Java

Servlets, Java Server Pages (JSP)

Server-API-Ansätze

Ausführbare Programmeinheiten im Web-Server

- Entwickelt um Nachteile der CGI-Skripte zu überwinden
- Programtechnischer Ansatz: Anweisungen stehen nicht als Script-Anweisungen in HTML-Datei, sondern bilden eigenständige Anwendung
- Übersetztes Programm wird über eine definierte Schnittstelle (Container) im Server eingebunden

Erweiterungen werden in den Adressraum des Servers geladen

- Werden nur einmal geladen
- Werden in Threads statt Prozessen ausgeführt

Bekannteste Vertreter

- ASP.NET (Microsoft)
- Java Servlets (Sun)

Was ist ein Servlet?

Servlet (Server-Applet)

- Eine abgeleitete Java-Klasse
- Von einem Container verwaltet wird
- Generiert dynamisch HTML-Inhalte

Servlet-Engine (Container)

- Netzwerkschnittstelle
- Stell die Dienste zum Empfangen von Anfragen und Senden von Antworten bereit
- Enthält und verwaltet die Servlets über ihren gesamten Lebenszyklus

Servlets
Grundlagen

Servlets und Applets sind konzeptionell ähnlich

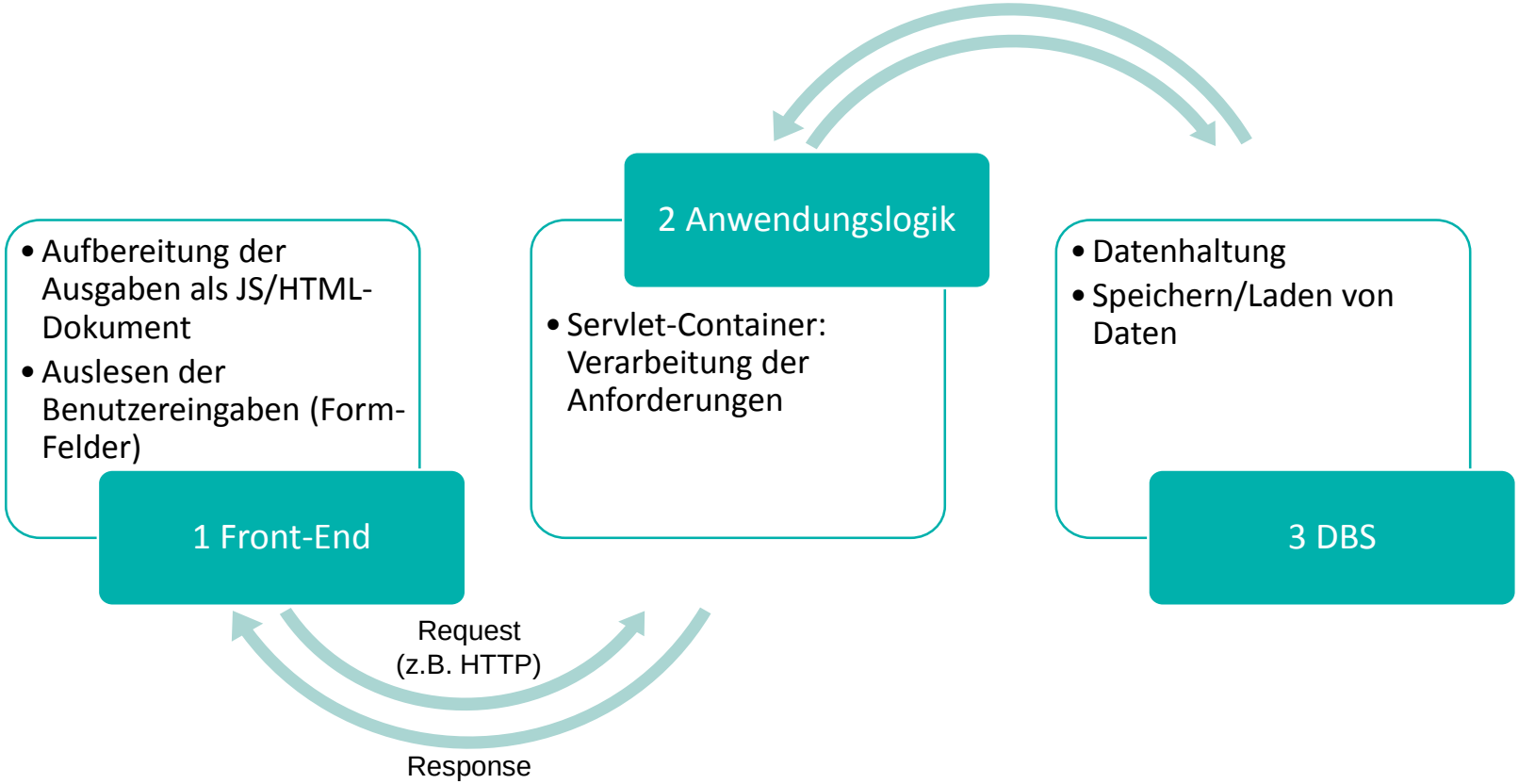
- Ein Applet wird von einem Container im Browser verwaltet
- Ein Servlet von einem Container im Web-Server
- Beide besitzen also keine main()-Methode und werden trotzdem ausgeführt

Programmtechnische Nutzung durch die Servlet-API

- Teil des Software Development Kits der Java Enterprise Edition (JEE)
- javax.servlet: Protokoll-unabhängige Klassen und Interfaces
- javax.servlet.http: http-spezifische Erweiterungen
- Steuerungsmöglichkeiten durch Annotationen

Ablauf einer HTTP-Anfrage (3-Tier-Architektur)

- Anmerkung: Es gibt auch nicht http-spezifische Servlets



Servlets
Vergleich der serverseitigen Konzepte

CGI
- Schnittstelle um externe Programme auszuführen
- Jede Anfrage erzeugt einen eigenen Prozess
- Sprach-/System-unabhängiges Konzept

PHP
- Aktive Anweisungen im HTML-Objekt
- Werden beim Einlesen vom HTTP-Server zur Laufzeit interpretiert
- Alle Anfragen besitzen jeweils eine eigene Instanz
- Session-Management-Funktionen

Servlet
- Dauerhafter Prozess der auch über Anfragen hinweg existent ist
- Mehrere Anfragen interagieren mit der gleichen Instanz

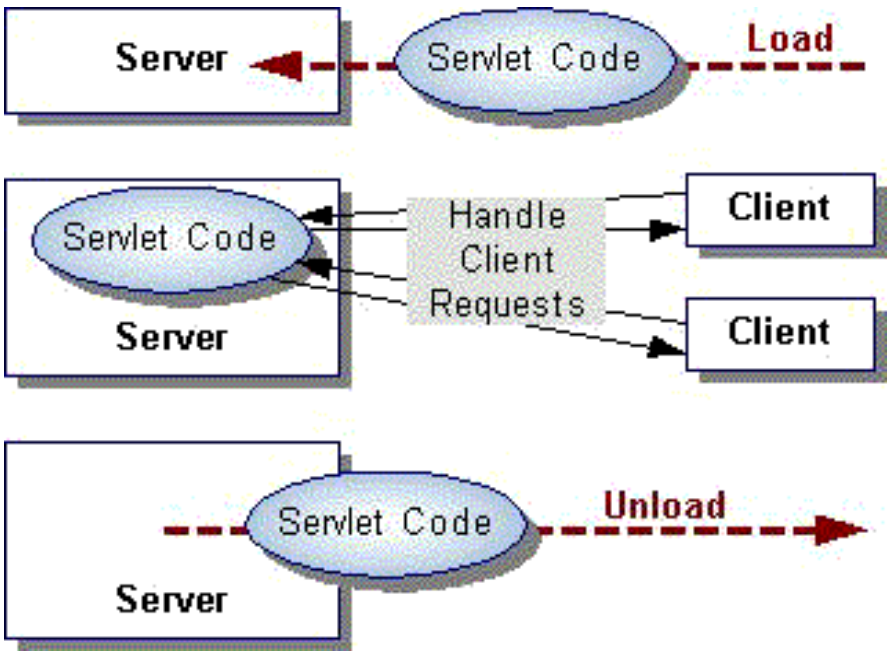
Servlets
Lebenszyklus

Ein Servlet unterliegt einem genau festgelegten Lebenszyklus

1. Laden der Servlet-Klasse und instanziieren (on demand)
2. Initialisieren des Servlet-Objekts
3. Verarbeitung der verschiedenen Anforderungen
4. Entfernen des Servlet-Objekts
5. Entladen der Servlet-Klasse

- Dieser Lebenszyklus wird von einem Container verwaltet und bestimmt somit die programmtechnische Nutzung.
- Sie wird mit den Methoden *init*, *service* und *destroy* realisiert

