage) des Servers greifen auf das gleiche Objekt zu, unter Umständen gleichzeitig. Falls also während der Bearbeitung der Anfrage Instanzvariablen verändert werden, ist es notwendig **synchronized** Ausdrücke zu benutzen, um die Konsistenz der Instanzvariablen zu schützen. Durch den Ausdruck „synchronized“ wird garantiert, daß jeweils nur ein Thread Zugriff auf das so geschützte Objekt bzw. die so geschützte Methode hat. Die anderen Threads warten solange,

was die Antwortzeit verlängert.

Allerdings kann man i.d.R. ohne Instanzvariablen auskommen und so dieses Problem umgehen. Variablen, die nur innerhalb der Methoden des Servlets existieren, sind von diesem Problem nicht betroffen.

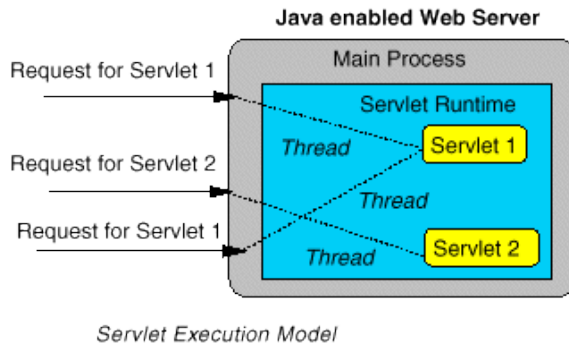


Abbildung 2: Modell – das Ausführen von Servlets

Wenn man die Performanz von Servlets mit anderen Alternativen vergleicht, sind die Ergebnisse unterschiedlich. Im Vergleich zu CGI Skriptsprachen wie Perl, sind Servlets deutlich schneller. Anders ist es mit Server Plugins⁵, die auf einen bestimmten Webserver zugeschnitten sind und in nativem Code programmiert sind. Sie haben eine sehr hohe Performanz, sind aber an den jeweiligen Server gebunden und fehlerträchtiger in der Programmierung.

2.5 Vorteile von Servlets

Die Verwendung von Servlets für das Erstellen von dynamischen Web-Inhalten bietet jedoch auch eine ganze Reihe von Vorteilen:

Java basierend: Wie bereits erwähnt, werden Servlets in Java geschrieben. Man profitiert also von den positiven Eigenschaften dieser Programmiersprache, wie z.B. der **Objektorientierung** und der **Modularisierung**, die **Einfachheit** der Sprache und den **einfachen Datenbankzugriffen** (JDBC). Dadurch kann Programmcode leichter wiederverwendet werden und die benötigte Entwicklungszeit verkürzt sich.

Portabel: Die Portabilität zwischen verschiedenen Betriebssystemen wird durch die Verwendung der Programmiersprache Java gewährleistet. Durch das Servlet API wird außerdem eine serverunabhängige Schnittstelle definiert, was die Servlets auch portabel zwischen verschiedenen Servern macht.

Leistungsstark: Servlets können das ganze Leistungsspektrum von Java ausnutzen, wie z.B. die Unterstützung von Netzwerkprogrammierung, Threads, Bildmanipulation, Datenkompression, Datenbankzugriffe, Internationalisierung, RMI, CORBA und vieles mehr.

Performant: Folgende Faktoren wirken sich positiv auf die Performanz aus:

- a) Jedes Servlet wird i.d.R. nur einmal von Webserver geladen und instanziiert. Es befindet sich dann im Speicher und kann sofort auf Anforderungen (Requests) des Klienten reagieren.

⁵z.B. NSAPI von Netscape und ISAPI von Microsoft

- b) Die Servlets laufen in mehreren Threads, aber im Prozeß des Applikationsservers. Es muß also nicht für jedes Servlet ein Betriebssystem-Prozeß erzeugt werden.
- c) Der WebSphere Applikationsserver bietet Verbindungs-Pools für die Datenbankverbindungen. Es muß also nicht für jeden Zugriff auf eine Datenbank auch eine Verbindung erzeugt werden, was Zeit einspart.

Trennung von Programmlogik und Daten-Darstellung: Durch die Verwendung von Servlets für die Programmlogik und JavaServerPages für die Darstellung wird ein hohe Flexibilität erreicht. Für größere Projekte kann auch das sogenannte MVC (**Modell-View-Controller**) Design implementiert werden. JSPs und HTML-Seiten implementieren die Darstellung, Java Beans das Modell und Servlets den Programmablauf.

Außerdem ist dadurch eine gute Aufgabenteilung möglich, d.h. die Web-basierte Benutzerschnittstelle kann von HTML-Designern erstellt werden, die auch mit JSP arbeiten, während Programmierer die Entwicklung der Applikation übernehmen. So kann eine optimale Kompetenzausnutzung erreicht werden.

Sessions: Dieses Objekt des Servlet APIs wird eindeutig einem Benutzer (über ein Cookie) zugeordnet. In eine Session können Daten von verschiedenen Servlets geschrieben und gelesen werden. Auf diese einfache Art und Weise können komplexe Vorgänge (z.B. eine Bestellung) in mehrere Servlets aufgeteilt werden.

3. HTTP Servlets⁶ Programmieren

3.1 Kurzeinführung ins HTTP Protokoll

Das HTTP Protokoll ist ein einfaches, **verbindungsloses** Protokoll. Verbindungslos heißt, daß zwischen Client (Webbrowser) und Server keine permanente Verbindung besteht, sondern nur eine temporäre.

Wenn der Client etwas vom Server will, schickt er einen sogenannten **Request** (Anforderung) an den Server. Ein Request enthält einen HTTP Befehl, welcher angibt, was der Server tun soll. Nachdem der Server den Befehl verarbeitet hat, antwortet er mit einer **Response**.

Die am häufigsten benutzten HTTP Befehle sind die Kommandos **GET** und **POST**. Etwas vereinfacht formuliert dient das GET Kommando dazu, Daten von dem Server zu bekommen, und das POST Kommando dazu, Daten an der Server zu senden.

3.2 Das Request / Response Prinzip

Die HTTP Servlets greifen dieses Request / Response Prinzip auf. Im Java Servlet API ist ein Interface für die Bearbeitung von solchen Anforderungen und das Zurücksenden von Antworten definiert.

Auf dem folgenden Bild sind die Einzelschritte einer solchen Interaktion dargestellt:

1. Der Client (Webbrowser) schickt einen Request an den Webserver.
2. Der Webserver gibt diesen an das Servlet weiter.
3. Das Servlet verarbeitet die Request, bindet u.U. externe Ressourcen (z.B. eine Datenbank) ein und generiert eine Response, die an den Webserver zurückgegeben wird.
4. Der Webserver sendet die Response an den Client.

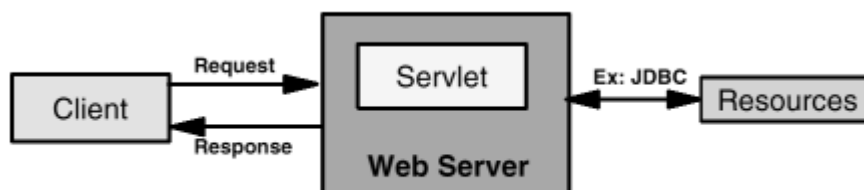


Figure 31. High-level client to servlet process flow

Abbildung 3: Das Request / Response Prinzip (Grobdarstellung)

3.3 HTTP Servlets erstellen

Um ein HTTP Servlet zu erstellen, wird die Klasse `HttpServlet`, aus dem Package `javax.servlet.http` erweitert. Wie Ihnen bereits bekannt ist, erhält ein solches Servlet Requests von einem Webserver und schickt Responses zurück.

⁶ HTTP Servlets: Es handelt sich um spezielle Servlets, die HTTP-Client Anforderungen empfangen können und eine generierte Antwort zurückgeben.

In der Regel ist es mindestens notwendig, die Methode `doGet(...)` bzw. `doPost(...)` zu überschreiben.

Im folgenden werden die wichtigsten Methoden und Objekte eines HTTP Servlets kurz vorgestellt. Eine ausführliche Beschreibung dieser Objekte finden Sie in der Dokumentation des Java Service API, unter der URL:
<http://www.javasoft.com/products/servlet/download.html#specs>

3.3.1 Die Methode `init()`

Diese Methode wird einmal, während der Instanziierung des Servlets aufgerufen.⁷ In der Regel ist es nicht notwendig, die standardmäßig definierte `init()` Methode zu überschreiben.

Sie können jedoch eine benutzerdefinierte `init()` Methode erstellen, wenn Sie bestimmte Ressourcen dem Servlet zur Verfügung stellen wollen, die während der gesamten Lebensdauer des Servlets unverändert bleiben. Dadurch können Sie die Leistung des Servlets verbessern.

Ein Beispiel für eine solche Ressource ist die Initialisierung einer Datenbankverbindung. Ein anderes Beispiel ist das Einlesen einer Datei, die in jede Antwort eingefügt werden soll. Sie braucht dann nicht bei jeder Antwort eingelesen zu werden, sondern ist schon verfügbar.

3.3.2 Die Methode `service(HttpServletRequest, HttpServletResponse)`

Diese Methode wird bei jeder Request (Anforderung) eines Clients aufgerufen.

In der Klasse `HttpServlet` ist diese Methode bereits vorhanden. Sie ruft eine `doXXX` Methode auf, welche der HTTP-Anforderung entspricht. Die wichtigsten Methoden dieses Typs sind `doGet(...)` und `doPost(...)`, für die HTTP Befehle GET und POST. Es existieren jedoch noch weitere Methoden dieser Art für die anderen HTTP Befehle. Sie können diese in der Servlet API Dokumentation nachschlagen (URL siehe oben).

Es ist nicht notwendig, diese Methode zu überschreiben. Überladen Sie lediglich die entsprechenden `doXXX` Methoden.

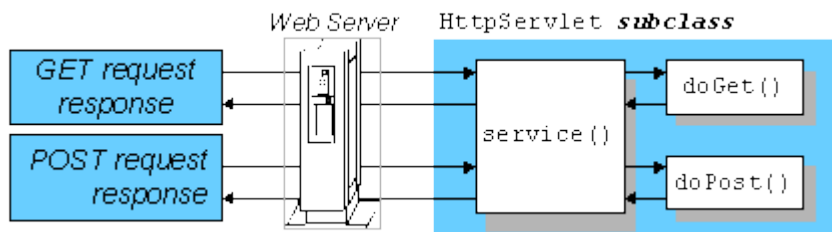


Abbildung 4: Ein HTTP Servlet, das GET und POST Requests verarbeitet

3.3.3 Die Methode `doGet(HttpServletRequest, HttpServletResponse)`

Diese Methode verarbeitet eine HTTP GET Request, die sie von der `service(...)` Methode erhält.

⁷ Diese Methode gibt es erst seit der Version 2.1 des Java Servlet APIs. Wenn Sie Servlets für ältere API Versionen erstellen, müssen sie statt dessen die Methode `init (ServletConfig config)` überschreiben und dann als erstes `super.init (config)` aufrufen.

HTTP GET erlaubt dem Clienten, Informationen vom Server zu erhalten. Der Client hat die Möglichkeit, eine Zeichenkette (query string) an die URL anzufügen, um dem Server Informationen zu übermitteln.

Wenn Sie diese Methode überschreiben, sind in der Regel folgende Arbeiten zu erledigen. Sie müssen u.U. Daten aus dem Request auslesen, die Header-Informationen für die Response setzen, den PrintWriter aus der Response holen und dort die Antwort hineinschreiben. Am besten zu verstehen ist dies durch den Beispielcode im Abschnitt E.1.

3.3.4 Die Methode doPost(HttpServletRequest, HttpServletResponse)

Diese Methode verarbeitet einen HTTP POST Request, den sie von der service (...) Methode erhält.

Über HTTP POST hat der Client die Möglichkeit, Daten von unbeschränkter Länge einmal, an den Server zu senden. Über HTTP POST kann, z.B., der Inhalt von Web-Formularen an den Server übermittelt werden.

Wenn Sie diese Methode überschreiben, sind in der Regel ähnliche Arbeiten wie bei der doGet(...) Methode zu erledigen: Daten aus dem Request auslesen, Header-Informationen setzen usw. Am besten zu verstehen ist dies durch den Beispielcode im Abschnitt E.1.

3.3.5 Die Methode destroy()

Diese Methode wird beim Entladen (herausnehmen) des Servlets aus dem Speicher, aufgerufen. In der Regel ist es nicht notwendig, die standardmäßig definierte destroy() Methode zu überschreiben.

Sie können jedoch eine benutzerdefinierte destroy() Methode erstellen, wenn Sie bestimmte Aktionen während der Beendigung des Servlets ausführen wollen. Dies kann zum Beispiel der Fall sein, wenn das Servlet während seiner Ausführung Daten sammelt, die vor der Beendigung gesichert werden sollen. So können auch z.B. Datenbankverbindungen und Ressourcen freigegeben werden, die während der Initialisierung erstellt wurden.

Während der Entladung des Servlets, wird die destroy() Methode erst dann aufgerufen, wenn alle Methoden des Typs service(...) beendet wurden oder ein Zeitlimit überschritten wurde. Werden in der service(...) Methode Threads erzeugt, könnte es sein, daß deren Ausführung noch nicht beendet ist, wenn destroy() aufgerufen wird. Um Probleme zu vermeiden, sollten Sie sicherstellen, das solche Threads beendet sind, wenn destroy() aufgerufen wird.

3.3.6 Das Objekt HttpServletRequest

Dieses Objekt enthält Informationen über den HTTP Request des Clients, wie z.B. Header-Informationen, Daten von Cookies, das zugehörige Session Objekt, die aufgerufene URL und anderes.

Diese Daten können über eine Reihe von Methoden ausgelesen werden. Mehr Informationen dazu finden sie in der Servlet API Dokumentation.

3.3.7 Das Objekt HttpServletResponse

Dieses Objekt definiert eine HTTP Servlet Response (Antwort), die an den Webserver weitergeleitet wird. Der Webserver schickt die Response dann an den Client zurück.

Die wichtigsten Methoden dieses Objektes sind **setContentLength(int)**, welche den Inhalt der Antwort spezifiziert (z.B. "text/html" oder "image/gif"), und **getWriter()**, welche ein PrintWriter Objekt zurückgibt, in das die Daten für die Antwort geschrieben werden.

Über eine Reihe von Methoden des HttpServletResponse Objektes können außerdem verschiedene Eigenschaften der Antwort verändert werden. So kann über die Methoden **setHeader(String, String)** und **setDateHeader(String, int)** das Zwischenspeichern (Caching) der Seite unterbunden werden (siehe auch Abschnitt E.1).

Mehr Informationen über die verschiedenen Methoden des Objektes können Sie in der Servlet API Dokumentation finden.

3.4 Wie werden HTTP Servlets aufgerufen?

Servlets können auf folgende Arten aufgerufen werden:

3.4.1 Über eine URL

Wenn Sie eine Web-Anwendung konfigurieren, werden Sie den Klassennamen für die einzelnen Servlets angeben, die eine Komponente der Anwendung sind, und diese mindestens einem Servlet-Web-Pfad (URL) zuordnen. In examples wurde beispielsweise dem ServletEngineConfigDumper (Code-Name) die Servlet-URL / showCfg zugeordnet. Wenn Sie die URL, die einem Servlet zugeordnet ist, nicht kennen, lassen Sie sich die Servlet-Merkmale mit Hilfe der WebSphere Administrationskonsole anzeigen.

Sie können dann ein Servlet über dessen Servlet-URL aufrufen:

`http://your.server.name/application_Web_path/servlet_Web_path`

Beispiel: <http://localhost/webapp/examples/showCfg>

3.4.2 Über ein HTML-Formular

Sie können eine Servlet in einem <FORM> Tag aufrufen. Ein HTML-Formular ermöglicht es dem Benutzer, Daten in einer Webseite (in einem Browser) einzugeben und diese Daten anschließend einem Servlet zu übergeben.

Um ein Servlet aufzurufen gehen Sie wie folgt vor:

```
<FORM METHOD="POST|GET" ACTION="application_URI/Servlet_URL">
<!-- Tags für Texteingabe, Knöpfe u.a. Elemente einfügen -->
</FORM>
```

Abhängig davon ob METHOD den Wert POST oder GET hat, muß das Servlet entweder die doPost(...) Methode oder die doGet(...) Methode implementieren.

3.5 Der Lebenszyklus eines Servlets

Folgendes Bild stellt den Lebenszyklus eines Servlets dar. Nach der Instanziierung des Servlets wird zuerst die init() Methode aufgerufen. Für jede eingehende Client-Anforderung (Request) wird die service(...) Methode aufgerufen. Beim

Entfernen des Servlets aus dem Arbeitsspeicher wird die `destroy()` Methode aufgerufen.