Servlets
Lebenszyklus



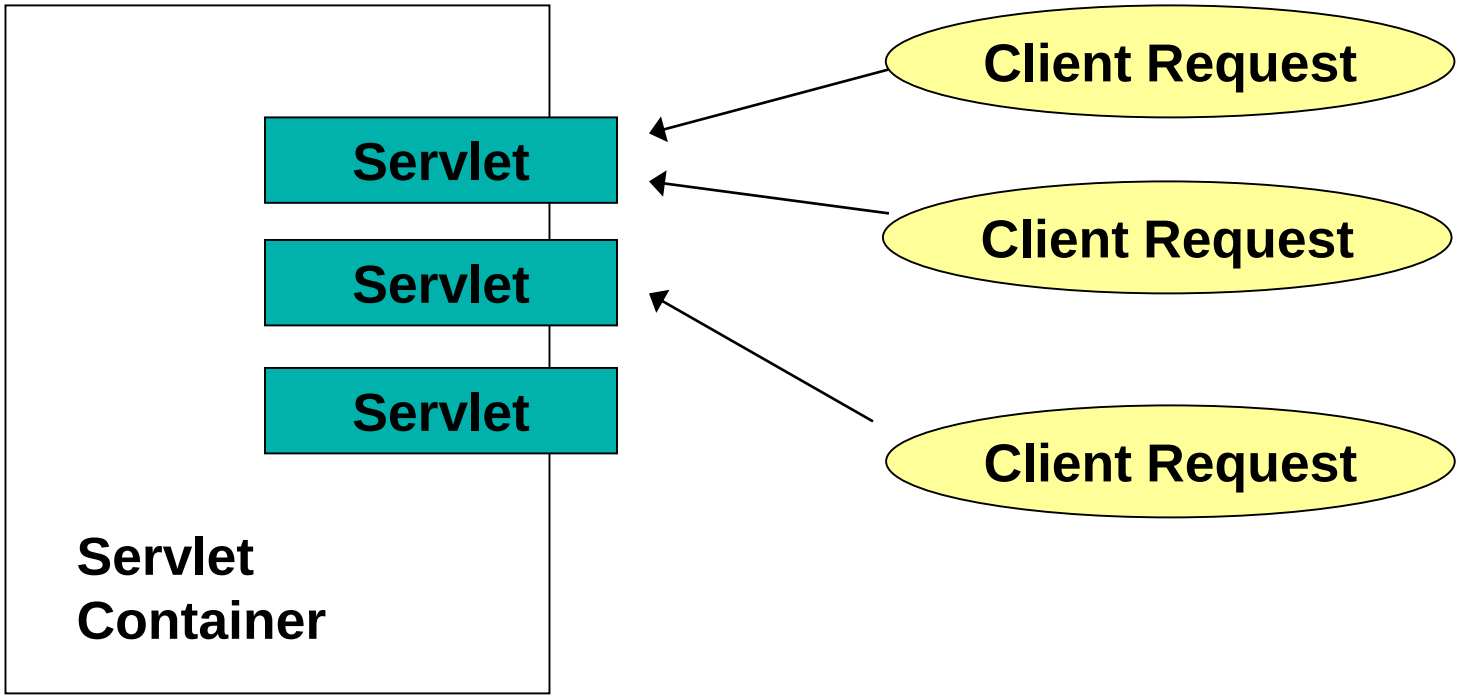
Laden der Servlet-Klasse
Erzeugen einer Instanz
Ausführen der init()-Methode

Prüfen, ob neuere Versionen des *.class-Files vorhanden sind: gegebenenfalls neu laden

Ausführen der Methode service()

Bei Herunterfahren des Servers:
Ausführen der destroy()-Methode

Servlets
Threads



Verschiedene Klienten können Anfragen **zur selben Servlet-Instanz** verschicken. Hierzu wird für **jede Anfrage** ein eigenständiger Thread verwendet

Servlets
Lebenszyklus: API

Optionen zum Laden und Instanziieren des Objektes

- Beim Start des Servlet-Containers (Web-Server)
- Beim Empfang der ersten Anfrage

Initialisierung

- Hängt ab von der Konfiguration (web.xml oder Annotation): Wenn hier `<load-on-startup> 1`, dann wird dies beim Starten des Containers gemacht (und nicht bei Erstanforderung des Servlets)
- Aufruf der *init*-Methode des Servlets (die überschrieben werden muss oder durch eine Annotation verändert wird)
- Globale Initialisierungsaufgaben, die für alle Anfragen erforderlich sind
  - > Herstellen einer Datenbank- oder Netzwerkverbindung
  - > Einlesen einer Konfigurationsdatei
  - > Starten eines Threads

Client-Anfragen bearbeiten

- Eine Möglichkeit der Anfragebearbeitung ist das Überladen der *service*-Methode
- Hierbei handelt es sich um eine generische Methode, die prinzipiell nicht an HTTP gebunden ist
- Der Programmcode kann auf ein Objekt vom Typ *ServletRequest* zugreifen, wodurch der Zugriff auf alle Daten der Anfrage ermöglicht wird
- Die Methode erzeugt eine generische, nicht HTTP-spezifische Antwort mittels eines Objektes vom Typ *ServletResponse*

Client-Anfragen bearbeiten (Fortsetzung)

- Wird die *service*-Methode nicht implementiert, so kann man bei einer Ableitung eines HTTP-Servlets auch die spezifischen Methoden *doGet* und *doPost* ableiten.
- Auch diese Methoden haben Zugriff auf die Daten durch ein spezielles Objekt. Bei einem HTTP-Servlet ist das Objekt vom Typ *HttpServletRequest*
- Analog wird die Antwort durch ein Objekt vom Typ *HttpServletResponse* manipuliert

Anmerkung

- Da Servlets i.allg. im Kontext des HTTP-Protokolls genutzt werden ist dies die gebräuchliche Form der Implementierung

Servlets
HelloWorld

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

public class HelloWorld extends HttpServlet {

   public void doGet(HttpServletRequest request,
          HttpServletResponse response)
          throws IOException, ServletException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
```

Servlets
HelloWorld

```
Date d = new Date();

out.println(
    "<html>\n" +
    "<head>\n" +
    "<title>Hello World</title>\n" +
    "</head>\n" +
    "<body>\n" +
    "<h1>Hello World</h1>\n" +
    "Wir haben heute den " + d.toString() +

     "</body>\n" +
    "</html>\n"
);
out.close();
```

Servlet-Klasse wieder entladen

- Der Servlet Container entscheidet, wann die Servlet-Instanz wieder aus dem Speicher entfernt wird
- Vorher wird die Methode *destroy* aufgerufen

Servlets
Ein einfaches Template für Servlets

```
import javax.servlet.*;
import javax.servlet.http.*;
// other imports
public class TemplateServlet extends HttpServlet {

   public void init() {
      // Wird bei Erstellung des Servlets aufgerufen
   }

   public void destroy() {
      // Wird bei Beendigung des Servlets aufgerufen
   }

   ...
```

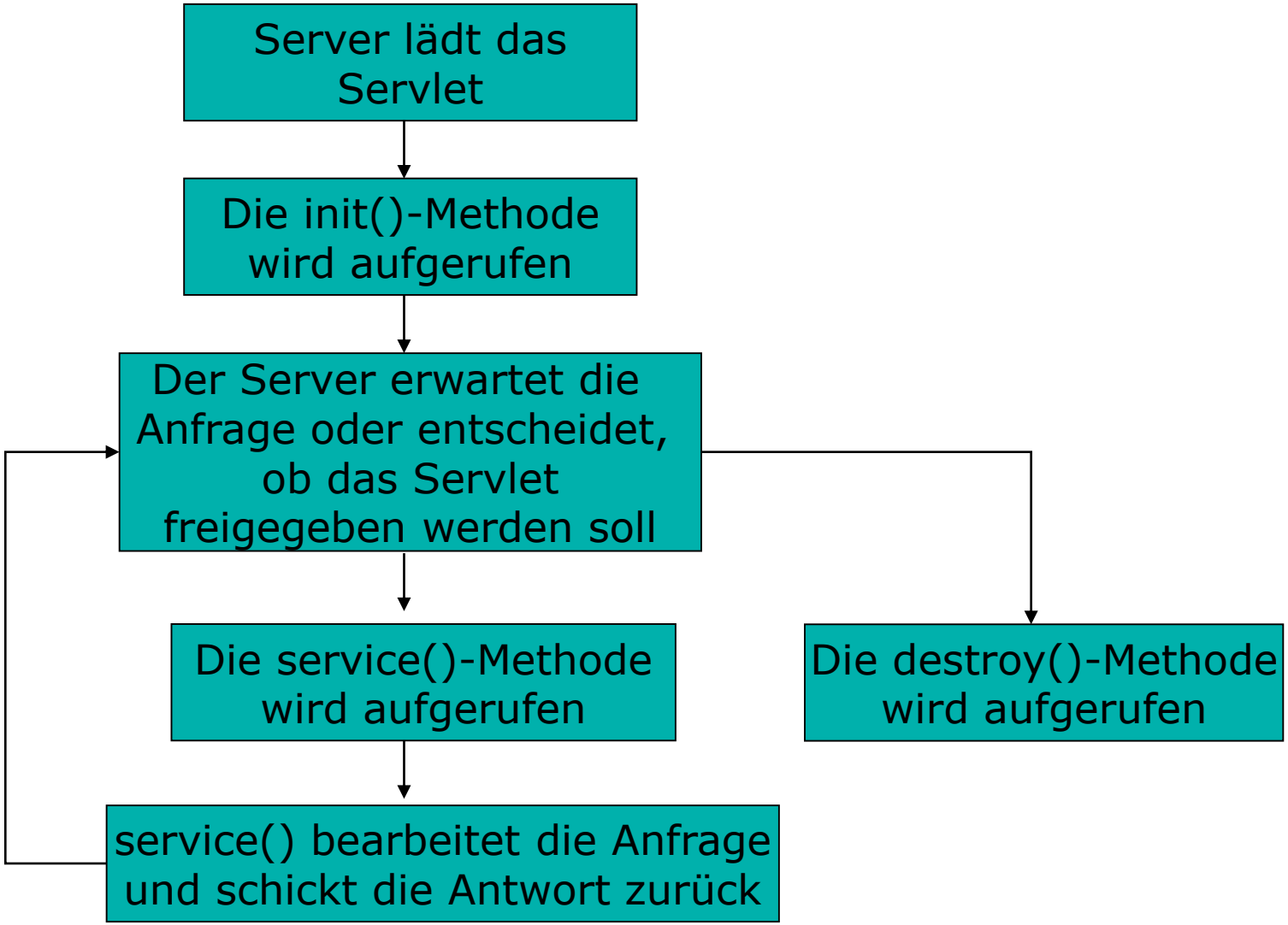
Servlets
Ein einfaches Template für Servlets

```
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {
        // Abarbeitung einer Get-Anfrage
    }

    public void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {
        // Abarbeitung einer Post-Anfrage
    }

    // Weitere Methoden
}
```

Servlets
Ablauf



Servlets
Annotationen

Mit der Servlets 3.0-API wurden Annotationen eingeführt:

- @HandlesTypes
- @HttpConstraint
- @HttpMethodConstraint
- @MultipartConfig
- @ServletSecurity
- @WebFilter
- @WebInitParam
- @WebListener
- @WebServlet

@WebServlet

- Diese Annotation zeichnet die Klasse als Servlet-Klasse aus und ermöglicht beispielsweise die Vergabe eines Namens (auf den sich z.B. Konfigurationen beziehen können)

```
@WebServlet(
    name = "AnnotatedServlet",
    description = "A sample annotated servlet",
    urlPatterns = {"/QuickServlet"}
)
```

@WebFilter

- Diese Annotation ermöglicht eine Vorverarbeitung der Anfragen, um z.B. nur authentifzierte oder gar autorisierte Anfragen zu erhalten. Basis hier sind die Methoden

```
public void init(FilterConfig filterConfig)
public void doFilter (ServletRequest, ServletResponse,
FilterChain)
public void destroy()
```

@WebFilter

```
@WebFilter(urlPatterns = {"/*"}, description = "Session Checker Filter")
public class SessionCheckerFilter implements Filter {
    ...
    public void doFilter(ServletRequest req, ServletResponse res,
            FilterChain chain) throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
        if (!request.getRequestURI().endsWith("login.jsp") &&
                request.getSession().getAttribute("AUTHENTICATED") == null) {
            response.sendRedirect(request.getContextPath() + "/login.jsp");
        }
        chain.doFilter(req, res);
    }
...
```

@WebInitParms

- Mit dieser Annotation lassen sich die init-Methoden der @WebServlet- und der @WebFiler-Annotation parametrisieren

```
@WebServlet(
   urlPatterns = "/uploadFiles",
   initParams = @WebInitParam(name = "location",value = "/tmp"))
public class FileUploadServlet extends HttpServlet {
    // implement servlet doPost() and doGet()...
}
@WebFilter(
   urlPatterns = "/uploadFilter",
   initParams = @WebInitParam(name = "fileTypes", value =
"doc;xls;zip")
)
public class UploadFilter implements Filter {
    // overrides filter methods ..
}
```

Servlets
Umgebungsvariablen

Der Anwendungsentwickler muss bei Servlets nicht extensiv auf Umgebungsvariablen achten.

Hierzu gibt es individuelle Funktionen:

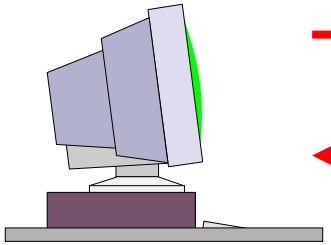
- PATH_INFO request.getPathInfo()
- REMOTE_HOST request.getRemoteHost()
- QUERY_STRING request.getQueryString()

Wie können mehrstufige Operationen/Transaktionen vorgenommen werden?

- Das Servlet muss erkennen, dass der nächste Aufruf der Seite in einem Kontext geschieht
- Lösung wie gehabt: Einsatz von Session-Ids mittels
  - > Cookies
  - > URL-Rewriting
  - > Form Fields

```
Cookie myCookie = new Cookie("name", "value");
response.addCookie(myCookie);
```

Servlets
Datenbankanbindung



HyperText Transfer Protocol

HTML

http-Server

Thread

<FORM METHOD="POST"
ACTION=„MyServlet“

```
public class MyServlet extends HttpServlet ()
{
  while (....)

}          s.executeQuery(select...)
```

JDBC

DBMS

Servlets
Zusammenfassung

Ablauf

1. Erweitern der Klasse-HttpServlet
2. Überschreiben der doGet(....) und/oder doPost-Methode
3. Einlesen der Benutzerparameter mittels HttpServletRequest
   > getParameter("paramName")
4. Erstellen der Antwort mittels HttpServletResponse
   > Content-Type setzen
   > PrintWriter holen
   > HTML-Befehle mittels PrintWriter zum Klienten senden
   > Ggf. weitere Header-Werte setzen, z.B. Cookies
5. Servlets nutzen i.allg. eher Klassenvariablen und verwenden daher nur zu Hilfszwecken Instanzvariablen

Servlets
Bewertung

Vorteile

- Höhere Leistungsfähigkeit
  - > Zustände durch Sitzungskonzept, z.B. für DB-Verbindung
  - > Sehr flexibel
- Threads reduzieren den Ressourcenverbrauch
- Ergeben zusammen mit Enterprise Java Beans Komponenten eine mächtige Entwicklungsbasis

Nachteile

- Abhängig von den Fähigkeiten des http-Servers (kein nativer)
- Verwaltung der verschiedenen Sitzungen muss implementiert werden
- Vermischung von Präsentations- und Anwendungslogik

Servlets
Bewertung

Servlets eignen sich insbesondere für die Anbindung an komplexe Standardanwendungen

- Anwendungslogik wird dabei oft in Beans Container *) realisiert (JBoss/Tomcat)
- Präsentationslogik kann besser gelöst werden

*) Enterprise JavaBeans (EJB) sind standardisierte Komponenten innerhalb eines Java-EE-Servers (Java Enterprise Edition). Sie vereinfachen die Entwicklung komplexer mehrschichtiger verteilter Softwaresysteme mittels Java (z.B. bei der GUI oder bei der Anwendungslogik).

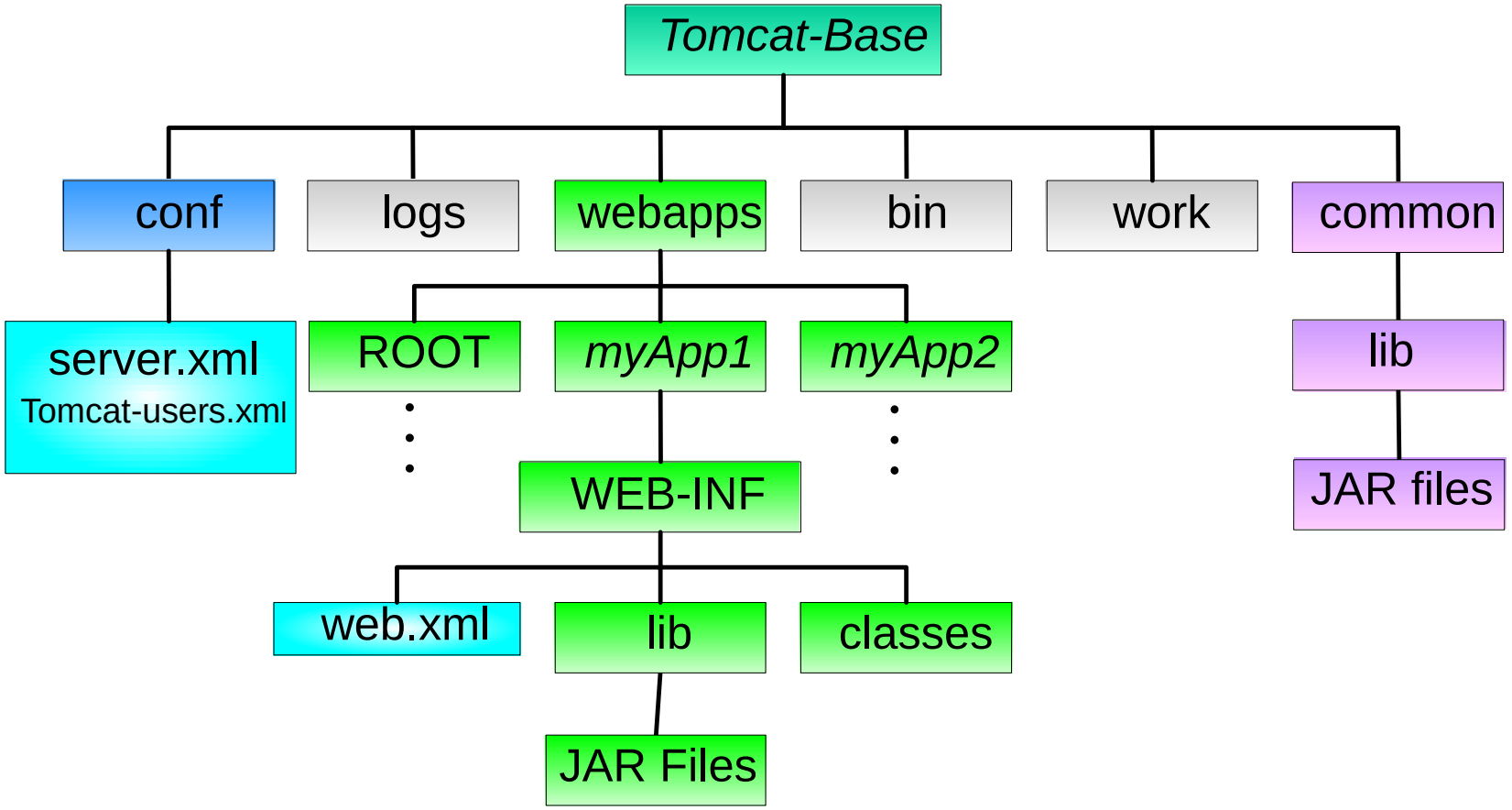
Tomcat ist ein Container für Servlet- und Java Server Pages

- Open Source Referenzimplementierung der Apache Software Foundation des Servlet API
- Wurde bis 2011 im Rahmen des Apache Jakarta Projektes vollständig in Java entwickelt und ist somit für viele Betriebssysteme, unter anderem Linux und Windows, verfügbar

- Die Entwicklung von Tomcat begann bereits 1999 als Nachfolger von Apache JServ
  - > Sun, IBM und Apache gründeten zusammen das Projekt „Jakarta“
  - > Ziel: Implementierung der bis dato nur theoretisch vorhandenen Spezifikation => Referenzimplementierung

Servlets
Apache Tomcat

Servlets
Apache Tomcat



Java Server Pages (JSP)

Idee

- Umkehren des Servlet-Prinzips: „Java-Code generiert HTML“
- JSP-Seiten sind HTML-Seiten, die auch Java-Code beinhalten
- Abgrenzung der Anweisungen durch spezielle Tags (ähnlich zu dem PHP-Prinzip).
  - > Nutzung einer „Skript“-Sprache (analog zu PHP)
  - > Spezielles Verfahren zur Behandlung von JSP-Seiten erforderlich
  - > Auch ohne Java-Anweisungen würde dieses Verfahren angewendet

Java Server Pages sind Servlets

JSP-Seiten werden vom Server automatisch in Servlets übersetzt

- Der Server weiß JSP-Seiten von normalen HTML-Seiten zu unterscheiden
- Er kompiliert mit Hilfe eines JSP-Compilers die Code-Segmente und erstellt ein Servlet
- Das Servlet beinhaltet die HTML-Anweisungen als übliche Ausgabe
- Die globale Datei web.xml gibt die Endung .jsp vor

Anmerkung

- Der Übersetzungsvorgang von JSP in ein Servlet muss dann nur einmal getätigt werden, danach benutzt der Servlet-Container direkt die übersetzte Klasse.

Java Server Pages sind Servlets

Prinzip

- Eine JSP-Seite ist im Prinzip nur eine **Kurzschreibweise** für ein Servlet
- Die Java-Anweisungen werden vom Server automatisch in ein zur Laufzeit erzeugtes Servlet integriert
- Die HTML-Anweisungen der JSP-Datei werden beim Kompilationsschritt einfach an den mit der service-Methode des Servlets erzeugten Ausgabestrom beigefügt

Java Server Pages
Beispiel