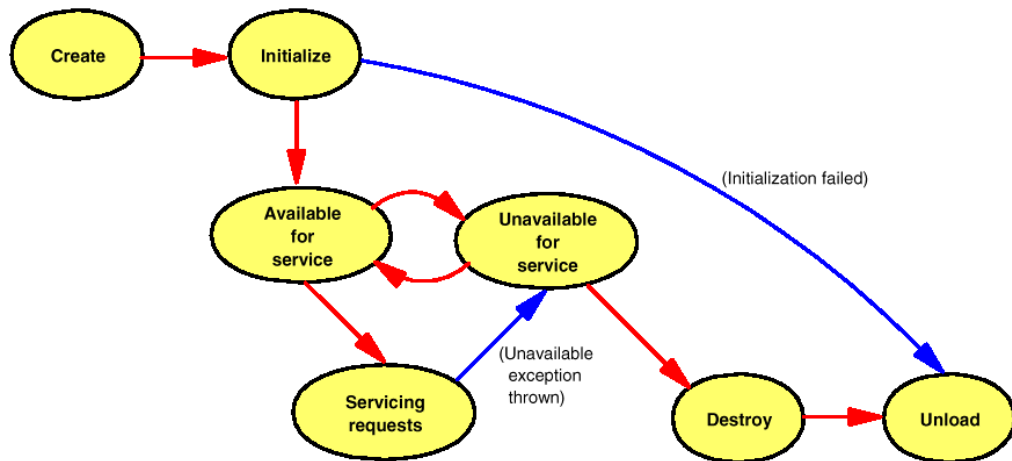


Abbildung 5: Der Lebenszyklus eines Servlets

4. JavaServerPages (JSPs)

4.1 Einführung

JavaServerPages sind "normale" HTML Seiten, welche zusätzlich eingebettete Java Befehle enthalten. Wenn eine JSP zum ersten Mal aufgerufen wird, erstellt der Server automatisch ein Servlet dieser. Der HTML Code aus der JSP und die Java Befehle werden auf diese Weise zusammen ausgeführt. Für den Benutzer ist dies absolut transparent.

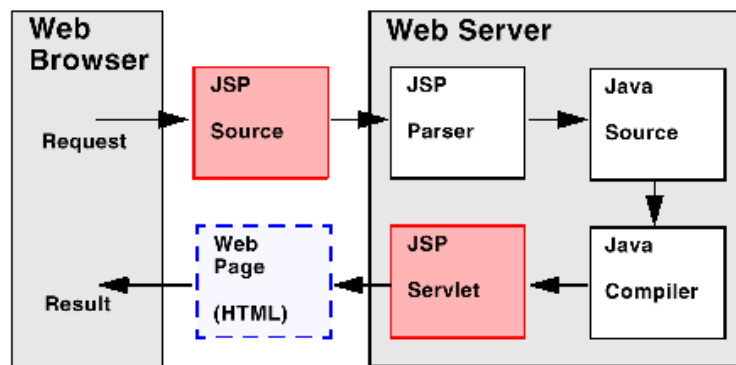


Abbildung 6: Erstellung eines Servlets aus einer JSP (beim 1. Aufruf)

Durch JSPs wird es möglich, die Programmlogik von der Darstellung der Daten zu trennen. Die Steuerung der Anwendung bzw. die Datenverarbeitung erfolgt in den Servlets. Diese schicken, über Beans, die Daten an JSPs, welche dann die Anzeige der Daten übernehmen.

Der Applikationsserver WebSphere unterstützt sowohl die neue Version 1.0 der JSP API, als auch die ältere Version 0.91.

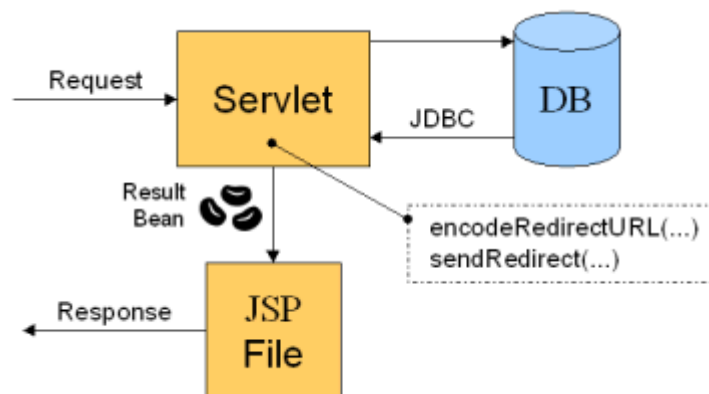


Abbildung 7: Zusammenarbeit von Servlet und JSP

4.2 JSPs erstellen: Die JSP Komponenten

Eine JSP Datei muß die Endung `.jsp` haben. Zusätzlich zu HTML Tags können JSP Komponenten und JSP Tags benutzt werden. Im folgenden werden die JSP Komponenten der JSP API, in der Version 1.0, vorgestellt.

4.2.1 Directives

Mit JSP Direktiven ist es möglich, globale Parameter der JSP einzustellen. Diese Einstellungen sind unabhängig vom Inhalt des jeweiligen Request (Anforderung) an die JSP. Sie werden zum Beispiel verwendet, um zu spezifizieren, welche Skriptsprache benutzt wird, welche Java Klassen importiert werden sollen und anderes.

Der Syntax einer Direktive lautet:

```
<%@ directive directive_attr_name = "value" %>
```

Die wichtigsten Direktiven sind **page** und **include**.

4.2.1.1 Die page Directive

Die **page** Direktive spezifiziert seitenabhängige Attribute einer JSP Seite.

Beispiel:

```
<%@ page language="java" %>
<%@ page import = "java.util.*" %>
<%@ page import = "java.sql.*" %>
<%@ page contentType = "text/html" %>
```

Die verwendeten JSP **page** Direktiven müssen **vor allen anderen** JSP Elementen vorkommen. Dies ist wegen der Arbeitsweise des JSP-Parsers notwendig, welcher Servlets aus den JSP Dateien erzeugt.

Folgende sind die am häufigsten verwendeten Werte für den Parameter `directive_attr_name`. Eine vollständige (englische) Auflistung finden Sie in Anhang C.

- **language** – Spezifiziert, welche Skriptsprache in dieser JSP benutzt wird. *Das JSP API sieht nur den Wert "java" vor.* Der WebSphere Applikationsserver unterstützt jedoch auch noch andere Skriptsprachen.
- **import** - Gibt an, welche Klassen durch die JSP importiert werden sollen. Diese Direktive kann mehr als einmal benutzt werden, um mehrere Packages zu laden. Ihre Benutzung macht nur Sinn, wenn „language“ auf den Wert „java“ gesetzt ist.
- **contentType** – Gibt die Art des Seiteninhalts, also den sog. MIME-Typ an (z.B. "text/html" oder "image/gif").

4.2.1.2 Die include Directive

An der Stelle der **include** Direktive wird eine angegebene Datei eingefügt, die dann in der generierten HTML Seite erscheint.

Beispiel:

```
<%@ include file="copyright.html" %>
```

Der Inhalt der Datei wird in das Servlet eingefügt, wenn dies aus der JSP erzeugt wird. Nach einer Veränderung der eingefügten Datei nachträglich verändert, muß das Servlet neu erzeugt werden, damit die Änderungen übernommen werden.

4.2.2 Declarations

Ein Deklarations-Block enthält klassenweite Variablen und Methoden, die dann innerhalb der ganzen JSP aus einer sogenannten *expression* (siehe 4.2.5) aufgerufen werden können. Der Code innerhalb des Deklarations-Blocks wird, in der Regel,

in Java geschrieben; WebSphere unterstützt jedoch auch noch andere Skriptsprachen.

Der Syntax einer Deklaration ist:

```
<%! declaration(s) %>
```

Beispiel:

```
<%!
private static int counter = 0;
private GregorianCalendar myGC = new GregorianCalendar();
private String runSince = myGC.get(Calendar.DATE);

private String since() { return runSince; }
private void resetCount() { counter = 0; }
%>
```

4.2.3 Scriptlets

Scriptlets enthalten Java Code-Fragmente innerhalb einer JSP. Der Code wird ausgeführt; das Ergebnis wird jedoch nicht in die erzeugte Seite geschrieben.

Der Syntax eines Scriptlets ist:

```
<% scriptlet; %>
```

Beispiel:

```
<% resetCount(); %>
<% out.println("Hello World"); %>
```

4.2.4 Kommentare

Innerhalb einer JSP werden zwei Arten von Kommentarblöcken unterstützt.

HTML Kommentare, welche auch in der generierten HTML Seite erscheinen.

Deren Syntax ist:

```
<!-- HTML Kommentar ... -->
```

JSP Kommentare: Diese erscheinen nicht in der generierten HTML Seite. Sie können benutzt werden, um ganze Teile der JSP Seite auszublenden. Der Syntax dieser Kommentare ist:

```
<%-- JSP Kommentar ... --%>
```

4.2.5 Expressions

Mit Hilfe von Expressions kann der Output (als String) aus Java Variablen und Methoden in die HTML Seite eingefügt werden. Der Inhalt des Strings wird dann, an Stelle der Expression, in die erzeugte HTML Seite eingefügt.

Primitive Typen, wie int und float, werden automatisch nach String konvertiert. Eigene Klassen müssen eine **toString()** Methode enthalten. Falls das Ergebnis des Codes innerhalb der Expression nicht nach String konvertiert werden kann, wird eine **ClassCastException** ausgelöst.

Der Syntax dieser Komponente ist:

```
<%= expression %>
```

Beispiele:

```
<%= (counter++) %>
<%= since() %>
```

Bemerkung: Ein Semikolon hinter der `expression` verursacht einen Fehler während der Kompilierung. Bei der Fehlersuche ist dieser Syntaxfehler leicht zu übersehen.

4.2.6 Implizite Objekte

Es gibt eine Reihe von sogenannten impliziten Objekten, die innerhalb von Scriptlets und Expressions verfügbar sind, ohne daß diese vorher deklariert oder importiert wurden.

Im folgenden Beispiel wird das implizite Objekt `out` benutzt, um einen String in die generierte HTML Seite zu schreiben:

```
<% out.println("Using the implicit object <b>out</b>"); %>
```

Die am häufigsten Benutzten impliziten Objekte sind:

- `request`, vom Typ `HttpServletRequest`
- `response`, vom Typ `HttpServletResponse`
- `out`, vom Typ `PrintWriter`
- `session`, vom Typ `HttpSession`

4.3 JSPs erstellen: Die JSP Tags

Zusätzlich zu den JSP Komponenten, die selbstgeschriebenen Java Code enthalten, können JSP Tags benutzt werden, welche eine vordefinierte Aktion auslösen.

Im folgenden werden die am häufigsten benutzten JSP Tags kurz vorgestellt. Tags, die der JSP Spezifikation angehören, beginnen mit **jsp:**. Tags die von WebSphere zur Verfügung gestellt werden und **nicht** der JSP Spezifikation angehören, beginnen mit **tsx:**.

Ausführlichere Informationen über die JSP- und WebSphere-spezifischen Tags finden Sie in der WebSphere Produktdokumentation. Eine vollständige Beschreibung aller JSP Tags finden Sie in der JavaServerPages Spezifikation, Version 1.0, auf der Web-Site der Firma Sun unter der URL:

<http://www.javasoft.com/products/jsp/download.html>.

4.3.1 Tags zur Bearbeitung von Beans

Um ein JavaBean Objekt zu deklarieren, das dann innerhalb der JSP benutzt werden kann, wird der **jsp:useBean** Tag benutzt. Nach der Deklaration können Daten aus dem Bean gelesen und in dies geschrieben werden. Dieses geschieht über die **jsp:setProperty** und **jsp:getProperty** Tags.

Wenn der `jsp:useBean` Tag ausgeführt wird, versucht der Server, das Bean zu finden, in dem er die Werte der Attribute **id** und **scope** des Tags, benutzt. Kann das Bean nicht gefunden werden, wird der Server die Werte der Attribute **scope** und **class** benutzen, um das Bean zu erzeugen.

Ein JavaBean kann jede Klasse sein, die das Interface **serializable** implementiert und über `getXXX` und `setXXX` Methoden verfügt. Über Beans können Daten aus Servlets an JSPs übermittelt werden. Die JSPs können die Daten auslesen und anzeigen.

Die ausführliche Beschreibung einzelnen Attribute dieser Tags, würde den Rahmen dieser Dokumentation sprengen. Konsultieren Sie dazu bitte die WebSphere Produktdokumentation, bzw. die JSP API Dokumentation.

4.3.2. tsx:repeat

Dieser Tag führt einen Block aus HTML und JSP Code mehrfach aus.

Die Schleife wird solange durchlaufen, bis der Wert **ending_index** erreicht wird, oder eine **ArrayIndexOutOfBoundsException** Exception ausgelöst wird. Falls dies in der Mitte der Schleife passiert, werden in der Regel unvollständige HTML Tags generiert! Die Exception sollte also am besten am Anfang des <tsx:repeat>-Tags auftauchen.

Der Syntax dieses Tags ist:

```
<tsx:repeat index=name start=starting_index end=ending_index>
</tsx:repeat>
```

wobei mit:

- **index=name** - der Name der Schleifenvariable definiert wird.
- **start=starting_index** - der Startwert der Schleifenvariable gesetzt wird. Der Standardwert ist 0.
- **end=ending_index** - der Endwert der Schleifenvariable gesetzt wird. Wenn der Tag weggelassen wird, wird als Endwert der Maximalwert genommen. Dieser ist 2.147.483.647.

Beispiel:

```
<tsx:repeat index="idx" start="0" end="5">
  <%= idx %><br>
</tsx:repeat>
```

4.4 JSPs aufrufen

Unter anderem gibt es folgende Möglichkeiten, eine JSP aufzurufen:

4.4.1 JSP über ihre URL aufrufen

Wenn Sie eine Web-Anwendung auf dem Applikationsserver konfigurieren, werden Sie eine Anwendungs-URI angeben, die für Servlets, JSPs, statisches HTML und andere, der Anwendung zugeordnete Komponenten, die Root-URL darstellt. Außerdem können Sie das Anwendungsstammverzeichnis für Dokumente angeben, in dem die JSPs, HTML-Dateien, GIF-Dateien und andere Dateien für die Anwendung abgelegt werden.

Sie können dann eine JSP durch das Öffnen folgender URL-Adressen aufrufen: `http://your.server.name/application_URI/JSP_URL`

Beispiel: <http://localhost/webapp/examples/simple.jsp>

4.4.2 JSP über ein HTML Formular aufrufen

Sie können eine JSP in einem <FORM> Tag aufrufen. Ein HTML-Formular ermöglicht es dem Benutzer, Daten in einer Webseite (in einem Browser) einzugeben und diese Daten anschließend einer JSP zu übergeben.

Um ein JSP aufzurufen gehen Sie wie folgt vor:

```
<FORM METHOD="POST|GET" ACTION="application_URI/JSP_URL">
<!-- Tags für Texteingabe, Knöpfe u.a. Elemente einfügen -->
</FORM>
```

4.4.3 JSP aus einer JSP aufrufen

Wenn Sie die JSP 1.0 Spezifikation benutzen, können sie die Tags `<jsp:forward>` bzw. `<jsp:include>` verwenden, um aus einer JSP heraus eine andere JSP aufzurufen.

5. Datenbankanbindung mit Servlets

Einer der besten Anwendungsmöglichkeiten von Servlets ist die Anbindung von Datenbanken an eine Weboberfläche. In diesem Kapitel wird die Datenbank-anbindung mittels JDBC beschrieben und dann, wie dieses Prinzip in Servlets genutzt werden kann.

5.1 JDBC

Das JDBC API (Java Database Connectivity) bietet eine allgemeine Schnittstelle zum Ausführen von SQL Anweisungen, auf einer relationalen Datenbank, durch ein Java Programm.

Diese Schnittstelle ist unabhängig, sowohl von der verwendeten Datenbank, als auch von der Art der Verbindung zur Datenbank (lokal, Netzwerk). Die Verbindung wird über einen sogenannten Treiber (Driver) hergestellt, der erst zur Laufzeit ausgewählt wird. Die Art der Verbindung und das benutzte Kommunikationsprotokoll wird durch die Wahl der Treibers entschieden.

5.2 JDBC-Treiber

Um über JDBC auf eine Datenbank zugreifen zu können, ist ein, für die Datenbank passender, Treiber notwendig. Dieser wird meist vom Datenbankhersteller zur Verfügung gestellt.