```
<%@ page contentType="text/html"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>JSP Beispiel - Hallo Welt!</title>
</head>
<body>

<% out.println("Hello, World!"); %>

</body>
</html>
```

Java Server Pages in Tomcat

JSP-Dateien können unter dem Anwendungsverzeichnis unter webapps gespeichert werden

- Endung ist .jsp
- Gerne in einem eigenen Verzeichnis

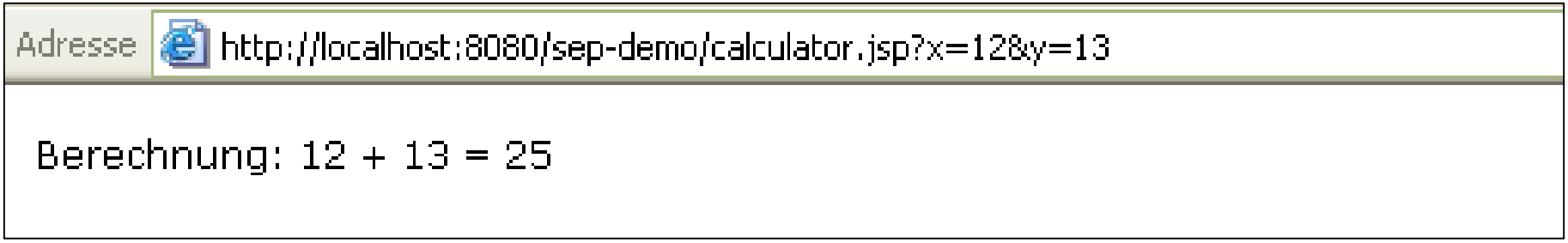
Bei Nutzung des Eclipse-Plugins braucht nur eine .jsp-Datei in einem Tomcat-Projekt angelegt zu werden

- Beim ersten Kontaktieren wird aus der JSP-Datei ein Servlet erzeugt. Dieses finden Sie unter:
  > <workspace>\Tomcat-Projektname\org\apache\jsp\...

Java Server Pages
Beispiel

```
<%-- Parameter lesen --%>
<% int x = Integer.parseInt(request.getParameter("x"));
   int y = Integer.parseInt(request.getParameter("y")); %>

<html>
  <head>
    <title>Calculator</title>
  </head>
  <body>
    Berechnung: <%= x %> + <%= y %> = <%= x+y %>
  </body>
</html>
```



Java Server Pages
Konstrukte

Es gibt neben den HTML-Elementen drei wichtige JSP-Konstrukte:

1. **Scripting-Elemente:** Spezifikation von Java-Fragmenten, die in das erstellte Servlet eingefügt werden
2. **Direktiven:** Sie kontrollieren die Gesamtstruktur des Servlets, z.B. durch Importieren einer Klasse, die in späteren Scripting-Elementen benutzt wird
3. **Aktionen:** Sie starten zusätzliche Funktionalität **zur Laufzeit** und können so die Ausführung des dynamisch erzeugten Servlets beeinflussen und sichern dessen Aktualität

Java Server Pages
Scripting Elemente

In Scripting-Elementen kann implizit auf vorhandene Objekte zugegriffen werden

- **pageContext** – Zugriff auf Objekte in den verschiedenen Gültigkeitsbereichen einer jsp-Seite
- **HttpServletRequest** –response Objekte
- **session** – sitzungsorientierte Anwendungen
- **application** – schreiben von Meldungen in die log-Datei des Servers, Abfrage von Informationen
- **out** – ermöglicht die Ausgabe
- **config** – lesender Zugriff auf Konfigurationsparameter des Servers
- **page** – entspricht der Referenz this des erzeugten Servlets
- **exception** – in jsp-Fehlerseiten verfügbar

Java Server Pages
Scripting Elemente

Ausdrücke:

- Einfache Java-Anweisungen, die in das Servlet an entsprechender Stelle integriert werden (Ausdruck wird mittels out.print() eingefügt)
- Ermöglichen das Einfügen dynamischer Werte in eine HTML-Datei

Syntax:

- Standard: <%= Java-Ausdruck %>
- Alternative XML-Syntax:

  <jsp:expression> Java-Ausdruck </jsp:expression>
- Unterstützt werden nur einfache Ausdrücke - keine komplexen Strukturen wie z.B. Schleifen oder bedingte Anweisungen

```
Date: <%= new java.util.Date() %>
Host: <%= request.getRemoteHost() %>
```

Java Server Pages
Scriptlets

Komplexere Java-Anweisungen

- Idee: Die *service*-Methode des generierten Servlets ruft eine *_jspService*-Methode auf, die das Programmfragment enthält. Damit werden etwaige Ausgaben mittels der verfügbaren PrintWriter-Methoden erstellt (Referenz out).
- Erlauben z.B. die Programmierung komplexerer Strukturen wie Schleifen und bedingte Verzweigungen sowie das Setzen von Headern und Statusinformationen.

Syntax:

- Standard: <% Java-Code %>
- Alternative XML-Syntax: <jsp:scriptlet> Java-Code </jsp:scriptlet>
- Code-Fragmente müssen nicht notwendigerweise vollständig sein

```
<%
   String s = request.getQueryString();
   out.println( "GET data: " + s );
   if( boolVar ) {
%>
   ….
<jsp:forward page='<%= Second.jsp %>' /%>
<% } %>
```

Java Server Pages
Deklarationen

Deklarationen:

- Methoden oder Deklarationen, die in den Hauptrumpf der Servlet-Klasse (außerhalb der _jspService-Methode) eingefügt werden

Syntax:

- Standard: <%! Java-Code %>
- Alternative XML-Syntax:

  <jsp:declaration> Java-Code </jsp:declaration>

Beispiel: JSP, die die Anzahl der bisherigen Zugriffe ausgibt