Die Treiber werden in vier Typen eingeteilt:

- **JDBC-ODBC Bridge Treiber** (Typ 1): Diese Treiber bilden eine Brücke zwischen JDBC und ODBC. Es wird (lokal) der passende ODBC⁸ Treiber benötigt.
- **Native-API Partly-Java Treiber** (Typ 2): Diese Treiber verwenden native Bibliotheksaufrufe, um mit der Datenbank zu kommunizieren, und bieten eine java-basierte Schnittstelle, auf der JDBC Seite. Sie bieten eine bessere Performanz als Treiber die komplett in Java geschrieben sind. Auch bei diesem Typ ist eine client-seitig installierte Software notwendig, z.B. braucht der Typ-2 Treiber von Oracle, der sog. „JDBC OCI Driver“, client-seitig diverse DLLs.
- **Net-Protocol All-Java Treiber** (Typ 3): Diese Treiber sind, client-seitig, komplett in Java geschrieben und kommunizieren über ein Netzwerkprotokoll mit einem Server. Dieser übersetzt die Anweisungen in ein Datenbankprotokoll und übermittelt diese an die Datenbank.
- **Native-Protocol All-Java Treiber** (Typ 4): Dieser, komplett in Java geschriebener Treiber, kommuniziert direkt mit der Datenbank.

Der Typ-4 Treiber von Oracle ist der sog. „JDBC Thin Driver“.

⁸ ODBC: OpenDatabase Connectivity: Ein Standard für den Zugriff von Applikationen auf Datenbanken.

Weil Typ 3 und Typ 4 Treiber in reinem Java geschrieben sind, können sie auch in Java Applets benutzt werden.

Eine Übersicht über verfügbare JDBC Treiber existiert unter:
<http://industry.java.sun.com/products/jdbc/drivers>.

5.3 Das Package java.sql

Über die Schnittstellen (Interfaces) im Paket `java.sql` sind Datenbankzugriffe via JDBC möglich. Die Treiber stellen die konkrete Implementierung dieser Interfaces dar.

Im folgenden werden die wichtigsten Klassen und Interfaces dieses Paketes kurz vorgestellt:

- Über die Klasse `DriverManager` wird ein Treiber geladen und eine Datenbankverbindung aufgebaut.
- Die Verbindung ist durch das Interface `Connection` spezifiziert.
- SQL-Anweisungen können über das Interface `Statement`, bzw. dessen Erweiterungen `PreparedStatement` und `CallableStatement`, abgegeben werden.
- Über das Interface `ResultSet` kann auf die Ergebnisse eines Statements zugegriffen werden.
- Die verschiedenen SQL-Datentypen sind als Java-Klassen abgebildet. Diese Zuordnung können Sie in Anhang D nachschlagen.

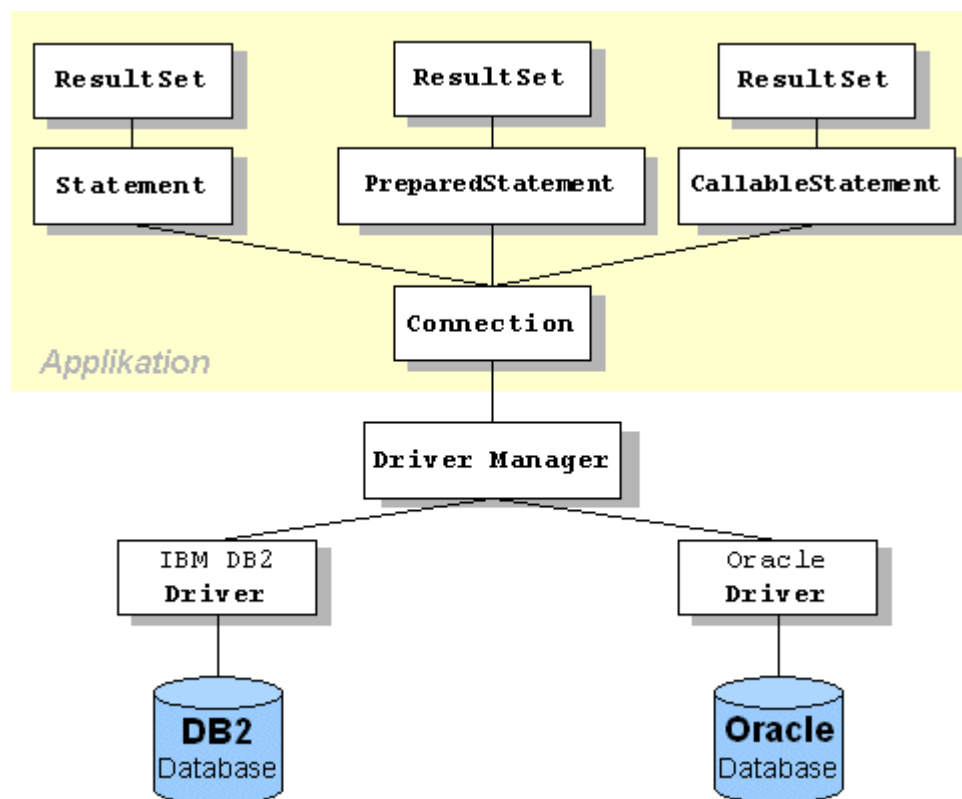


Abbildung 8: Übersicht JDBC

5.3.1 Die Klasse DriverManager

Die Klasse `DriverManager` erfüllt zwei Funktionen. Sie verwaltet die verwendeten JDBC Treiber und erzeugt Verbindungen zu Datenbanken.

Die am häufigsten verwendeten Methoden von `DriverManager` sind:

- Die Methode `registerDriver(Driver)`. Über sie wird ein Treiber dem `DriverManager` bekanntgemacht. Dies ist notwendig bevor der Treiber verwendet werden kann.
- Über die `getConnection` Methode wird eine Verbindung zu einer Datenbank aufgebaut. Beim Aufruf dieser Methode wird die Liste der registrierten Treiber durchlaufen und der erste Treiber verwendet, der eine Verbindung mit der angegebenen Datenbank herstellen kann. Die Methode gibt ein Verbindungs-Objekt zurück, welches das `Connection` Interface implementiert.

Die Methode kann mit einem Argument (URL⁹ String mit Verbindungsziel), zwei Argumenten (URL String und Properties, die Name/Wert Paare enthalten) oder drei Argumenten (URL, Benutzername und Paßwort, als Strings) aufgerufen werden.

Die URL der Datenbank hat i.d.R. folgende Form:

```
jdbc:subprotocol:subname
```

Die Komponente `subprotocol` enthält den Namen des Treibers und/oder die Art der Verbindung (z.B. `oracle:thin`). Die Komponente `subname` hängt von dem verwendeten subprotocol ab. Durch sie wird die Datenbank identifiziert. (z.B. `@127.0.0.1:1521:orcl`).

Beispiele für Datenbank URLs¹⁰:

```
jdbc:oracle:thin:@127.0.0.1:1521:orcl
jdbc:oracle:oci8:@orcl
jdbc:db2:pizzasvc
```

5.3.2 Das Interface Connection

Jede Datenbankverbindung wird durch ein Objekt beschrieben, welches die Schnittstelle `Connection` implementiert. Eine Verbindung erhält man über die Methoden `getConnection(...)` der Klasse `DriverManager`.

Durch Aufrufen von Methoden der Schnittstelle `Connection` können verschiedene Aktionen ausgelöst werden, u.a.:

- Über die Methode `createStatement()` bzw. `prepareStatement(String)` wird ein Objekt zurückgegeben, welches das Interface `Statement` bzw. `PreparedStatement` implementiert. Über diese Objekte können SQL-Befehle auf der Datenbank ausgeführt werden.
- Über `setReadOnly(boolean)` wird die Verbindung in einen „nur-lesen“ Modus versetzt. Dies kann die Effizienz steigern, wenn der Treiber bzw. die Datenbank dies unterstützt.
- Über `commit()` bzw. `rollback()` kann eine Transaktion durchgeschrieben oder zurückgenommen werden. Über `setAutoCommit(boolean)` wird der

⁹ URL: Uniform Resource Locator

¹⁰ Die URLs sind für folgende Treiber (in dieser Reihenfolge): Oracle Thin Driver (Typ 4), Oracle OCI Driver (Typ 2), DB2 App Driver (Typ 2).

„autocommit“ Modus, d.h. Commit nach jedem Statement, an- bzw. ausgeschaltet; standardmäßig ist er aktiviert. Über `getAutoCommit()` kann geprüft werden, ob er aktiv- bzw. inaktiv ist.

- Über die Methode `close()` wird die Verbindung explizit geschlossen. Über `isClosed()` kann dies nachgeprüft werden.

5.3.3 Das Interface Statement

Über das Interface `Statement` werden SQL-Befehle auf einer Datenbank ausgeführt. Objekte, die dieses Interface implementieren, erhält man über die Methode `createStatement()` eines `Connection` Objektes.

In allen Arten von Statements können nur SQL-Befehle nach der SQL-92 Spezifikation verwendet werden. Außerdem kann noch eine Reihe von SQL Funktionen, Prädikaten und Ausdrücken benutzt werden.

U.a. können folgende Methoden verwendet werden:

- Mittels `executeQuery(String sql)` wird ein `SELECT` Statement ausgeführt. Das Ergebnis der Anfrage wird in einem Objekt zurückgegeben, welches das Interface `ResultSet` implementiert.
- Die SQL-Befehle `INSERT`, `UPDATE` und `DELETE`, sowie die unterstützten `DDL`¹¹-Befehle, können mittels `executeUpdate(String sql)` ausgeführt werden. Zurückgegeben wird nur die Anzahl der betroffenen Tupel oder gar nichts.
- Über die `close()` Methode wird ein Statement gelöscht und die verwendeten Ressourcen freigegeben. Dies passiert zwar automatisch wenn das Objekt bei der „garbage collection“ eingesammelt wird, aber es macht Sinn nicht so lange zu warten. Falls ein zugehöriges `ResultSet` existiert, wird dies ebenfalls gelöscht.

5.3.4 Das Interface PreparedStatement

`PreparedStatement` stellt eine Erweiterung des `Statement` Interfaces dar. Es führt einen vorbereiteten, festen (vorkompilierten), SQL-Befehl aus. Die Parameter des Befehls können zur Laufzeit mit Hilfe von Parametermarken verändert werden.

Es unterscheidet sich von einem `Statement` in folgenden Punkten:

- Der Text des SQL-Befehls wird beim Erzeugen des Objektes angegeben. Die Methoden `execute()`, `executeQuery()` und `update()` benötigen deshalb keine Eingabeparameter mehr.
- In einem `PreparedStatement` können, mit dem Zeichen `?`, feste Parametermarken definiert werden. Beispiel:

```
con.prepareStatement(
    "INSERT INTO MUSKETEERS (NAME) VALUES (?)");
```
- Diese können über Methoden der Form `set<Typ>(int, typ)` verändert werden, wobei `int` die Position der Parametermarke angibt und `typ` den einzusetzenden Wert enthält, z.B. `setString(int, String)` bzw. `setDouble(int, double)`.

¹¹ Data Definition Language; enthält Befehle zur Definition von Datenstrukturen.

- Ein `PreparedStatement` ist effizienter als ein `Statement`, da es von der Datenbank vorkompiliert wird.

Folgendes **Problem** kann auftreten, wenn Statements zusammen mit Datenbanken benutzt werden, welche Strings zwischen einfache Anführungszeichen packen. Wenn ein String nur selbst ein einfaches Anführungszeichen enthält (z.B. „John d´Artagan“) wird folgender, normalerweise korrekte, Befehl falsch:

```
String cmd
  = "INSERT INTO MUSKETEERS (NAME) VALUES ('John d´Artagan')";
stmt.executeUpdate(cmd);
```

Dies ist z.B. bei Oracle der Fall. Eine Lösungsmöglichkeit ist, jeden String zu durchsuchen, und darin enthaltene einfache Anführungszeichen durch doppelte einfache Anführungszeichen, also `', zu ersetzen. Dies ist die „escape“ Sequenz, die in Oracle dafür vorgesehen ist, ist jedoch eine umständliche Lösung. Die zweite Möglichkeit ist ein `PreparedStatement` zu verwenden und Parameter über die `setString(int, String)` Methode einzusetzen.

```
PreparedStatement pstmt
  = con.prepareStatement(
    "INSERT INTO MUSKETEERS (NAME) VALUES (?)");
pstmt.setString(1, "John d´Artagan");
pstmt.executeUpdate();
```

5.3.5 Das Interface ResultSet

Über Objekte, die das Interface `ResultSet` implementieren, wird auf die Ergebnisse der Befehle zugegriffen, die mit einem `Statement` ausgeführt wurden. Der Zugriff auf die Tupel des Ergebnisses kann nur sequentiell erfolgen.

Einige Methoden dieses Interfaces sind:

- Die Methode `next()`. Mit ihr wird das erste bzw. das nächste Tupel des Ergebnisses angesteuert. Sie liefert *true* zurück, solange noch weitere, ungelesene Tupel existieren
- Über Methoden der Form `get<Typ>(int columnIndex)` oder `get<Typ>(String columnName)` können einzelne Werte, aus dem aktuell sichtbaren Tupel, gelesen werden. *Typ* ist ein Java-Typ (z.B. `String` oder `Int`); *columnIndex* die Spaltenzahl (beginnend mit 1), *columnName* ist ein Spaltenname.

Es ist i.d.R. besser, über die Spaltennamen auf das Tupel zuzugreifen und **nicht** über die Spaltenzahl. Dies hat den Vorteil, daß das Programm Änderungen der Tabellenstruktur in der Datenbank, z.B. hinzufügen von zusätzlichen Spalten, toleriert.

- Über die Methode `wasNull()` kann nachgeprüft werden, ob das zuletzt aus dem Tupel gelesene Element (Attribut), den SQL-Wert `NULL` hatte (im Gegensatz zum Java Wert *null*).
- Der Aufruf `close()` schließt das Objekt. Dies geschieht auch, wenn das zugehörige `Statement` Objekt geschlossen wird.

5.4 WebSphere Datenquellen

Unter WebSphere v3.0 können für die Datenbankzugriffe, neben `JDBC` Aufrufen, auch sog. Datenquellen (**DataSource Objekte**) benutzt werden. Innerhalb von

WebSphere bedeutet die Verwendung eines DataSource Objekts zum Abrufen von Verbindungen, daß die Effizienz des **Verbindungs-Poolings** automatisch zur Geltung kommt.

5.4.1 Was ist Verbindungs-Pooling?

Durch Verbindungs-Pooling wird die Anzahl der zu öffnenden und zu schließenden Datenbankverbindungen reduziert.

Dies geschieht nach folgendem Prinzip: Die Servlets holen sich ihre Datenbankverbindungen nicht mehr direkt von der Datenbank sondern aus einem **Verbindungs-Pool**. Dieser enthält eine vordefinierte Anzahl von unbenutzten Datenbankverbindungen und gibt eine davon an das Servlet weiter. Während dessen ist die Verbindung als **benutzt** gekennzeichnet und kann nicht von anderen Servlets verwendet werden. Hat das Servlet seinen Datenbankzugriff beendet, gibt es die Verbindung an den Pool zurück und dieser kann sie nun einem anderen Servlet zuteilen.

Auf diese Weise kann eine Verbindung von mehreren Servlets hintereinander benutzt werden, und es wird ein ständiges Öffnen und Schließen von Datenbankverbindungen vermieden. Da diese beiden Aktivitäten eine hohe Systembelastung verursachen, wird durch deren Minimierung die Geschwindigkeit der Datenbankzugriffe beschleunigt und die Systemleistung erhöht.

Das Verbindungs-Pooling ist als Teil der JDBC 2.0 Standard Extension API definiert. Dieses API ermöglicht den Herstellern, zusätzliche JDBC Funktionsarten bereit zu stellen. Die entsprechenden Klassen und Schnittstellen wurden von IBM in den Paketen `com.ibm.db2.jdbc.app.stdext.javax.sql` und `com.ibm.ejs.dbm.jdbcext` implementiert.

Ein weiterer Teil der API ermöglicht den Zugriff auf DataSource Objekte und die Benutzung des Java Naming and Directory Interface (JNDI) für den Zugriff auf relationale Daten-Server.

5.4.2 Erzeugen einer Datenquelle

Bevor ein Verbindungs-Pool benutzt werden kann, müssen ein JDBC-Treiber und eine Datenquelle (DataSource Objekt) in WebSphere angemeldet werden. Dies geschieht über die WebSphere Administrationskonsole.

Folgendes Bild zeigt wie der Oracle JDBC-Treiber angemeldet wird:



Abbildung 9: Oracle JDBC-Treiber in WebSphere anmelden

Dieses Bild zeigt die Erstellung einer Datenquelle. Sie wird vom Servlet im Abschnitt E.3.3 verwendet.

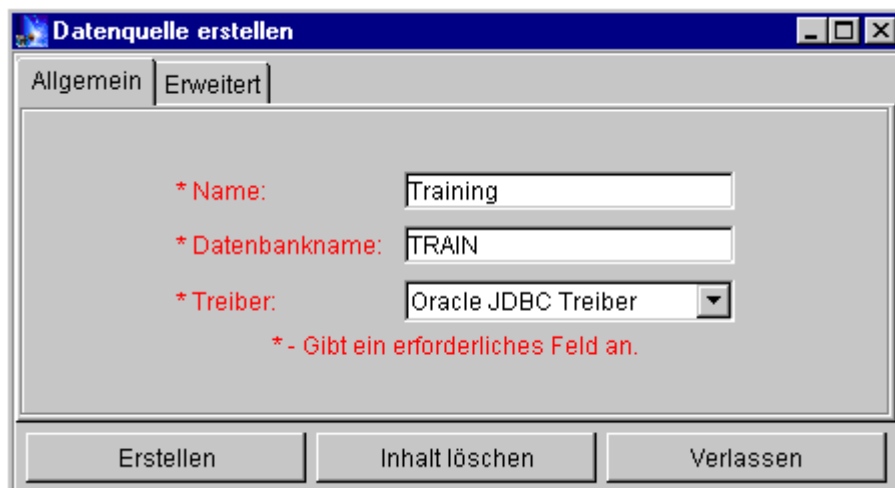


Abbildung 10: Datenquelle erstellen

In der Karte **Erweitert** können die Parameter für das Verbindungs-Pooling eingestellt werden: Die Mindestgröße an Verbindungen im Pool, die Maximalanzahl an Verbindungen, das Verbindungs-Zeitlimit, das Leerlauf-Zeitlimit und das Zeitlimit für verwaiste Prozesse. Diese Eigenschaften werden in der WebSphere Produktdokumentation erläutert.

5.4.3 Benutzen einer Datenquelle

Um die erzeugte Datenquelle zu benutzen sind folgende Schritte notwendig:

1. Importieren der JDBC Pakete, der Implementierung der JDBC Standard Extensions und des JNDI¹² Interface:

```
import java.sql.*;
import java.math.*;
// Packages für JDBC Extensions von IBM und Naming Service
import com.ibm.db2.jdbc.app.stdext.javax.sql.*;
import com.ibm.ejs.dbm.jdbcext.*;
import javax.naming.*;
```

- Vom Servlet wird der Namens-Service verwendet, um auf ein `DataSource` Objekt zugreifen zu können, welches vom WebSphere Administrator erstellt wurde. Dazu muß zuerst ein `Hashtable` Objekt erstellt werden, in dem die, für den Zugriff auf den Namens-Service erforderlichen, Parameter gespeichert werden.

```
Hashtable parms = new Hashtable();
```

- Nun werden die Parameter in das `Hashtable` geladen:

```
parms.put(Context.INITIAL_CONTEXT_FACTORY,
           "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
```

- Nun kann ein `Context` Objekt erstellt werden, in dem der Namenskontext gespeichert wird. Über dieses, kann dann auf den Inhalt des Namens-Service zugegriffen werden.

```
Context ctx = new InitialContext(parms);
```

- Mit Hilfe des `Context` Objektes kann auf eine Datenquelle (`DataSource`) zugegriffen werden.

```
DataSource ds = (DataSource)ctx.lookup(source);
```

Über den String `source` wird die Datenquelle identifiziert. Es hat die Form "`jdbc/Datenquelle-Name`", also z.B. "`jdbc/Training`".

- Das `DataSource` Objekt wird (anstatt eines `DriverManager` Objektes) benutzt, um eine Verbindung aus dem Pool zu bekommen. Eine Verbindung wird für jede Client Anforderung (`doPost()` bzw. `doGet()`) hergestellt.

```
Connection conn = ds.getConnection(db_user, db_password);
```

- Nun können, wie bereits in den vorhergehenden Abschnitten gezeigt, SQL Befehle über `Statement` Objekte abgegeben werden und deren Ergebnisse abgefragt werden.
- Zuletzt wird die Verbindung, über `conn.close()`, an den Pool zurückgegeben.

Wie dies implementiert wird, wird durch das Beispiel in Abschnitt E.3.3 demonstriert.

5.6 Data Access Beans

Die Entwicklungsumgebung IBM Visual Age for Java, in der Ausführung Enterprise, bietet ein Tool zum visuellen Erstellen von Data Access Beans und von Servlets.

Über **Data Access Beans** kann auf einfache Art auf relationale Daten zugegriffen werden, da sie mit Hilfe eines Assistenten, visuell erstellt werden.

Servlets können mit dem Tool **Servlet Builder**, visuell erstellt werden. In visuell erstellte Servlets, können dann Data Access Beans eingebaut werden. Auf diese Weise können, ohne großen Programmieraufwand, Datenbankinhalte in Servlets eingebaut werden.

Die Vorstellung dieser beiden Tools würde den Umfang dieser Dokumentation sprengen. Sie sind in dem **Redbook SG24-5265-00** (siehe Anhang A), auf über 100 Seiten, beschrieben.

6. Zustandserhaltung

Das HTTP Protokoll ist ein **zustandloses** Protokoll. Ein Client hat keine permanente Verbindung zum Webserver, sondern verbindet sich jedesmal neu, wenn er eine Anfrage sendet. Im Normalfall kann der Server die einzelnen Anfragen nicht einem Clienten zuordnen. Dies ist jedoch wünschenswert, um einen Ablauf in mehrere Einzelschritte unterteilen zu können (z.B. eine Bestellung) und trotzdem die Daten aus den einzelnen Schritten zusammenfügen zu können.

Um dies möglich zu machen, ist es notwendig, daß sich der Client gegenüber dem Server bei jeder Anfrage identifiziert, um so eine **Zustandserhaltung** zu ermöglichen. Die dazu am häufigsten verwendeten Möglichkeiten sind:

- **URL-Rewriting:** Jede URL, die der Benutzer anklicken kann, wird dynamisch modifiziert, um zusätzliche Informationen an den Server zu senden. Da die Länge einer URL beschränkt ist, werden nur die notwendigsten Informationen an den Server geschickt, wie z.B. eine Identifikationskennung.
- **Cookies:** Diese spezielle Zeichenkette wird vom Server an den Clienten geschickt. Der Client sendet fortan bei jeder Anfrage das Cookie zurück. Ein Cookie hat i.d.R. eine Maximallänge von 4096 Bytes und ein definiertes Verfallsdatum. Ob ein Cookie zurückgeschickt wird, wird nach bestimmten Regeln entschieden.

Mit diesen Hilfsmitteln läßt sich Zustandserhaltung auch in Servlets realisieren. Dies selbst zu entwickeln ist aber noch sehr umständlich, weil die Zuordnung der Benutzeridentifikation zu den Daten nicht automatisch erfolgt.

Um dies zu erleichtern, wurde im Servlet API das **HttpSession Objekt** definiert, welches im nächsten Abschnitt vorgestellt wird.


6.1 Das HttpSession Objekt

Das HttpSession Objekt bietet eine besonders einfache Möglichkeit, um Zustandserhaltung zu realisieren.

Beim Erzeugen eines solches Objektes schickt der Server automatisch einen **nicht-persistenten**¹³ Cookie, mit der ID der Session, an den Client zurück. Auf diese Weise können die weiteren Anfragen des Clients dem zugehörigen Session Objekt zugeordnet werden. Über entsprechende Methoden können Daten in das Session Objekt geschrieben und daraus gelesen werden.

Bleibt das erzeugte Session Objekt für eine längere Zeit unbenutzt (unter WebSphere standardmäßig 30 Minuten), wird es vom Server gelöscht, um den verwendeten Speicher wieder freizugeben.

¹³ Ein nicht-persistenter Cookie wird beim Beenden des Webbrowsers gelöscht. WebSphere bietet zusätzlich die Möglichkeit **persistente** Cookies zu verwenden, die vom Browser gespeichert werden. Diese können zusammen mit persistenten Session Objekten, die in einer Datenbank gespeichert werden, benutzt werden, um Daten auch zu einem späteren Zeitpunkt diesem Client (Browser) zu zuordnen.

Das Verhalten von HttpSession Objekten wird über das Element **Session Manager** , in der **Topologie** Ansicht der WebSphere Administrationskonsole bestimmt. Näheres darüber erfahren Sie in der WebSphere Produktdokumentation.

Um eine HttpSession für eine Anfrage zu erzeugen, werden folgende Methoden des **HttpServletRequest** Objektes benutzt:

- **getSession(boolean create)** – Liefert die mit diesem Client assoziierte Session zurück bzw. erzeugt eine neue. Ob eine neue Session erzeugt wurde, kann über die `isNew()` Methode des `HttpSession` Objektes abgefragt werden. Wurde `getSession` mit dem Wert `false` aufgerufen, und existiert keine assoziierte Session, so hat das zurückgegebene Session Objekt den Wert `null`.
- **getSession()** – Entspricht dem Aufruf `getSession(true)`.

Die am häufigsten verwendeten Methoden des **HttpSession** Objektes sind:

- **isNew()** – Liefert `true` zurück, falls das Session Objekt eben neu erzeugt wurde, also der Client noch nicht über diese Session informiert wurde¹⁴.
- **putValue(String name, Objekt value)** – Schreibt ein Objekt *value* mit dem Namen *name* in die Session. Der Wert von *name* wird verwendet, um das Objekt wieder aus der Session herauszuholen (über `getValue`).
- **getValue(String name)** – Holt das Objekt mit der angegebenen Kennung *name* aus der Session. Ist kein Objekt mit diesem Namen in der Session vorhanden, wird `null` zurückgeliefert. Nach dem Herausholen muß ein `Typecast` (eine Umwandlung) von `java.lang.Object` in die benutzte Klasse erfolgen.
- **getValueNames()** – Liefert ein Array von Strings zurück, welches alle Kennungen der in der Session gespeicherten Objekte enthält
- **removeValue(String name)** – Löscht das Objekt mit dem angegebenen Namen aus der Session.

Anmerkung: Im allerneuesten Servlet API v. 2.2 sind diese Methoden als *deprecated* markiert und wurden durch `getAttribute`, `putAttribute`, `removeAttribute` usw. ersetzt. In der API v. 2.1, die von WebSphere unterstützt wird, gelten jedoch noch die oben angegebenen Methoden.

¹⁴ Sind Cookies client-seitig deaktiviert und werden nur auf Cookies basierende Sessions unterstützt (Standardeinstellung), so wird die Methode immer `true` zurückliefern.

Anhang A: Quellen und Links

A.1 Quellen

- IBM WebSphere Studio and VisualAge for Java
Servlet and JSP Programming
Redpiece¹⁵ SG24-5755-00,
Benutzte Version: 14.3.2000. Aktuelle Version: 1.5.2000.
Engültige Fassung voraussichtlich am 15.5.2000 erhältlich.
<http://www.redbooks.ibm.com/redpieces/abstracts/sg245755.html>
- IBM WebSphere and VisualAge for Java
Database Integration with DB2, Oracle and SQL Servler
Redbook SG24-5471-00
<http://www.redbooks.ibm.com/abstracts/sg245471.html>
- Java Servlet Programming
Jason Hunter, William Crawford
O'Reilly Verlag, Oktober 1998
ISBN 1-56592-391-X

A.2 Links

- IBM WebSphere Application Server,
Informationen und Download unter:
<http://www-4.ibm.com/software/webservers/appserv/>
- Java Servlet API Dokumentation & Spezifikation
Download unter:
<http://www.javasoft.com/products/servlet/download.html#specs>
- Java JavaServerPages Spezifikation
Download unter:
<http://java.sun.com/products/jsp/download.html>
- Übersicht - Erhältliche JDBC Treiber
<http://industry.java.sun.com/products/jdbc/drivers>

A.3 Weiterführende Literatur

- VisualAge for Java Enterprise Version 2:
Data Access Beans – Servlets – CICS Connector
Redbook SG24-5265-00
<http://www.redbooks.ibm.com/abstracts/sg245265.html>
- The Front of IBM WebSphere
Building e-business User Interfaces
Redbook SG24-5488-00
<http://www.redbooks.ibm.com/abstracts/sg245488.html>

¹⁵ Ein Redpiece ist ein noch nicht abgeschlossenes Redbook, d.h. es wird noch bearbeitet.

- Programmieren mit dem älteren WebSphere v2.0 bzw. JSP API v. 0.92:
WebSphere Application Servers:
Standard and Advanced Editions
Redbook SG-24-5460-00
<http://www.redbooks.ibm.com/abstracts/sg245460.html>
- JDBC API Tutorial and Reference, Second Edition
Universal Data Access for the Java 2 Platform
White, Fisher, Cattell, Hamilton, Hapner
Addison Wesley Verlag, September 1999
ISBN 0-201-43328-1

Anhang B: Quellennachweis der Bilder

Abbildung 1: IBM Redpiece SG24-5755-00, Seite 4

Abbildung 2: IBM Redbook SG24-5423-00, Seite 8

Abbildung 3: IBM Redpiece SG24-5755-00, Seite 40

Abbildung 4: Java Servlet Programming, Hunter & Crawford, O' Reily Verlag, Seite 18

Abbildung 5: IBM Redpiece SG24-5755-00, Seite 43

Abbildung 8: IBM Redpiece SG24-5755-00, Seite 96

Alle anderen Bilder wurden selbst erstellt.

Anhang C: Attribute der JSP Directive „page“

Attribute Name	Description
language	Identifies the scripting language used in scriptlets in the JSP file or any of its included files. JSP supports only the value of "java". WebSphere extensions provide support for other scripting languages. <%@ page language = "java" %>
extends	The fully-qualified name of the superclass for which this JSP page will be derived. Using this attribute can effect the JSP engine's ability to select specialized superclasses based on the JSP file content and should be used with care.
import	When the language attribute of "java" is defined, the attribute specifies the additional files containing the types used within the scripting environment. <%@ page import = "java.util.*" %>
session {true false}	If true, specifies that the page will participate in an Http session and enables the JSP file access to the implicit session object. The default value is "true".
buffer {none sizekb}	Indicates the buffer size for the JspWriter. If none, the output from the JSP is written directly to the ServletResponse PrintWriter object. Any other value results in the JspWriter buffering the output up to the specified size. The buffer is flushed in accordance with the value of the autoFlush attribute. The default buffer size is no less than 8kb.
autoFlush {true false}	If true, the buffer will be flushed automatically. If false, an exception is raised when the buffer becomes full. The default value is "true".
isThreadSafe {true false}	If true, the JSP processor may send multiple outstanding clients requests to the page concurrently. If false, the JSP processor sends outstanding client requests to the page consecutively, in the same order in which they were received. The default is "true".
info	Allows the definition of a string value that can be retrieved using Servlet.getServletInfo().
errorPage	Specifies the URL to be directed to for error handling if an exception is thrown and not caught within the page implementation. In the JSP 1.0 specification, this URL must point to a JSP page.
isErrorPage {true false}	Identifies that the JSP page refers to a URL identified in another JSP's errorPage attribute. When this value is true, the implicit variable exception is defined and its value set to references the Throwable object of the JSP source file which cause the error.
contentType	Specifies the character encoding and MIME type of the JSP response. Default value for contentType is text/html. Default value for charSet is ISO-8859-1. The syntax format is: contentType="text/html"; charset="ISO-8859-1"

Anhang D: Zuordnung SQL-Datentypen zu Java-Datentypen

SQL-Datentyp	Java-Datentyp
CHAR, VARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.TimeStamp

Anhang E: Programmcode Beispiele

E.1 Codebeispiele für Servlets

E.1.1 Das HelloWorld Servlet

Dieses einfache Servlet gibt bloß einen kurzen Gruß aus. Es kann sowohl HTTP GET als auch POST Befehle verarbeiten. Sie können sehen, wie der Inhalts-Typ (ContentType) der Antwort gesetzt wird, und wie eine Antwort in das Response Objekt geschrieben wird.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

/**
 * Ein einfaches HelloWorld Servlet
 */
public class HelloWorldServlet extends javax.servlet.http.HttpServlet
{

    /**
     * Verarbeitet HTTP GET Requests
     */
    public void doGet(HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException, IOException
    {
        // Inhalts-Typ der Response setzen. Immer dann notwendig
        // wenn der PrintWriter der ServletResponse benutzt wird.
        res.setContentType("text/html");

        // Hole PrintWriter der ServletResponse
        PrintWriter out = res.getWriter();
        // Erzeuge HTML Seite mit Antwort
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hallo Welt Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h2>Hallo Welt</h2>");
        out.println("</body>");
        out.println("</html>");
    }

    /**
     * Verarbeitet HTTP POST Requests
     */
    public void doPost(HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException, IOException
    {
        /* Ruft doGet(...) auf. So unterstützt das Servlet sowohl GET
        als auch POST Requests. Es ist auch möglich unterschiedliche
        Aktionen für doGet und doPost zu programmieren, indem hier
        doGet NICHT aufgerufen wird.