```
<%! private int accessCount = 0; %>
    Accesses to page since server reboot:
    <%= ++accessCount %>
```

Java Server Pages
Direktiven

Erneut gibt es drei Arten von Direktiven:

1. **page**: Steuert die Struktur eines Servlets z.B. durch das Importieren einer Klasse oder durch das Beeinflussen des Übersetzungsprozesses:

```
<%@ page import="java.sql.*" %>
```

2. **include**: Fügt zum **Zeitpunkt der Übersetzung** der JSP-Datei eine weitere JSP-Datei ein. Die Anweisung sollte dort stehen, wo der Text der Datei stehen soll:

```
<%@ include file="Gut-Erprobtes.jsp" %>
```

   > **Achtung**: Hauptseite und inkludierte Seite haben den gleichen Namensraum!

3. **taglib**: Ermöglicht das Einführen eigener JSP-Tags, die dann mittels einer sogenannten Tag-Handler-Klasse ausprogrammierte Programmsequenzen über die übliche HTML-Tag-Notation zugänglich macht

```
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
...
    <Title><csajsp: Ein Beispiel /><Title/>
```

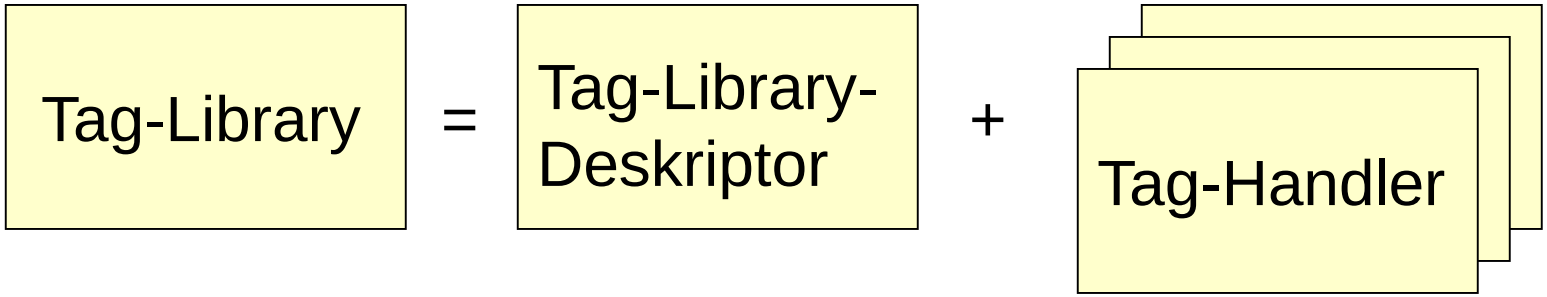
Java Server Pages
Funktionsweise einer Taglib

Tag-Library-Deskriptor (TLD)

- XML-Datei
- Beschreibt, welche Tags Tag-Library bereitstellt
- Legt für jedes Tag Tag-Handler-Klasse fest

Tag-Handler (in WEB-INF/classes | lib)

- Java-Klassen (eine Klasse pro Tag)
- Implementieren Aktionen, die beim Auftreten des Tags ausgeführt werden



Java Server Pages
Aktionen

Include

- Einfügen einer HTML- oder JSP-Seite zur Laufzeit (<%@ include file="..." %> dagegen fügt zu Compilierzeit ein.).

- Wichtig:
- Die eingefügte Seite verfügt über einen **eigenen Namensraum**. Konflikte mit gleichnamigen Variablen gibt es nicht.
- In der Zielseite dürfen **keine HTTP-Header** mehr gesetzt werden! Andernfalls gibt es eine Exception.

```
<jsp:include page="mycommon.jsp"> <jsp:param name="extraparam"
value="myvalue"/> </jsp:include>
```

- Die Zielseite kann **dynamisch** ausgewählt werden. Hier im Beispiel hängt die Zielseite von einer Zufallszahl ab.

```
<% int zufall = (int) (Math.random()*100); %>
<jsp:include page =‘ <%= (zufall%2) = = 0 ? "seite1.jsp" :
"seite2.jsp" %> ‘ />
```

Java Server Pages
Aktionen

Forward

- Anfrage und Antwort werden an eine andere JSP-Seite, anderes Servlet übergeben. Die Steuerung kommt nicht mehr zur gegenwärtigen JSP zurück.

```
<jsp:forward page="subpage.jsp">
  <jsp:param name="forwardedFrom" value="this.jsp"/>
</jsp:forward>
```

- Regeln:
  1. Wenn die Seitenausgaben über *out* gepuffert sind - die aufrufende Seite bereits eine Ausgabe erzeugt hat- , wird der Puffer vor der Weiterleitung der Seite gelöscht
  2. Wenn die Seitenausgaben über *out* gepuffert sind und der Puffer bereits geleert ist (flush), endet der Versuch einer Weiterleitung mit einer *IllegalStateException*.
  3. Wenn die Seitenausgabe ungepuffert ist und irgendetwas bereits über *out* zurückgegeben wurde, endet der Versuch einer Weiterleitung ebenfalls mit einer *IllegalStateException*
- Weiterleitung und Redirection ist eine Header-Angelegenheit!

Java Server Pages
Nutzung in Servlets

Bisher hatten wir ein entweder-oder-Konzept kennen gelernt. Entweder mussten Servlets die Aufgabe übernehmen, HTML-Seiten als Ausgabe zu Erzeugen, oder aber eine jsp-Dateien musste die Java-Anweisungen enthalten. Mit JSP kann eine Trennung zwischen Präsentations- und Anwendungslogik stattfinden!

- Logik/Ablaufsteuerung wird mittels Servlets implementiert
- Präsentationsform durch HTML in JSP-Seiten.
- Vorteil: Es können auch die üblichen HTML-Generatoren genutzt werden

Java Server Pages
Nutzung in Servlets

Das klassische Servlet-Konzept übernimmt die grundlegende Kontrolle. Das Schlüsselelement mit dem dieses Servlets verlassen werden kann heißt RequestDispatcher

- Ein RequestDispatcher wird durch die Methode getRequestDispatcher von ServletContext instanziiert. Hierbei erhält die Methode einen Pfad, in dem z.B. die JSP-Seite relativ zum Verzeichnis der Anwendung zu finden ist

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(Pfad);
```

- Mittels der *forward*-Methode des RequestDispatchers kann dann die Steuerung **vollständig** an die JSP-Seite geleitet werden
- Die *include*-Methode **bindet den Inhalt** eines Servlets, JSP oder einer Web-Seite in den aktuellen Datenstrom (response) **ein**. Die **Kontrolle bleibt beim Servlet**

Java Server Pages
Beispiel zur Nutzung in Servlets

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

    public class ServletJSP extends HttpServlet {
      public void service( HttpServletRequest  request,
                           HttpServletResponse response
            throws IOException, ServletException ) {
         // Zunächst zeigen wir, wie wir Werte übergeben
         String test = "Ich wurde vom Servlet ermittelt";
         request.setAttribute("Parameter1", test);

        // Aufruf der JSP-Seite

        // Erzeuge RequestDispatcher
        RequestDispatcher meinDispatcher =
    getServletContext().getRequestDispatcher("/ServletJSP/Jsp.jsp");

        // Fuehre den Aufruf durch
        meinDispatcher.forward(request, response);
      }
    }
```

Java Server Pages
Die JSP-Seite