        Sollen keine POST Requests bearbeitet werden,
        kann diese Methode gelöscht werden.
        */
        doGet(req, res);
    }
}
```

E.1.2 Das Sample1Servlet

Dieses etwas kompliziertere Servlet demonstriert die Benutzung der `init()` Methode. Außerdem werden einige Informationen aus dem Request gelesen und angezeigt. Zusätzlich wird gezeigt, wie Sie verhindern können, daß die Seite im Cache des Webbrowsers gespeichert wird. So können Sie sicherstellen, daß der Benutzer immer die aktuellste Seite erhält.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

/**
 * Dieses Servlet demonstriert die Verwendung der init() Methode und
 * die Möglichkeit Daten aus der Request auszulesen.
 */
public class Sample1Servlet extends javax.servlet.http.HttpServlet {
    java.util.GregorianCalendar myGC;
    public String runningSince;

    /**
     * Verarbeitet HTTP GET Anforderungen.
     */
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        // Inhalts-Typ der Response setzen. Immer dann notwendig
        // wenn der PrintWriter der ServletResponse benutzt wird.
        res.setContentType("text/html");

        // Caching ausschalten (für die Seite die erzeugt wird)
        // Verhindert das eine, moeglicherweise im Cache vorhandene,
        // ältere Seite angezeigt wird.
        res.setHeader("Pragma", "no-cache");
        res.setDateHeader("Expires", 0);
        res.setHeader("Cache-Control", "no-cache");

        // Hole PrintWriter der ServletResponse
        PrintWriter out = res.getWriter();
        // Erzeuge HTML Seite mit Antwort
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hallo Welt</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h2>Hallo Welt</h2>");

        // Einige Daten aus der Request auslesen
        out.println("Servlet wurde mit folgender URL aufgerufen:<b> ");
        out.println(req.getRequestURI());
        out.println("</b><br>Es wurde folgender HTTP Befehl verarbeitet:<b>
");
        out.println(req.getMethod());

        // Eine Instanzvariable abfragen
        out.println("</b><br><br>Dieses Servlet la&uuml;ft seit:<b>
"+runningSince);
        out.println("</b></body>");
        out.println("</html>");
    }

    /**
     * Initialisiert den String "runningSince".
     * Diese Methode wird automatisch beim Instanzieren des
     * Servlets aufgerufen, also nur einmal (da Servlets nur
     * einmal instanziiert werden).
     */
    public void init() throws javax.servlet.ServletException {
        myGC = new java.util.GregorianCalendar();
        runningSince = myGC.get(java.util.Calendar.YEAR) + "/"
```

```

    + myGC.get(java.util.Calendar.MONTH) + "/"
    + myGC.get(java.util.Calendar.DATE) + " "
    + myGC.get(java.util.Calendar.HOUR) + ":"
    + myGC.get(java.util.Calendar.MINUTE);
}
}

```

E.1.3 HTTP POST verarbeiten (Formular auswerten)

Bis jetzt haben wir noch nicht gezeigt, wie Sie Daten verarbeiten können, die über HTTP POST übermittelt wurden. Dies wird in diesem Abschnitt demonstriert. Dazu wird ein einfaches Formular verwendet, welches diese Elemente enthält:

- Ein Feld „Name“
- Eine Liste Begrüßung
- Zwei Schaltflächen „Knopf 1“ und „Knopf 2“

Das Formular wird durch folgenden HTML-Seite erzeugt:

```

<html>
<head><title>Form (POST) Sample</title></head>

<body text="#000000" bgcolor="#CCFFFF">
  <center>
    <font color="#000099" size=+2>Form Input Page</font>
  </center>

  <form action = "/servlet/FormServlet" method = "POST">
  <br><br>
  <center>
    <table cols=2 width="75%">
    <tr>
      <td>Name:</td>
      <td><input type = "text" name = "fieldName"></td>
    </tr>
    <tr>
      <td>Begrüßung:</td>
      <td><select name="liste" size="1">
        <option value="100">Guten morgen</option>
        <option value="200">Guten abend</option>
      </select>
    </tr>
    <tr><td>&nbsp;</td>
      <td>&nbsp;</td>
    </tr>
    <tr>
      <td><input type="submit" name="button" value="Knopf 1"></td>
      <td><input type="submit" name="button" value="Knopf 2"></td>
    </tr>
  </table>
  <p>Drücken Sie auf eine der Schaltflächen !</p>
  </center>
</form>
</body>
</html>

```

Wie Sie im HTML-Code sehen können, wird das Servlet „FormServlet“ über den Befehl: `<form action = "/servlet/FormServlet" method = "POST">` aufgerufen.

Das Servlet besteht aus folgendem Code:

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

/**

```

```

* Wertet ein HTML Formular aus.
*/
public class FormServlet extends javax.servlet.http.HttpServlet {

/**
* Verarbeitet einen HTTP POST Befehl.
* Diverse Daten werden aus einem Formular übermittelt.
*/
public void doPost(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
    PrintWriter pw;
    java.util.Enumeration parameters;
    String listValue, buttonValue;

    res.setContentType("text/html");
    pw = res.getWriter();

    pw.println("<html>");
    pw.println("<head>");
    pw.println("<title>FormServlet Output</title>");
    pw.println("</head>");
    pw.println("<body>");
    pw.println("<h2>FormServlet Output:</h2>");

    // Zeige alle übermittelten Parameter an
    parameters = req.getParameterNames();
    pw.println("<p>Folgende Parameter wurden
&uuml;bermittelt:<br><br>");
    while (parameters.hasMoreElements())
        pw.println((java.lang.String) parameters.nextElement()+"<br>");

    pw.println("</p><p>");
    // Liste auslesen
    listValue = req.getParameterValues("liste")[0];
    if (listValue.equals("100"))
        pw.println("Guten morgen, ");
    else if (listValue.equals("200"))
        pw.println("Guten abend, ");

    // Feld Name auslesen
    pw.println(req.getParameterValues("fieldName")[0]);
    pw.println("</p>");

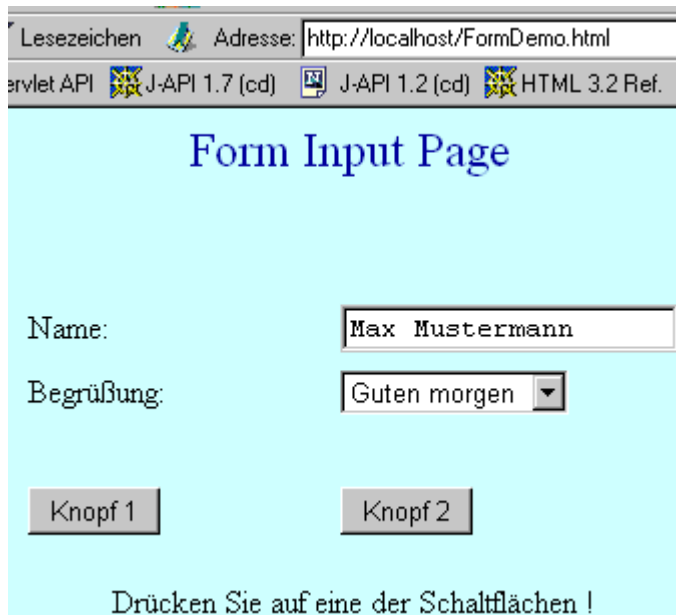
    // Welcher Knopf wurde gedrückt ?
    buttonValue = req.getParameterValues("button")[0];
    if (buttonValue.equals("Knopf 1"))
        pw.println("<br><b>Knopf 1</b> wurde gedr&uuml;ckt.");
    else if (buttonValue.equals("Knopf 2"))
        pw.println("<br><b>Knopf 2</b> wurde gedr&uuml;ckt.");

    pw.println("</body></html>");
}
}

```

Interessant sind dabei folgende Befehle: **req.getParameterNames()** erzeugt ein Enumeration Objekt mit den Namen aller übermittelten Parameter. Dies ist in dem Fall nützlich, in dem die Namen der übermittelten Parameter aus der HTML-Seite nicht bekannt sind. **req.getParameterValues(String)** liefert ein Array Objekt zurück, welches alle Werte des im String angegebenen Parameters enthält, bzw. den Wert *null*, falls dieser Parameter nicht existiert.

Folgende Bilder zeigen das Formular und die vom Servlet generierte Antwort:



Lesenzeichen Adresse: http://localhost/FormDemo.html

Servlet API J-API 1.7 (cd) J-API 1.2 (cd) HTML 3.2 Ref.

Form Input Page

Name:

Begrüßung:

Drücken Sie auf eine der Schaltflächen !

Abbildung 11: HTML Formular



Adresse: http://localhost/servlet/FormServlet

FormServlet Output:

Folgende Parameter wurden übermittelt:

- liste
- feldName
- button

Guten morgen, Max Mustermann

Knopf 2 wurde gedrückt.

Abbildung 12: Antwort des Servlets FormServlet

E.2 Codebeispiele für JSPs

E.2.1 Ein sehr einfaches JSP

Dieses ist ein Beispiel für eine sehr einfache JSP. Es wird die (immer notwendige) Direktive `page` benutzt, um dem JSP-Parser die verwendete Skriptsprache und die Art des Seiteninhaltes (`text/html`) mitzuteilen. Außerdem wird ein String, mittels des impliziten Objektes `out`, in die Seite geschrieben.

```
<html>
<title>Ein einfaches JSP</title>

<body>

<!-- Direktiven -->
<%@ page language = "java" %>
<%@ page contentType = "text/html" %>

<h1>Ein einfaches JSP</h1>

<p>
  Hallo Welt !
</p>

<!-- Implizites Objekt benutzen -->
<% out.println("Dies ist <b>simple.jsp</b>"); %>

</body>
</html>
```

Die JSP erzeugt folgende Ausgabe:



Ein einfaches JSP

Hallo Welt !

Dies ist **simple.jsp**

Abbildung 13: Antwort von simple.jsp

E.2.2 Formular auslesen durch eine JSP

Diese JSP demonstriert wie man Daten aus einem Formular auswerten kann.

Sie liest den Parameter "parm1" aus der impliziten Objekt `request` aus. Der Parameter wurde vom Web-Formular, per HTTP POST, an die JSP übermittelt. Außerdem wird, über das implizite Objekt `response`, das Zwischenspeichern der Seite verhindert.

Die JSP wird durch folgenden Code erzeugt:

```
<html>
<head>
  <title>Formular auslesen - JSP</title>
</head>

<!-- page Direktive -->
<%@ page language = "java" %>
<%@ page contentType = "text/html" %>

<body text="#000000" bgcolor="#FFFF99">
```



```

<!-- Declaration -->
<%!
// Fuer den Wert aus dem Formular
private String parm1Val = "test";
%>

<!-- Scriptlet: Benutzt implizite Objekte request und response -->
<%
// Parameter "parm1" aus Formular auslesen
try {
parm1Val = request.getParameterValues("parm1")[0];
}
catch (NullPointerException e) { parm1Val = "leer"; }

// Zwischenspeichern der Seite verhindern
response.setHeader("Pragma","no-cache");
response.setHeader("Cache-Control","no-cache");
response.setDateHeader("Expires",0);
%>

<H1>Formular auslesen - JSP</H1>
<p>Das Formular hat folgenden Wert &uuml;bermittelt:<b>
<%= parm1Val %>
</b>
</p>
</body>
</html>^

```

Das verwendete Formular wird durch diesen HTML-Code erzeugt:

```

<html>
<head>
<title>JSP Beispiel</title>
</head>

<body text="#000000" bgcolor="#CCFFFF">
<center><font color="#000099" size=+2>
Input Page
</font></center>

<form action = "form.jsp" method = "POST">
<center>
<table cols=2 width="75%">
<tr>
<td>Eingabe:</td>
<td><input type = "text" name = "parm1"></td>
</tr>
<tr>
<td>Bitte auf den Knopf dr&uuml;cken...</td>
<td><input type = "submit" VALUE="Senden"></td>
</tr>
</table>
</center>
</form>
</body>
</html>

```

E.2.3 Zusammenarbeit von Servlet und JSP mit Bean

Servlets können Daten in Java Beans packen und diese an JSPs übermitteln. Die JSPs greifen auf die Beans zu und zeigen die dort gespeicherten Daten an.

Im diesem Beispiel wird eine Textnachricht (String) in einem einfachen Bean gespeichert (MessageBean). Das Bean wird in einem HttpSession Objekt hinterlegt. Danach wird die JSP aufgerufen. Diese holt sich das Bean aus der Session und zeigt die gespeicherte Nachricht an. Mehr über das HttpSession Objekt erfahren Sie in Kapitel 7.

Damit das Bean vom Applikationsserver gefunden wird, muß es in einem Verzeichnis hinterlegt werden, daß den Namen der Package trägt (z.B. /

servlets/myBeans/).

Das MessageBean ist durch folgenden Code definiert:

```
package mybeans;

import java.io.*;

/** Diese Klasse implementiert ein Bean, welches eine Textnachricht
    (einen String) speichert.
    */
public class MessageBean implements Serializable {
    private String text;

    /** Erzeugt ein neues MessageBean mit leerer ("" ) Nachricht. */
    public MessageBean() {}
    /** Erzeugt ein neues MessageBean mit d. Nachricht aus "text".*/
    public MessageBean(String text) { this.text = text; }

    /** Holt die gespeicherte Nachricht aus dem Bean zurück. */
    public String getMessage() { return text; }
    /** Setzt die im Bean gespeicherte Nachricht auf "text". */
    public void setMessage(String text) { this.text = text; }
}

```

Dieser Code erzeugt das Servlet. Folgendes ist zu beachten: Die Ziel-URL der JSP Datei ist im String JSP_URL angegeben. Mit der Methode **encodeURL** wird diese URL für die Weiterleitung vorbereitet (genaueres siehe API Dokumentation). Mit **sendRedirect** wird die Weiterleitung eingeleitet.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import mybeans.MessageBean;

/**
 * Servlet das eine JSP mit einem Bean beliefert
 */
public class BeanDemoServlet extends HttpServlet {
    private static final String JSP_URL =
        "http://localhost/training/beanDemo.jsp";

    /**
     * Erzeugt ein Bean, speichert es in der Session
     * und ruft eine JSP auf.
     */
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        // Erzeuge Bean
        MessageBean msg =
            new MessageBean("Dieser Text wird im Bean gespeichert.");

        // Erzeuge session (wg. true)
        HttpSession mySession = req.getSession(true);
        // Schreibe Bean in die Session
        mySession.putValue("message", msg);

        // Erzeuge URL für Weiterleitung
        String goToURL = res.encodeRedirectURL(JSP_URL);
        // Rufe URL auf
        res.sendRedirect(goToURL);
    }
}

```

Die aufgerufen JSP holt das Bean aus dem HttpSession Objekt und instanziiert es mit dem **jsp:useBean** Tag. Mit dem Tag **jsp:getProperty** wird die Property Message abgefragt.

```
<html>
<title>Bean Demo JSP</title>
```

```
<body>

<!-- Directiven -->
<%@ page language = "java" %>
<%@ page contentType = "text/html" %>

<!-- Hole MessageBean mit Namen "message" aus dem Session Objekt -->
<jsp:useBean id="message" class="mybeans.MessageBean"
scope="session" />

<h1>Beispiel für Bean Benutzung</h1>

<p>Hier ist der Text aus dem Bean (von der Property Message):</p>
<p><b>
  <!-- Lese die Property Message aus -->
  <jsp:getProperty name="message" property="Message" />
</b></p>
</p>

</body>
</html>
```

Das Ergebnis sieht so aus:

Beispiel für Bean Benutzung

Hier ist der Text aus dem Bean (von der Property Message):

Dieser Text wird im Bean gespeichert.

Abbildung 14: Antwort von beanDemo.jsp

E.3 JDBC Beispiele

E.3.1 Programm mit JDBC Aufruf

Folgendes Programm baut eine JDBC Verbindung zu einer Oracle Datenbank auf und führt ein SELECT Statement aus.

Um das Programm kompilieren und ausführen zu können, muß der CLASSPATH die Datei classes111.zip enthalten¹⁶. Diese befindet sich i.d.R. im Verzeichnis \orant\jdbc\lib.

```
import java.sql.*;
import java.math.*;

public class JDBCQuery {
    // URL definiert benutzen Treiber und Datenbank
    // Typ 4
    private static String URL =
        "jdbc:oracle:thin:@127.0.0.1:1521:orcl";
    // Typ 2, DLLs!
    //private static final String URL = "jdbc:oracle:oci8:@orcl";

    static {
        try {
            // Treiber anmelden.
            DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Connection conn = null;

        try {
            // Datenbankverbindung öffnen.
            if (args.length == 0) {
                // Öffne Verbindung mit default id/passwort
                conn = DriverManager.getConnection(URL);
            }
            else if (args.length == 2) {
                String user = args[0];
                String pass = args[1];
                // Öffne Verbindung mit gegebener id/password.
                conn = DriverManager.getConnection(URL, user, pass);
            }
            else {
                System.out.println("\nAufruf mit: java JDBCQuery
[username password]\n");
                System.exit(0);
            }

            // Anfrage losschicken.
            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery("SELECT * FROM
train.dtypes");

            // Ergebnis auswerten.
            while (rset.next()) {
                System.out.print(rset.getInt("id")+", ");
                System.out.println(rset.getString("descr"));
            }

            // Ressourcen von stmt freigeben, rset wird damit auch
geschlossen
            stmt.close();
        }
    }
}
```

¹⁶ Dies gilt für Oracle 8.0.5.0. Für andere Versionen kann die Datei anders heißen. Konsultieren Sie dazu bitte die Dokumentation der verwendeten Datenbank.

```

        // Verbindung freigeben
        conn.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

E.3.2 Servlet mit JDBC Aufruf

Der oben vorgestellte Code kann, mit geringen Modifikationen, auch in einem Servlet benutzt werden.

Es wurden folgende Änderungen vorgenommen:

- Der JDBC-Treiber wird in der `init()` Methode des Servlets registriert.
- Die Benutzer-ID und das Paßwort für die Anmeldung bei der Datenbank können nicht von der Standardeingabe gelesen werden. Sie wurden deshalb in entsprechenden Strings gespeichert. Dies ist unflexibel; eine Verbesserung wird im nächsten Abschnitt vorgestellt.
- Das Ergebnis der Abfrage wird in einen `StringBuffer` geschrieben und dann an die `doGet` Methode zurückgegeben.
- Im Gegensatz zum Java Programm aus dem vorigen Abschnitt endet das Servlet nicht sofort, sondern nimmt so lange Anfragen entgegen, bis es vom Server entladen wird. Deshalb muß sichergestellt werden, daß jede geöffnete `Connection` auf jeden Fall geschlossen wird. Der entsprechende Befehl steht zu diesem Zweck in einem **finally** Block.

Das Servlet wird durch folgenden Code erzeugt:

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
// Packages für JDBC
import java.sql.*;
import java.math.*;

/** JDBC-Abfrage über ein Servlet */
public class JDBCQueryServlet extends HttpServlet {
    // Konfigurationsvariablen für JDBC (unflexibel)
    private String URL = "jdbc:oracle:thin:@127.0.0.1:1521:orcl";
    private String USER = "TRAIN";
    private String PASS = "train";
    private String lineSep = System.getProperty("line.separator");

    /** Erzeugt html Seite mit dem Ergebnis der Abfrage. */
    public void doGet(HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException, IOException
    {
        PrintWriter pw = res.getWriter();

        // Mime-Typ setzen und Zwischenspeicherung ausschalten.
        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setDateHeader("Expires", 0);
        res.setHeader("Cache-Control", "no-cache");

        pw.println("<html>");
        pw.println("<title>JDBC Query Demo</title>");
        pw.println("<body>");
        pw.println("<h2>JDBC Query Demo</h2><p>");
        pw.println(sampleQuery());
        pw.println("</p></body>");
        pw.println("</html>");
    }
}

```

```

}
/**
 * Wird beim instanziiiren ausgeführt.
 * Registriert den Treiber.
 */
public void init()
throws ServletException {
    try {
        // Treiber anmelden.
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver
());
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
/** Führt eine SQL Abfrage durch.
 * Liefert das Ergebnis oder ein Fehlermeldung zurück (als String).
 */
public String sampleQuery() {
    StringBuffer strBuff = new StringBuffer();
    Connection conn = null;
    boolean success = false;

    try {
        // Öffnet Verbindung mit vordefinierten Parametern
        conn = DriverManager.getConnection(URL, USER, PASS);

        // Anfrage losschicken
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery("SELECT * FROM
train.dtypes");

        // Ergebnis auswerten.
        while (rset.next()) {
            // Verwendet StringBuffer, weil schneller
            strBuff.append(rset.getInt("id"));
            strBuff.append(", ");
            strBuff.append(rset.getString("descr"));
            strBuff.append("<br>").append(lineSep);
        }
        success = true;

        // Statement und ResultSet schliessen.
        stmt.close();
    }
    catch (Exception e) {
        // Wird in die Log-Datei geschrieben
        System.out.println("Get connection, process, or close statement
exception: "
            + e.getMessage());
        e.printStackTrace();
    }
    finally { // Garantiert das Verbindung geschlossen wird
        // Verbindung schliessen
        if (conn != null)
            try {
                conn.close();
            }
            catch (Exception e) {
                System.out.println("Close connection exception: " +
e.getMessage());
                e.printStackTrace();
            }
    }
    if (success)
        return strBuff.toString();
    else
        return "DB Error!";
}
}

```

E.3.2.1 Konfigurationsvariablen in externe Dateien verlagern

In Abschnitt E.3.2 wurden die Konfigurationsvariablen für die Datenbankverbindung, nämlich die URL, Benutzer-ID und Paßwort, in das Servlet codiert. Wenn sich diese ändern, muß der Quellcode vorhanden sein und das Servlet neu kompiliert werden.

Dieses Problem kann, mit Hilfe des ResourceBundle Objektes, gelöst werden. Es erlaubt Paare der Form *Wert_Name = Wert_String* in einer Datei abzulegen. Die Daten werden dann zur Laufzeit eingelesen und den Variablen zugewiesen.

Folgender Code erzeugt ein ResourceBundle Objekt, welches die Datei "JDBCQueryServlet.properties" öffnet und die dort abgelegten Werte, "db_url", "db_user" und "db_pass", einliest.

- Über die Methode `getBundle(baseName)` wird die property-Datei geöffnet.
- Über die Methode `getString(Wert_Name)` wird der in der Datei abgelegter Wert (ein String) eingelesen.

```
private static java.util.ResourceBundle resJDBC2QueryServlet =
    java.util.ResourceBundle.getBundle("JDBC2QueryServlet");

private String URL = resJDBC2QueryServlet.getString("db_url");
private String USER = resJDBC2QueryServlet.getString("db_user");
private String PASS = resJDBC2QueryServlet.getString("db_pass");
private String lineSep = System.getProperty("line.separator");
```

Die Property-Datei mit den ausgelagerten Zeichenketten, muß sich im gleichen Verzeichnis wie die .class-Datei des Servlets befinden, welches diese Datei einliest. Wenn das Servlet mit dem „IBM Websphere Test Environment“ unter Visual Age ausgeführt wird, muß die Datei im Verzeichnis `"\IBM\Java\IDE\project_resources\IBM Websphere Test Environment"` abgelegt werden.

Die im Beispiel benutzte Datei "JDBCQueryServlet.properties", enthält folgende Daten:

```
# Properties für JDBCQueryServlet
db_url = jdbc:oracle:thin:@127.0.0.1:1521:orcl
db_user = TRAIN
db_pass = train
```

Unter Visual Age kann dieser Prozeß automatisiert werden, in dem Sie den Befehl **Zeichenfolgen auslagern** benutzen. Sie finden den Befehl, wenn Sie mit der Maus über eine Klasse fahren und mit dem rechten Knopf das Kontext-Menü öffnen. Wenn Sie den Befehl ausführen, erscheint dieses Fenster:

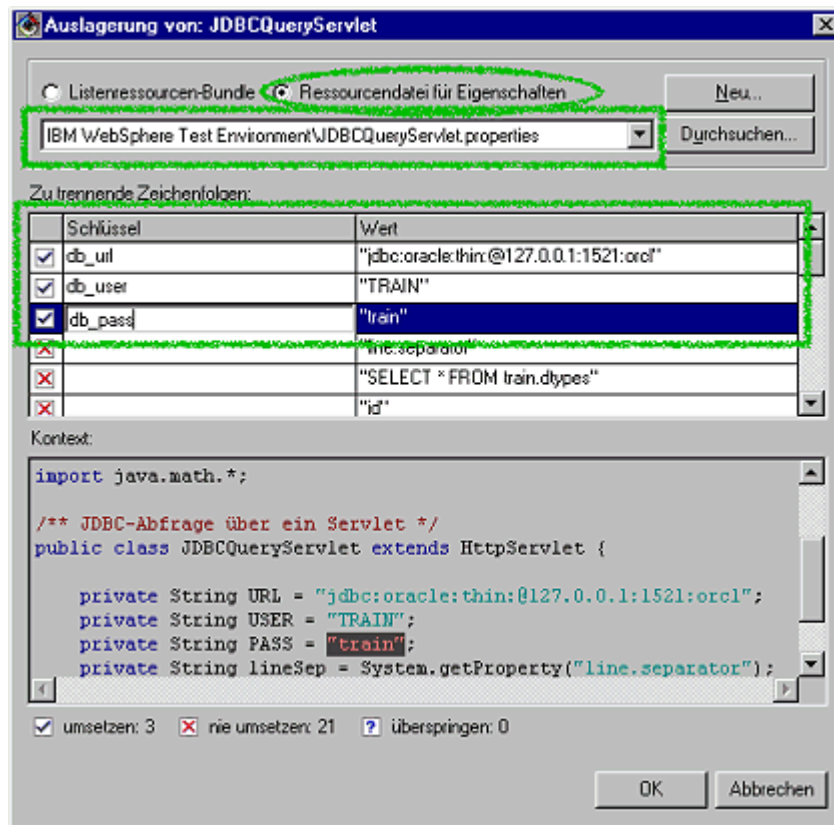


Abbildung 15: Auslagerung von Zeichenketten

Über die Schaltfläche **Neu** können Sie die zu erzeugende Property-Datei angeben. In der Auswahlliste, im mittleren Bereich des Fensters, können Sie die auszulagernden Zeichenketten mit markieren. Zeichenketten, die nicht ausgelagert werden sollen, werden mit einem Kreuz markiert.

E.3.3 Servlet mit Datenquellen Aufruf

Folgendes Beispielservlet führt die eben genannten Schritte durch, um über einen Verbindungs-Pool auf die Datenbank zuzugreifen:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

// Schritt 1: Erforderliche Pakete laden
// Pakete für JDBC
import java.sql.*;
import java.math.*;
// Pakete für JDBC Extensions von IBM und Naming Service
import com.ibm.db2.jdbc.app.stdext.javax.sql.*;
import com.ibm.ejs.dbm.jdbcext.*;
import javax.naming.*;

/** JDBC-Abfrage mit Verbindungs-Pool über ein Servlet */
public class DBQueryServlet extends HttpServlet {
    // Konfigurationsvariablen für JDBC (unflexibel)
    private String USER = "TRAIN";
    private String PASS = "train";
    // Zeichenfolge für die identifizierung des DataSource Objektes
    private String SOURCE = "jdbc/Training";

    private String lineSep = System.getProperty("line.separator");
    // DatenQuelle Objekt
    private DataSource ds = null;

    /** Erzeugt html Seite mit dem Ergebnis der Abfrage. */
```



```

public void doGet(HttpServletRequest req,
                  HttpServletResponse res)
throws ServletException, IOException
{
    PrintWriter pw = res.getWriter();

    // Mime-Typ setzen und Zwischenspeicherung ausschalten.
    res.setContentType("text/html");
    res.setHeader("Pragma", "no-cache");
    res.setDateHeader("Expires", 0);
    res.setHeader("Cache-Control", "no-cache");

    pw.println("<html>");
    pw.println("<title>JDBC Query Demo</title>");
    pw.println("<body>");
    pw.println("<h2>JDBC Query Demo</h2><p>");
    pw.println(sampleQuery());
    pw.println("</p></body>");
    pw.println("</html>");
}
/**
 * Wird beim Instanzieren ausgeführt.
 * Registriert den Treiber
 */
public void init()
throws ServletException {
    try {
        // Schritt 2: Hashtable für Namens-Service Parameter erstellen
        Hashtable parms = new Hashtable();
        // Schritt 3: Parameter laden
        parms.put(Context.INITIAL_CONTEXT_FACTORY,
                  "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        // Schritt 4: Namenskontext für Zugriff auf Namens-Service
        // erstellen
        Context ctx = new InitialContext(parms);
        // Step 5: Datenquelle über Namens-Service aufrufen
        ds = (DataSource)ctx.lookup(SOURCE);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
/** Führt eine SQL Abfrage über einen Verbindungs-Pool durch.
 * Liefert das Ergebnis oder eine Fehlermeldung zurück (als String).
 */
public String sampleQuery() {
    StringBuffer strBuff = new StringBuffer();
    Connection conn = null;
    boolean success = false;
    try {
        // Schritt 6: Verbindung aus dem Pool bekommen
        conn = ds.getConnection(USER, PASS);

        // Schritt 7: Anfrage losschicken und auswerten
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery("SELECT * FROM
train.dtypes");

        while (rset.next()) {
            strBuff.append(rset.getInt("id"));
            strBuff.append(", ");
            strBuff.append(rset.getString("descr"));
            strBuff.append("<br>").append(lineSep);
        }
        success = true;

        // Statement und ResultSet schließen.
        stmt.close();
    }
    catch (Exception e) {
        System.out.println("Get connection, process, or close statement
" + "exception: " + e.getMessage());
        e.printStackTrace();
    }
}

```

```

    }
    finally {
        // Schritt 8: Verbindung an den Pool zurückgeben
        if (conn != null) {
            try {
                conn.close();
            }
            catch (Exception e) {
                System.out.println("Close connection exception: " +
e.getMessage());
                e.printStackTrace();
            }
        }
    }

    if (success)
        return strBuff.toString();
    else
        return "DB Error!";
}
}

```

E.4 Servlet mit HttpSession Objekt

In diesem Abschnitt wird ein Servlet vorgestellt, welches das HttpSession Objekt benutzt, um in ihm die Anzahl der Seitenzugriffe jedes einzelnen Clients zu speichern. Es zeigt dann diese Information, zusammen mit der Gesamtanzahl der Seitenzugriffe und einigen Daten über die Session, an.

Anzahl der Aufrufe dieses Servlets:

Seit das Servlet geladen wurde ist es **8** mal aufgerufen worden.

Sie haben dieses Servlet:

2 mal aufgerufen.

Session Daten

Ihre session-id ist: **4BT40DQAAAAAM5YAAAAUTAA**
 Ihre Session wurde zu diesem Zeitpunkt erzeugt:
 957516063126 (Fri May 05 10:41:04 CEST 2000)

Falls cookies deaktiviert sind kann dieser [Link](#) benutzt werden.

Abbildung 16: Ausgabe von CallCounterServlet

Um die Anzahl der Seitenzugriffe jedes Clients zu speichern, wird in der Session folgendes, einfaches Bean hinterlegt:

```

package mybeans;

import java.io.*;

/** Einfaches Zähler Bean */
public class UserCallCounter implements Serializable {
    int called; // Zählervariable

    /** Konstruktor */
    public UserCallCounter() { called = 0; }

    /** Liefert Wert der Zählervariable zurück */

```

```

public int getCalled() { return called; }

/** Erhöht die Zählervariable um 1 */
public void incCalled() { ++called; }

/** Setzt die Zählervariable auf "newCalled" */
public void setCalled(int newCalled) { called = newCalled; }
}

```

Im nächsten Paragraph steht der Code für das zugehörige Servlet.

Es enthält auch ein Beispiel für URL-Umschreibung, denn es erzeugt auch einen Link auf sich selbst. Die geschieht über die `encodeURL(...)` Methode des Response Objekts. Über die Methode `getRequestURI()` kann die aktuelle Adresse des Servlets ermittelt werden. Der Link wird nur das gewünschte Ergebnis produzieren, wenn URL-Rewriting in WebSphere aktiviert ist.

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import mybeans.*;

public class CallCounterServlet extends HttpServlet {
    private int calledCount = 0; // Anzahl der Seitenzugriffe

    /** Demonstration des Session Objektes */
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException
    {
        PrintWriter pw = res.getWriter();

        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setDateHeader("Expires", 0);
        res.setHeader("Cache-Control", "no-cache");

        // Erzeugt bzw. hole Session Objekt
        HttpSession session = req.getSession(true);
        // Falls neu schreibe ein UserCallCounter Bean in die Session
        if (session.isNew() || session.getValue("userCount") == null)
            session.putValue("userCount", new UserCallCounter());

        // Hole UserCallCounter Bean aus der Session
        UserCallCounter ucc = (UserCallCounter)session.getValue(
            "userCount");
        ++calledCount;
        ucc.incCalled();

        pw.println("<html><title>UserSessionCounter
Servlet</title><body>");
        pw.println("<h3>Anzahl der Aufrufe dieses Servlets: </h3>");
        pw.println("Seit das Servlet geladen wurde ist es <b>");
        pw.println(calledCount+" </b>mal aufgerufen worden.<br>");

        pw.println("<h3>Sie haben dieses Servlet: </h3><b>" + ucc.getCalled
());
        pw.println("</b> mal aufgerufen.<br>");

        // Session ID ausdrucken
        pw.println("<h3>Session Daten</h3>Ihre session-id ist:<b>
"+session.getId());

        // Zeitpunkt der Erzeugung ausdrucken
        pw.println("</b><br>Ihre Session wurde zu diesem Zeitpunkt
erzeugt:<br> ");
        pw.println(session.getCreationTime());
        pw.println("(" + new Date(session.getLastAccessedTime()) + ")");

        // Alternative zu cookies: URL Umschreibung.
        pw.println("<p>Falls cookies deaktiviert sind kann dieser");

```

```
    pw.println("<a href=\"" + res.encodeURL(req.getRequestURI()) +  
    "\">Link</a>");  
    pw.println("benutzt werden.</body></html>");  
}  
}
```

Übungen – Allgemeine Vorbereitungen

Bevor Sie beginnen, folgen Sie der Demonstration, um folgende Aufgaben zu erledigen:

- Den Dienst „**IBM WAS AdminServer**“ zu starten.
- Den Cache in Netscape zu deaktivieren.
- Die Funktionsfähigkeit von WebSphere zu überprüfen.
- In Visual Age for Java, die Elemente „**IBM JSP Execution Monitor**“ und „**IBM WebSphere Test Environment**“ in die Workbench einzubinden.
- Das Projekt „**Servlet Training**“ zu erstellen bzw. zu laden.