```
<html>
   <body>
   <h1>Hallo ich wurde von einem Servlet erzeugt!</h1>

Und das war der Parameter:
<%= (String) request.getAttribute("Parameter1");%>
</body>
</html>
```

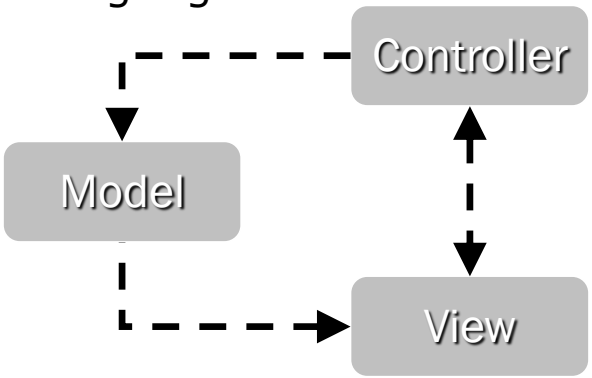
Die Nutzung der RequestDispatcher-Klasse erlaubt die Erstellung von JSP-Seiten mit nur wenig Java-Code-Anteilen

- Servlets für die Verarbeitung der Benutzeraktionen und die Applikationssteuerung
- JSPs für die Bereitstellung der Präsentation
- Extensive Nutzung von JSP Tag Libraries

Model View Controller

Das Prinzip

- Model-Objekt
  - > definiert die Datenstruktur der Anwendung
  - > speichert die Daten
  - > stellt Methoden zur Änderung der Daten zur Verfügung
  - > Realisiert manchmal die Geschäftslogik



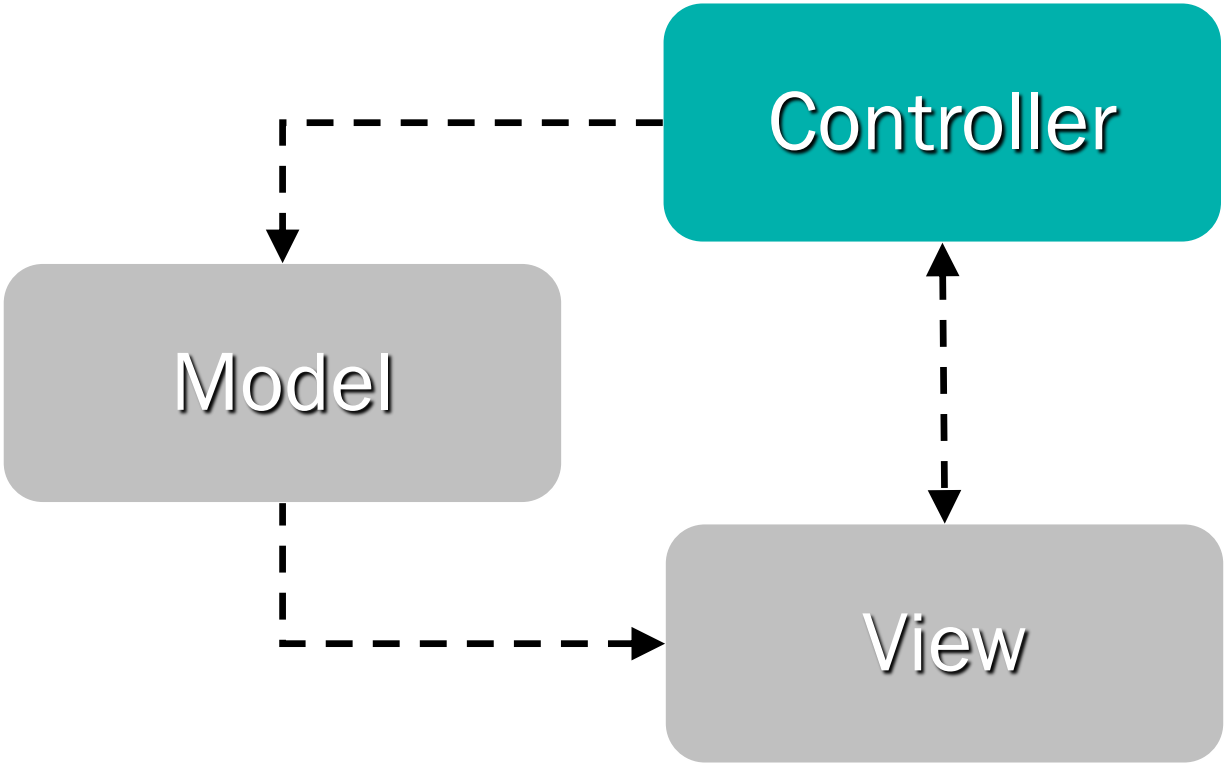
- View-Objekt
  - > stellt die Bildschirmrepräsentation dar
  - > Objekt erhält die Daten vom Model
  - > Benutzer führt auf der View die Aktionen aus
  - > die Aktionen werden durch den Controller an das Model weitergeleitet)

- Controller-Objekt
  - > Reaktion und Verarbeitung von Benutzereingaben
  - > Vermittler zwischen Model und View
  - > Beinhaltet oftmals die Geschäftslogik

Model View Controller

Der Controller

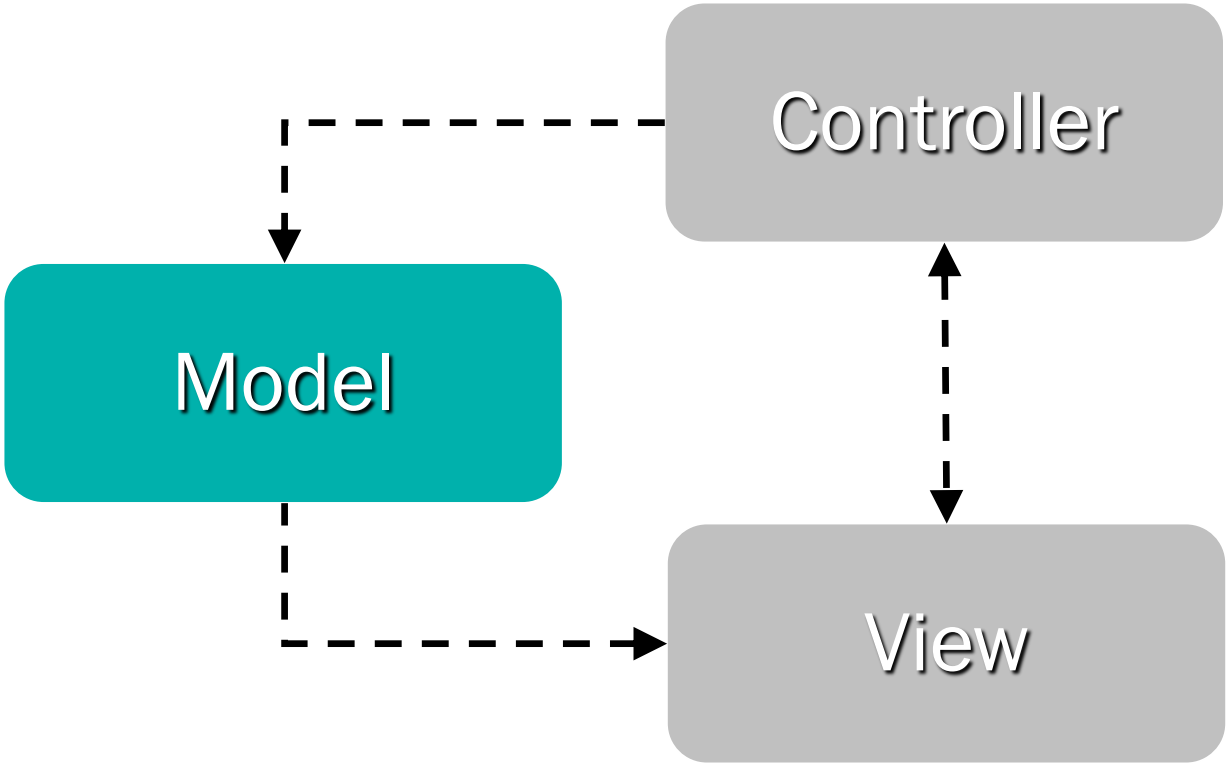
- Überwacht die Interaktion mit dem Benutzer (Tastatureingaben, Mausbewegung, etc.)
- Informiert das Model und/oder View über eventuelle Änderungen
- Implementiert manchmal auch die Geschäftslogik



Model View Controller

Das Model

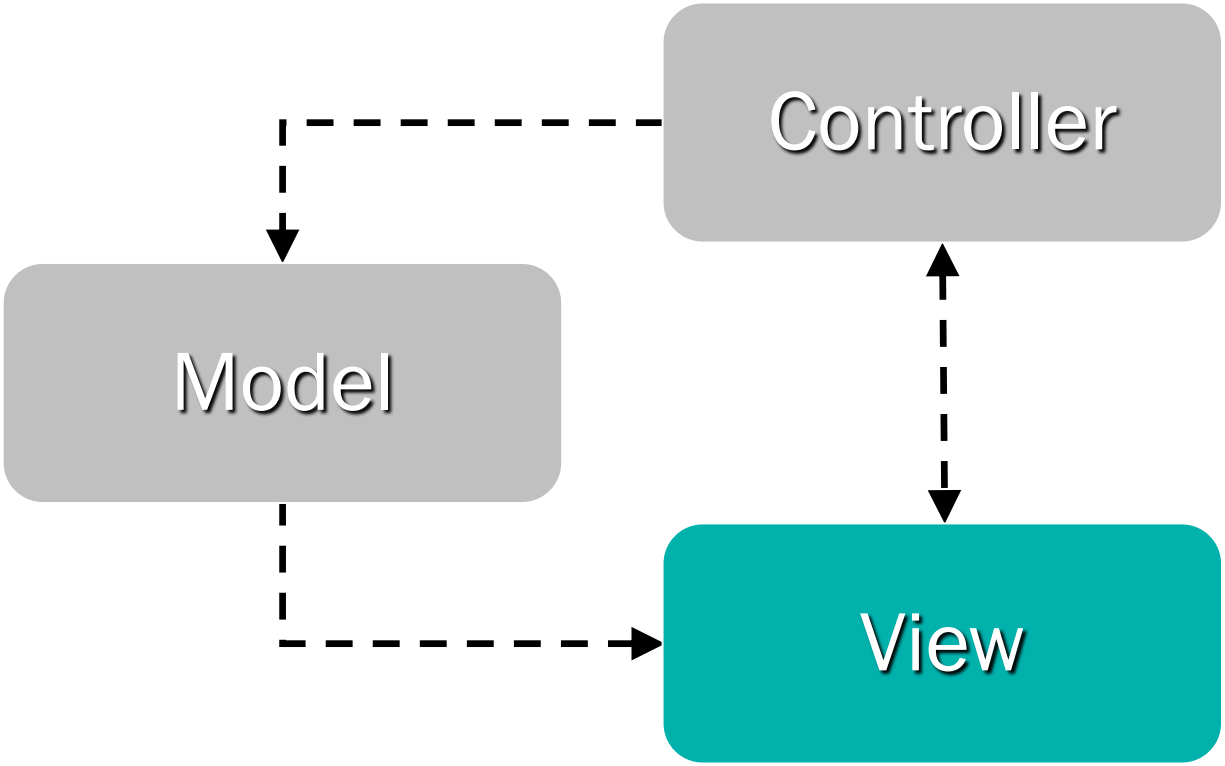
- Steuert das Verhalten der Applikation und den Zugriff auf Daten
- Implementiert manchmal die Geschäftslogik
- Antwortet auf Anfragen nach dem aktuellen Status
- Führt etwaige Datenänderungen durch (Persistenzsicherung)



Model View Controller

Die View

- Ist für die Darstellung der Informationen zuständig
- Benutzer führt auf der View die Aktionen aus. (die Aktionen werden durch den Controller verarbeitet und an das Model weitergeleitet)