1. Übung: Seitenzugriffszähler

1.1. Aufgabenstellung

Erstellen Sie ein kleines Servlet, welches die Anzahl der Zugriffe auf sich selbst zählt. Es soll eine HTML Seite erzeugen und die Anzahl der Zugriffe dort anzeigen.

Die Ausgabe könnte z.B. so aussehen:

Hit Counter Servlet

Dieses Servlet wurde 2 mal aufgerufen.

1.2 Hinweise

Die Programmierung läßt sich in folgende Teilschritte aufteilen:

1. Ein neues Servlet anlegen:
 - Erstellen Sie eine neue Klasse, die **javax.servlet.http.HttpServlet** erweitert.
 - Importieren Sie die Pakete **javax.servlet**, **javax.servlet.http** und **java.io**. Diese sind für jedes Servlet notwendig.
2. Die benötigte Instanzvariable anlegen:
 - Ein **int**, zum Speichern der Seitenzugriffe.
3. Die Methode:

public void doGet(HttpServletRequest req, HttpServletResponse res)
anlegen.

In dieser Methode sind folgende Aufgaben zu erledigen:

- Die Zählervariable erhöhen.
- Den Inhalt der Antwort auf „**text/html**“ setzen.
- Den **PrintWriter** des Response Objekts holen.
- Den Code für die HTML-Seite in den **PrintWriter** der Response schreiben.

1.3 Ausführen

Um das Servlet auszuführen, fahren Sie mit der Maus über die entsprechende Klasse und drücken Sie die rechte Taste. Wählen Sie dann **Tools -> Servlet Launcher -> Starten**. Falls anschließend ein Dialog-Fenster erscheint, drücken Sie ENTER.

1.4 Optionale Aufgaben

1.4.1 Cache Abschalten

Verhindern Sie das Zwischenspeichern der Seite um sicherzustellen, daß der Client immer die aktuelle Fassung anzeigt.

Dies geschieht mit folgenden Befehlen.

```
res.setHeader("Pragma", "no-cache");  
res.setDateHeader("Expires", 0);  
res.setHeader("Cache-Control", "no-cache");
```

Das Zwischenspeichern wird verhindert, indem die Informationen, die zu diesem Zweck durch die HTML Spezifikation definiert sind, in den HEAD Teil der zu erzeugenden Seite geschrieben werden. Der HEAD Teil enthält Meta-Informationen über die Seite.

1.4.2 Start-Zeit anzeigen.

Zusätzlich soll die Start-Zeit, also der Zeitpunkt des ersten Aufrufs des Servlets, angezeigt werden. Dazu sind folgende Schritte notwendig:

- Das Paket **java.util.GregorianCalendar** muß importiert werden.
- Folgende Instanzvariable muß erstellt werden: Ein **String**, zum Speichern der Startzeit.
- Die **public void init()** Methode muß angelegt werden.

In dieser Methode wird ein **GregorianCalendar** Objekt erzeugt. Über die Methode **get(int field)** können bestimmte Zeitinformationen ausgelesen werden. Verwenden Sie für *field* die Konstanten **Calendar.MONTH**, **Calendar.DATE**, **Calendar.HOUR** und **Calendar.MINUTE**.

Erstellen Sie einen String, in dem Datum und Uhrzeit des Aufrufs gespeichert werden und schreiben Sie diesen in die HTML-Seite.

2. Übung: Formular auslesen

2.1 Vorbereitung

Folgen Sie der Demonstration des Referenten, um die Datei **myForm.html** in das Verzeichnis „web“ des „WebSphere Test Environment“ zu kopieren.

2.2 Aufgabenstellung

Erstellen Sie ein Servlet, mit dem Namen **MyFormReaderServlet**, welches die Informationen ausliest, die es von dem Formular der HTML-Seite **myForm.html** übermittelt bekommt. Es soll eine Seite zurückliefern, in der diese Informationen angezeigt werden.

Folgende Abbildung zeigt die HTML-Seite mit dem Formular und die Namen der darin enthaltenen Felder. Über diese Namen, kann das Servlet auf die Daten der Felder zugreifen.

Informationsmaterial anfordern:		Namen der Parameter-Felder:
Name:	<input type="text"/>	feldName
Straße:	<input type="text"/>	feldStrasse
PLZ, Ort:	<input type="text"/>	feldOrt
	<input type="checkbox"/> Bitte senden Sie mir keine Werbung !	checkWerbung
Dokument Kategorie:	<input type="text" value="Geschäftsbericht"/> <input type="button" value="Weiter"/>	listDokumentTyp

Die Ausgabe des Servlets könnte z.B. so aussehen:

Formularabfrage - Daten anzeigen

Name: Max Mustermann
 Straße: Musterstraße 10
 PLZ, Ort: 12345, Musterstadt
 Dokument Typ: 300

2.3 Hinweise

Zur Lösung der Aufgabe sind folgende Einzelschritte notwendig:

- Neue Klasse (Servlet) mit dem Namen **MyFormReaderServlet** erzeugen.
- Eine Methode mit Namen:
public void doPost(HttpServletRequest req, HttpServletResponse res)
 erzeugen.

In dieser Methode sind folgende Aufgaben zu erledigen:

- Seiteninhalt auf **text/html** setzen.
- Variablen für die Speicherung der übermittelten Daten anlegen:
Es werden vier Strings benötigt um die Inhalte der Textfelder und der Liste aufzunehmen.
- Die Parameter aus dem Formular auslesen und in die Strings stecken.
- Über das `PrintWriter` Objekt der `HttpServletResponse`, die Ausgabe (HTML-Seite) erzeugen.

2.4 Ausführen - Testen

Um Ihr Servlet zu auszuführen, starten Sie zuerst die WebSphere Test-Umgebung, über das Menü **Arbeitsbereich -> Tools -> WebSphere Test Environment Starten**.

Öffnen Sie dann Netscape und geben Sie folgende URL an:

<http://127.0.0.1:8080/myForm.html>

2.5 Optionale Aufgaben

Wenn Sie wollen, können Sie folgende optionale Elemente einbauen:

- Die Zwischenspeicherung der Seite verhindern.
- Eine boolean Variable für den Parameter *checkWerbung* anlegen. Sie sollte auf den Wert **true** gesetzt werden, falls *checkWerbung* den Wert **null** hat (d.h. der Kasten auf der Seite enthält kein ✓).
- Die Daten aus dem Feld **listDocumentTyp** enthalten eine Zahl, codiert als String. Wandeln Sie diesen String in ein **int** Typ um.

Dazu müssen Sie zuerst eine Variable vom Typ **int** anlegen. Dann müssen Sie ein Integer Objekt erzeugen. Über die **intValue()** Methode des Integer Objektes, können Sie den Wert als **int** Typ erhalten.

Bei der Erzeugung des Integers kann eine **NumberFormatException** ausgelöst werden, die Abgefangen werden muß.

Beispiel:

```
int zahl;
try {
    zahl = new Integer(einStringMitEinzeZahl).intValue();
}
catch (NumberFormatException e) { zahl = -1; }
```

3. Übung: JSP-Counter

3.1 Aufgabenstellung

Es soll eine JSP erstellt werden, welches, ähnlich wie in Aufgabe 1, die Anzahl der Zugriffe auf sich selbst zählt. Diese Information soll in die HTML-Seite geschrieben werden.

3.2 Hinweise

Folgende Einzelschritte sind notwendig:

- Die Direktive **page** benutzen um die Sprache der Seite und den Inhalts-Typ der Seite zu spezifizieren.
- Ein Deklarations-Block anlegen. In ihm muß, für den Zähler, ein **int** angelegt werden.
- Den Zähler erhöhen.
- Das Ergebnis in die HTML Seite schreiben.

4. Übung: Datenanzeige mit JDBC

4.1 Vorbereitung

Folgen Sie der Demonstration des Referenten, um die Datei `MyJDBCQuery.java` zu importieren.

4.2 Aufgabenstellung

Erweitern Sie die **main**-Methode der Klasse so, daß eine einfache Datenbankabfrage über JDBC stattfindet. Die Datenbank ist auf der nächsten Seite dargestellt. Die Ergebnisse sollen nach **System.out** geschrieben werden. Benutzen Sie für die Abfrage den SQL-Befehl:
"SELECT * from TRAIN.DTYPES"

Folgende Schritte sind notwendig:

- Connection holen.
- Statement erstellen.
- ResultSet erstellen.
- Ergebnisse abrufen und anzeigen.
- Statement und Verbindung schließen.

Die Befehle können ein **SQLException** auslösen und müssen deshalb innerhalb eines **catch-try** Blocks geschrieben werden.

4.3 Die Datenbank

Schema: TRAIN

Tabelle: DTYPES

Name	Datentyp	Länge	Nullwert erlaubt
ID	NUMBER		Nein
DESCR	VARCHAR	50	Nein

Beispieleinträge:

100, Aktionärsbriefe

300, Geschäftsbericht

Schema: TRAIN

Tabelle: DDESCR

Name	Datentyp	Länge	Nullwert erlaubt
ID	NUMBER		Nein
TYPE	NUMBER		Nein
TITLE	VARCHAR	100	Nein
FILETYPE	VARCHAR	4	Ja
SIZEKB	NUMBER	5	Ja

Fremdschlüssel: TYPE; referenziert DTYPES(ID)

Beispieleinträge:

10, 100, Aktionärsbrief Jan.-Sep. 1997, PDF, 140

41, 300, Bericht des Aufsichtsrats, PDF, 92

5. Übung: Servlet mit JDBC Datenbankzugriff

5.1 Aufgabenstellung

Die Datenanzeige aus Aufgabe 2 soll nun erweitert werden. Es soll mit dem Wert von `listDokumentTyp` eine Datenbankabfrage durchgeführt werden und die entsprechenden Einträge aus der Datenbank angezeigt werden.

5.2 Hinweise

Folgende Teilschritte sind vor der Abfrage notwendig:

- Das Paket `java.sql` importieren.
- Eine `init`-Methode erstellen um den Treiber zu registrieren.
- Den Wert von `listDokumentTyp` in ein `int` Konvertieren.

Dazu müssen Sie zuerst eine Variable vom Typ `int` anlegen. Dann müssen Sie ein Integer Objekt erzeugen. Über die `intValue()` Methode des Integer Objektes, können Sie den Wert als `int` Typ erhalten.

Bei der Erzeugung des Integers kann eine `NumberFormatException` ausgelöst werden, die Abgefangen werden muß.

Beispiel:

```
int zahl;
try {
    zahl = new Integer(einStringMitEinzeZahl).intValue();
}
catch (NumberFormatException e) { zahl = -1; }
```

- Die `doPost` Methode um die Datenbankabfrage erweitern. Diese erfolgt ähnlich wie in Aufgabe 5. Es ist jedoch folgender SQL Befehl notwendig:

```
"SELECT * FROM train.ddescr WHERE type = "+documentTypAlsInt
```

- Den Befehl auswerten und die Ergebnisse in das `PrintWriter` Objekt schreiben.

5.3 Optionale Aufgabe

Schließen Sie die Verbindung in einem `finally` Block, und nur wenn diese nicht `null` ist. Dies garantiert, daß bei Problemen mit der Datenbankverbindung, keine Verbindungen offen bleiben.

6. Übung: HttpSession

6.1 Aufgabenstellung

Das Servlet aus Aufgabe 1, soll nun so erweitert werden, daß es zusätzlich die Anfragen von jedem Client zählt.

6.2 Hinweise

Folgende Teilschritte sind notwendig:

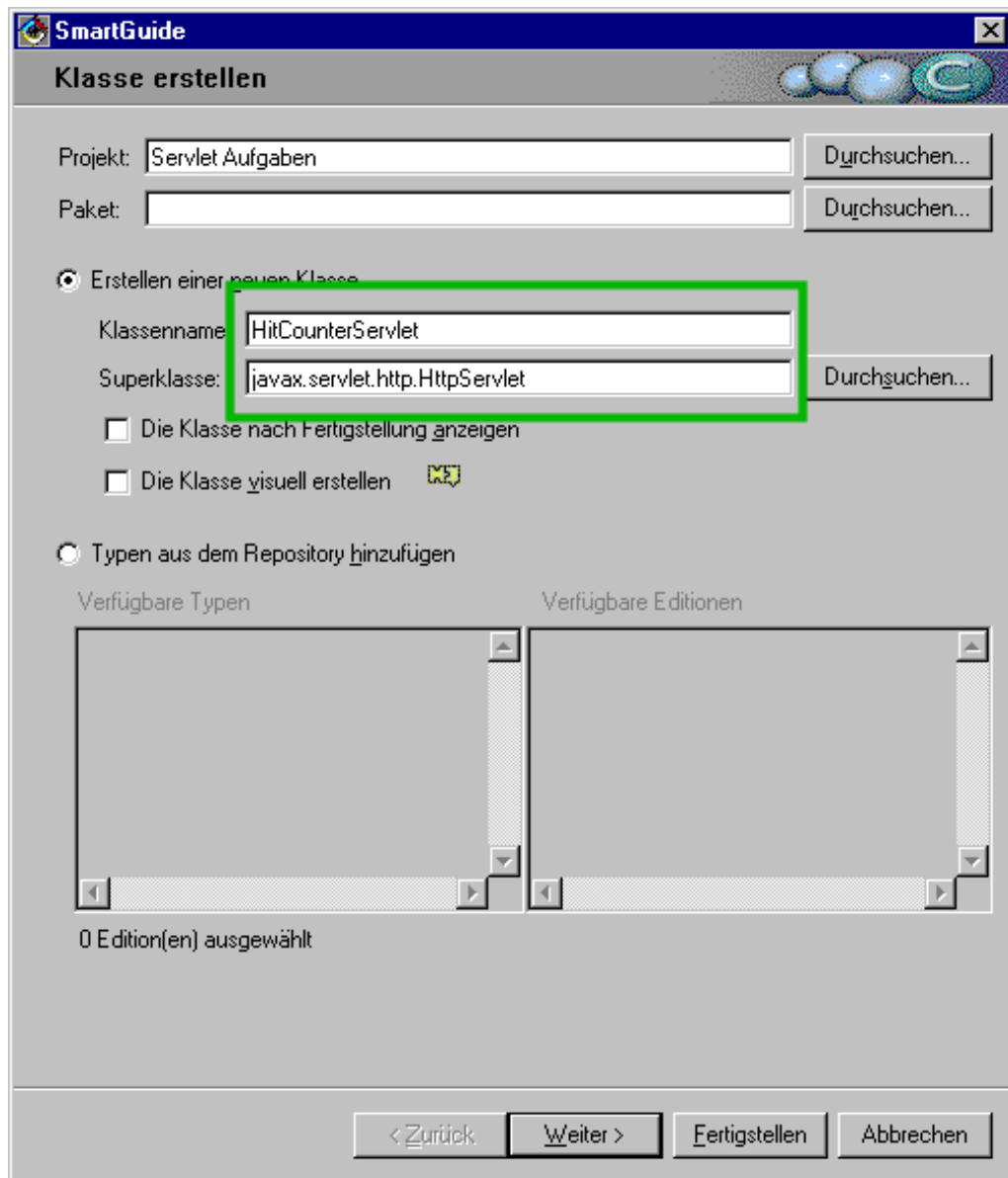
- Die Klasse **mybeans.UserCallCounter** importieren.
- Ein **HttpSession** Objekt aus dem Request Objekt holen.
- Überprüfen ob das HttpSession Objekt neu ist. Falls **ja**, ein neues UserCallCounter Objekt hineinschreiben.
- Das UserCallCounter Objekt aus der Session herauslesen und seinen Zähler, über seine `incCalled()` Methode, erhöhen.
- Die Daten ausgeben.

L. Lösungen

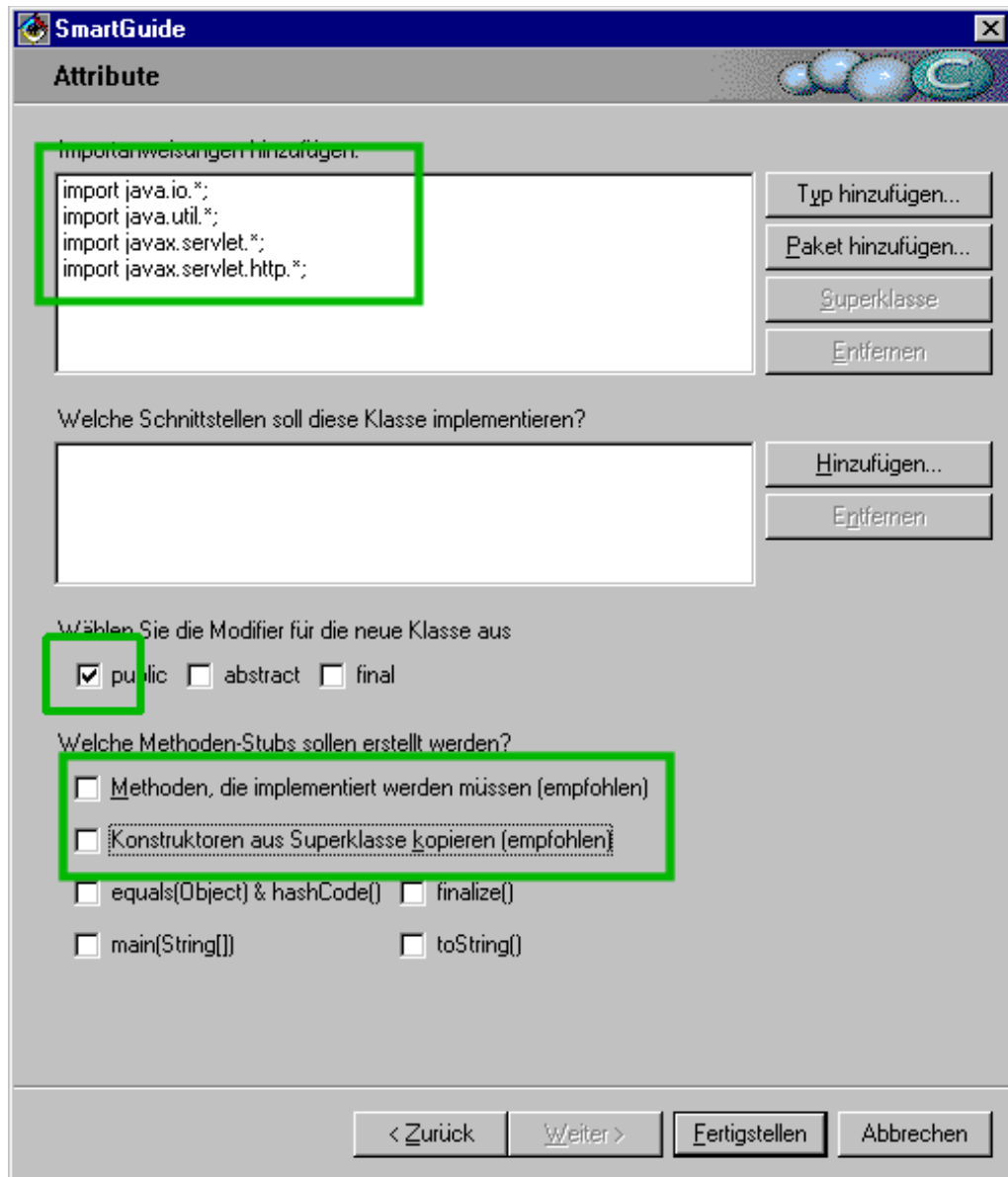
Lösung der Übung 1

Schritt 1: Erstellen des Servlet Objektes.

Dialog 1 von 2:



Dialog 2 von 2:



Schritt 2: Instanzvariablen in die Klasse schreiben

Das Ergebnis sieht so aus:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HitCounterServlet extends
    javax.servlet.http.HttpServlet {
    private int hitCounter;
}
```

Schritt 3: Die Methode doGet(...) erzeugen

```
/**
 * Verarbeitet HTTP GET Anforderungen.
 */
public void doGet(HttpServletRequest req, HttpServletResponse
res)
```



```

throws ServletException, IOException {
    // Inhalts-Typ der Response setzen. Immer dann notwendig
    // wenn der PrintWriter der ServletResponse benutzt wird.
    res.setContentType("text/html");

    // Caching ausschalten (für die Seite die erzeugt wird)
    // Verhindert das eine, moeglicherweise im Cache
    vorhandene,
    // ältere Seite angezeigt wird.
    res.setHeader("Pragma", "no-cache");
    res.setDateHeader("Expires", 0);
    res.setHeader("Cache-Control", "no-cache");

    // Erhöhe Anzahl der Seitenzugriffe
    hitCounter++;

    // Hole PrintWriter der ServletResponse
    PrintWriter out = res.getWriter();
    // Erzeuge HTML Seite mit Antwort
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Hit Counter Servlet</title>");
    out.println("</head>");

    // Body
    out.println("<body>");
    out.println("<h2>Hit Counter Servlet</h2>");
    // Eine Instzvariable abfragen
    out.println("Dieses Servlet wurde <b>"+hitCounter);
    out.println("</b> mal aufgerufen.</p>");

    out.println("</body></html>");
}

```

Lösung der Übung 2

Code des HTML-Formulars

Folgender Code erzeugt die HTML-Seite, welche von dem Servlet (der Lösung) ausgewertet wird. Die Seite wird vor Beginn der Aufgabe zur Verfügung gestellt und wird hier zum besseren Verständnis der Lösung abgebildet.

```

<html>
<head>
<title>Informationsmaterial anfordern</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
</head>

<body bgcolor="#FFFFCC">
<h2>Informationsmaterial anfordern:</h2>
<form action = "servlet/MyFormReaderServlet" method="POST">
<table width="70%" border="0">
  <tr>
    <td width="24%">Name:</td>
    <td width="76%">
      <input type="text" name="feldName" size="30">
    </td>
  </tr>
  <tr>
    <td width="24%">Straße:</td>
    <td width="76%">
      <input type="text" name="feldStrasse" size="30">
    </td>
  </tr>
  <tr>
    <td width="24%">PLZ, Ort:</td>
    <td width="76%">
      <input type="text" name="feldOrt" size="30">
    </td>
  </tr>
  <tr>
    <td width="24%" height="2">&nbsp;</td>
    <td width="76%" height="2"><br>
      <input type="checkbox" name="checkWerbung" value="NEIN">
      Bitte senden Sie mir keine Werbung !</td>
  </tr>
  <tr>
    <td width="24%">&nbsp;</td>
    <td width="76%">&nbsp;</td>
  </tr>
  <tr>
    <td width="24%">Dokument Kategorie:</td>
    <td width="76%">
      <select name="listDocumentTyp" size="1">
        <option value="300">Geschäftsbericht</option>
        <option value="100">Aktionsbriefe</option>
        <option value="90">Investor Relations</option>
        <option value="80">Währungsmanagement</option>
      </select>
      <input type="submit" name="button" value="Weiter">
    </td>
  </tr>
  <tr>
    <td width="24%">&nbsp;</td>
    <td width="76%">&nbsp;</td>
  </tr>
</table>
</form>

```

```
</body>
</html>
```

Schritt 1: Erstellung der Klasse

Die Erstellung der Klasse `MyFormReaderServlet` erfolgt analog zu Schritt 1 der ersten Aufgabe.

Schritt 2: Methode `doPost(...)` Programmieren

Folgender Code wertet die Daten des Formulars aus:

```
/** Wertet die Daten aus myForm.html aus */
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyFormReaderServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse
res) throws IOException {
        PrintWriter out;
        String name, strasse, ort;
        boolean werbung;
        int documentTypID;

        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setDateHeader("Expires", 0);
        res.setHeader("Cache-Control", "no-cache");

        name = req.getParameter("feldName");
        strasse = req.getParameter("feldStrasse");
        ort = req.getParameter("feldOrt");

        // Aus einem String einen int erzeugen
        try {
            documentTypID = new Integer(req.getParameter
("listDokumentType")).intValue();
        }
        catch (NumberFormatException e) {
            documentTypID = -1;
        }

        // Wenn Box nicht ausgewählt (also Werbung erwünscht),
        // dann Wert == null
        if (req.getParameter("checkWerbung") == null)
            werbung = true;
        else
            werbung = false;

        // Ausgabe erzeugen
        out = res.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Formularabfrage - Daten
anzeigen</title>");
        out.println("</head>");

        // Body
        out.println("<body bgcolor=\"#FFFFCC\">");
        out.println("<h2>Formularabfrage - Daten anzeigen</h2>");
        // Daten ausgeben
        out.println("Name: " + name);
```

```
        out.println("<br>Strasse: " + strasse);
        out.println("<br>PLZ, Ort: " + ort);
        out.println("<br>Dokument Typ: " + documentTypeID);
        out.println("<br>Werbesendungen möglich: " +
werbung);
        out.println("</body></html>");
    }
}
```

Lösung der Übung 3

Folgender Code erstellt das JSP für Aufgabe 3:

```
<html>
<title>Counter JSP</title>

<body>
<!-- Directiven -->
<%@ page language = "java" %>
<%@ page contentType = "text/html" %>

<%! private int count = 0;

    private void increase() { ++count; }
    private int getCount() { return count; }
%>

<h2>Counter JSP</h2>
<p>
<% increase(); %>
Diese JSP wurde <%= getCount() %> mal aufgerufen.
</p>
<p>

</body>
</html>
```

Lösung der Übung 4

Bevor das Programm ausgeführt werden kann, müssen sich die entsprechenden JDBC Treiber im CLASSPATH Pfad befinden.

```
import java.sql.*;

class MyJDBCQuery {
    // URL definiert benutzen Treiber und Datenbank
    private static String URL =
"jdbc:oracle:thin:@127.0.0.1:1521:orcl"; // Typ 3
    private static String USER = "TRAIN";
    private static String PASS = "train";

    static {
        try {
            // Treiber anmelden.
            DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

/**
 * Startet die Anwendung.
 * @param args ein Array von Befehlszeilenargumenten
 */
public static void main(java.lang.String[] args) {
    try {
        Connection conn = DriverManager.getConnection(URL,
USER, PASS);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * from
TRAIN.DTYPES");

        while (rs.next()) {
            System.out.println(rs.getInt(1)+"",
"+rs.getString(2));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

Lösung der Übung 5

Folgender Code ist eine Lösung zu Aufgabe 5. Interessant ist, daß der **conn.close()** Befehl in einem **finally** Block ist, um auf jeden Fall ausgeführt zu werden. Die Verbindung zu schließen ist jedoch nur notwendig, wenn diese nicht den Wert **null** hat.

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Aufgabe 5 - Formular auslesen mit DB Abfrage
 *
 * Test Environment starten und Formular mit folgender URL
 aufrufen:
 * http://localhost:8080/myForm2.html
 */
public class FormReader2Servlet extends HttpServlet {
    private static String URL =
"jdbc:oracle:thin:@127.0.0.1:1521:orcl"; // Typ 3
    private static String USER = "TRAIN";
    private static String PASS = "train";

    /**
     * Datenbankabfrage
     */
    public void dbQuery(PrintWriter pw, int docTyp) {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(URL, USER,
PASS);
            Statement st = conn.createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM
train.ddescr WHERE type = "+docTyp);

            pw.println("<br><br>");

            while (rs.next()) {
                pw.println(rs.getInt("id") + ", ";
                pw.println(rs.getString("title")+"<br>");
            }

            } catch(SQLException e) {
                e.printStackTrace();
            } finally {
                if (conn != null) {
                    try { conn.close(); }
                    catch (Exception e) { e.printStackTrace(); }
                }
            }
        }
    }
    /**
     * Verarbeitet die POST Request des Formulars.
     */
    public void doPost(HttpServletRequest req, HttpServletResponse
res)
throws ServletException, IOException
    {
        PrintWriter out = null;
        String name, strasse, ort, dokumentTyp;
        boolean werbung;
    }
}
```

```

// Document-Typ setzen
res.setContentType("text/html");

// Parameter aus Formular auslesen
name = req.getParameter("feldName");
strasse = req.getParameter("feldStrasse");
ort = req.getParameter("feldOrt");
dokumentTyp = req.getParameter("listDokumentTyp");

// Optional - check-box auswerten
if (req.getParameter("checkWerbung")==null)
    werbung = true;
else
    werbung = false;

// Antwort generieren
out = res.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Formularabfrage - Daten
anzeigen</title>");
out.println("</head>");

// Body
out.println("<body bgcolor=\"#FFFFCC\">");
out.println("<h2>Formularabfrage - Daten anzeigen</h2>");

out.println("Name: "+name);
out.println("<br>Stra&szlig;e: "+strasse);
out.println("<br>PLZ, Ort: "+ort);
out.println("<br>Dokument Typ: "+dokumentTyp);
out.println("<br>Werbesendungen m&ouml;glich: "+werbung);

int dokInt = string2Int(dokumentTyp);
dbQuery(out, dokInt);

out.println("</body></html>");
}

// Treiber Registrieren
public void init() {
    try {
        DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
    }
    catch (Exception e) {
        System.err.println(e.getMessage());
        e.printStackTrace();
    }
}

// Convertiert einen String nach Int.
public int string2Int(String aString) {
    int i;
    try { i = new Integer(aString).intValue(); }
    catch (NumberFormatException e) {
        e.printStackTrace();
        i=-1;
    }
    return i;
}
}

```


Lösung der Übung 6

Dieser Programmcode zeigt eine Lösung der sechsten Aufgabe:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import mybeans.*;

public class SimpleCallCounterServlet extends HttpServlet {
    private int hitCounter = 0; // Anzahl der Seitenzugriffe
    /**
     * Demonstration des Session Objektes
     */
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");

        // Erzeugt bzw. hole Session Objekt
        HttpSession session = req.getSession(true);
        // Falls neu schreibe ein UserCallCounter Bean in die
        Session
        if (session.isNew() || session.getValue("yourCount") ==
        null)
            session.putValue("yourCount", new UserCallCounter());
        // Hole UserCallCounter Bean aus der Session
        UserCallCounter ucc = (UserCallCounter)session.getValue
        ("yourCount");
        ucc.incCalled();

        ++hitCounter;

        // Ausgabe
        PrintWriter pw = res.getWriter();
        pw.println("<html><title>UserSessionCounter
        Servlet</title><body>");
        pw.println("<h3>Anzahl der Aufrufe dieses Servlets:
        </h3>");
        pw.println("Seit das Servlet geladen wurde ist es
        <b>"+hitCounter);
        pw.println("</b>mal aufgerufen worden.<br>");

        pw.println("Sie haben dieses Servlet: <b>"+ucc.getCalled
        ());
        pw.println("</b> mal aufgerufen.<br>");

        pw.close();
    }
}
```