- Eine Applikation kann mehrere Views haben



Model View Controller

Implementierung

- Model
  - > Realisation zumeist mittels Java Beans (*) nächste Folie)
  - > Java Beans enthalten die eigentliche Programmlogik
  - > Java Beans berechnen Ergebnisse und speichern Zustände
  - > Java Beans sind unabhängig von der Webschnittstelle
- View
  - > Nutzung von JSP-Seiten
  - > Anzeige des Ergebnisses
  - > Ziel: Möglichst wenig Java Code
- Controller
  - > Programmieren eines Servlets
  - > Einlesen und Überprüfen der übergebenen Parameter
  - > Aufrufen der eigentlichen Programmlogik im Model
  - > Weitergabe des Ergebnisses an das passende View

MVC - Enterprise Java Beans aus "Mastering EJB 4.0"

Enterprise Java Beans ... are meant to perform server side operations, such as executing complex algorithms or performing highly transactional business operations. Server components need to run in a highly available (7x24), fault tolerant, transactional, multi-user, secure environment. The application server provides such a server side environment for the enterprise beans...Typically, EJB components can perform any of the following tasks

- Perform business logik
- Access a database
- Integrate with other systems

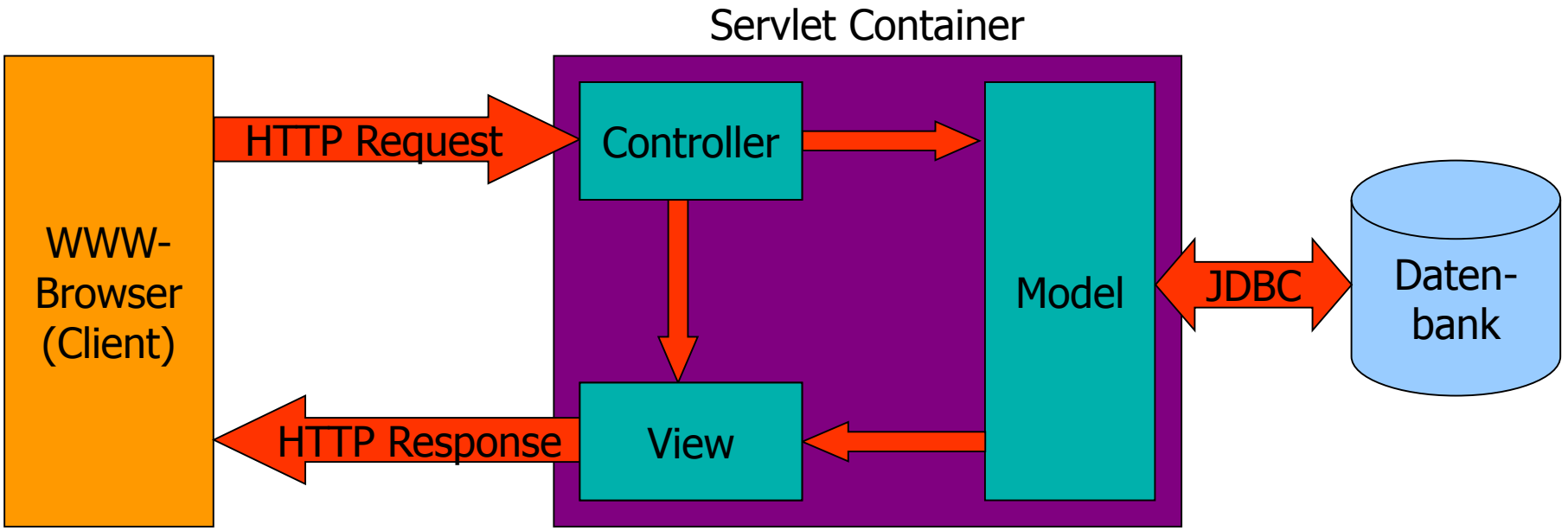
Model View Controller

Implementierung mittels Enterprise Java Beans

- Model
  - > Realisation zumeist mittels Java Beans
  - > Java Beans enthalten die eigentliche Programmlogik
  - > Java Beans berechnen Ergebnisse und speichern Zustände
  - > Java Beans sind unabhängig von der Webschnittstelle
- View
  - > Nutzung von JSP-Seiten
  - > Anzeige des Ergebnisses
  - > Ziel: Möglichst wenig Java Code
- Controller
  - > Programmieren eines Servlets
  - > Einlesen und Überprüfen der übergebenen Parameter
  - > Aufrufen der eigentlichen Programmlogik im Model
  - > Weitergabe des Ergebnisses an das passende View

Model View Controller

Enterprise Java Beans



Model View Controller 2

Was wir brauchen sind:

- eine View, die weiß, wie die Daten repräsentiert werden
- einen Controller, der weiß, wie mit der Geschäftslogik umgegangen werden soll
- ein Model, das die Daten hält aber von beiden nichts weiß

Aus beiden Komponenten (JSP und Servlets) musste ein neues Konzept –MVC²- erstellet werden.

Die Servlets als Controller steuern den Control-Workflow, während die JSP's als Views nur die Daten darstellen.

Model View Controller 2

Controller

- ist ein JavaServlet
- nimmt die Benutzeranfragen (Requests) entgegen
- erzeugt das Model (JavaBeans)
- greift auf die *get* und *set* Methoden des Models zu

- trennt die technischen Controller von fachlichen Controllern
- wählt die zu zeigende Seite aus
- ist konfigurierbar

Model View Controller 2

Model

- ist ein JavaBean
- hält die Daten
- weiß **nichts** über die View und den Controller

- stellt die Daten für die JSP bereit
- ermöglicht es dem Controller, die Daten durch *get* und *set* Methoden zu ändern
- repräsentiert den Zustand der Anwendung

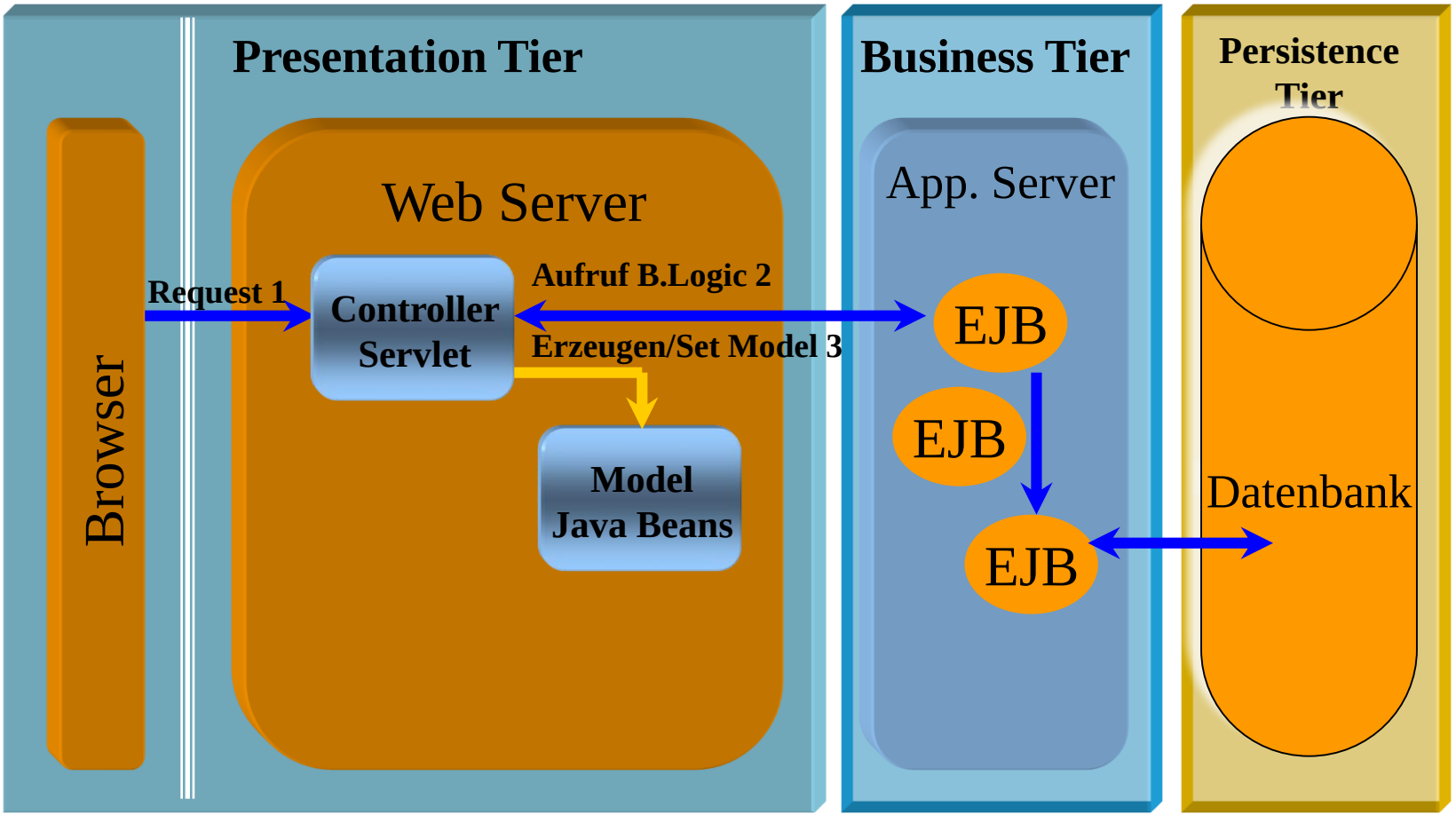
Model View Controller 2

View

- ist eine JSP
- liest die Daten aus dem Model
- erstellt die darzustellenden Dokumente
- stellt die Daten dar
- greift nicht auf die *set* Methoden des Models zu

- stellt die Daten für die JSP bereit
- ermöglicht es dem Controller, die Daten durch *get* und *set* Methoden zu ändern
- repräsentiert den Zustand der Anwendung

Model View Controller 2



Model View Controller 2

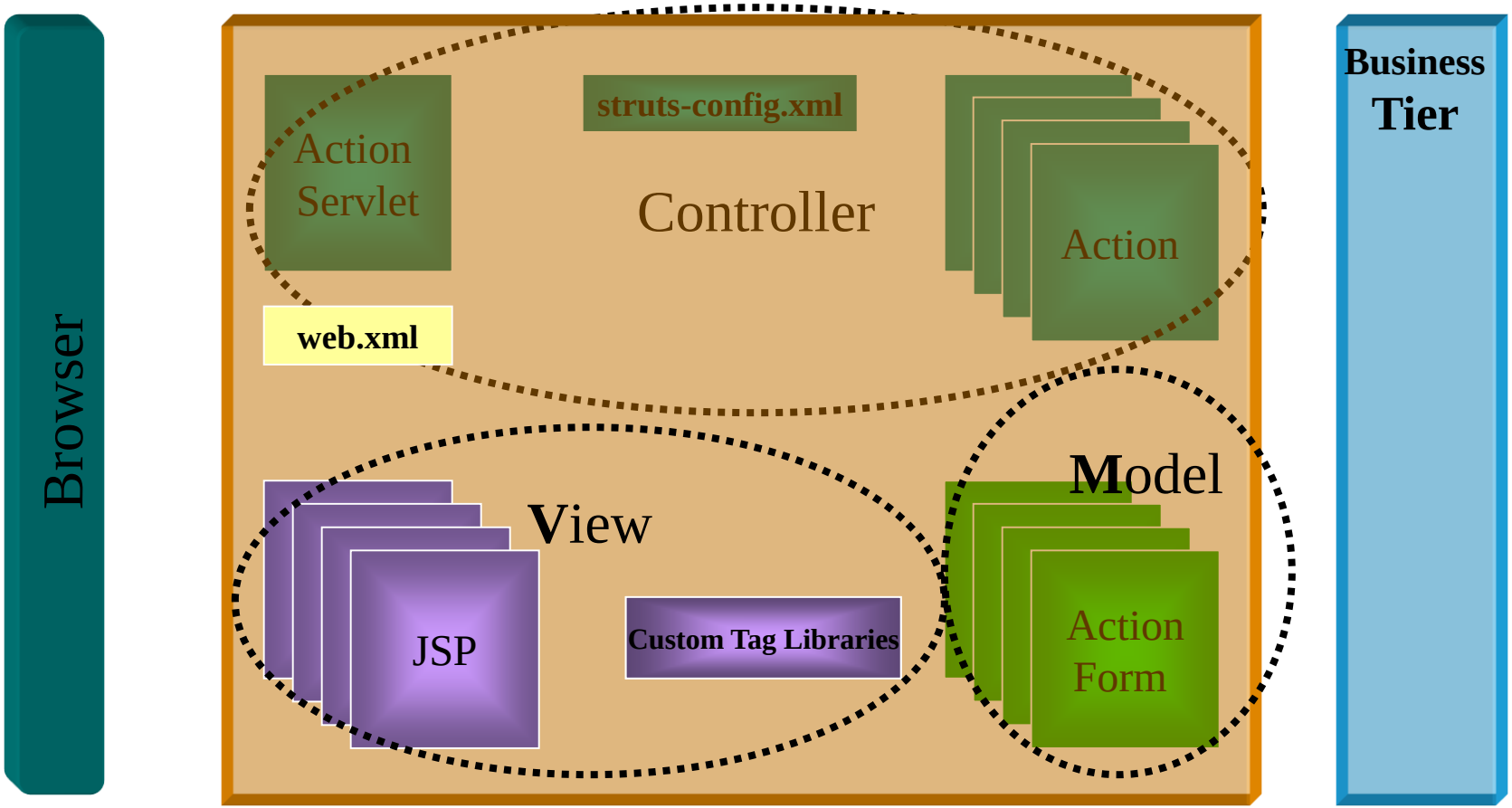
Struts

- Ein MVC **Model²** basiertes „Open-Source Framework" für die **Präsentationsschicht** von Web-Anwendungen.
- Bestandteile:
  - > kooperierende Klassen
  - > Servlets
  - > JSP-Tags

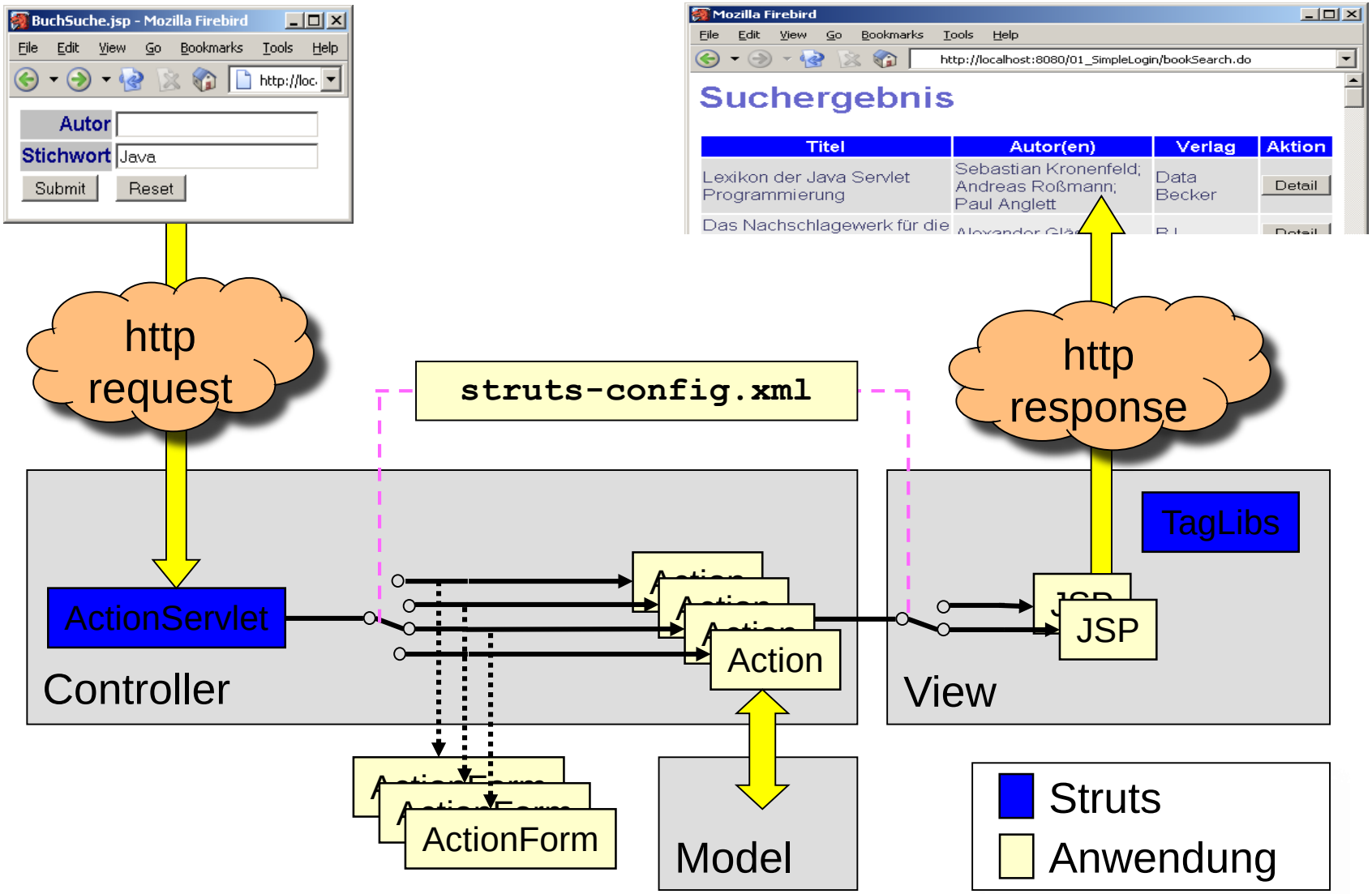
- Bei Struts[1] ist das Controller-Servlet schon implementiert. Es kann via Konfigurationsparameter den Bedürfnissen angepasst werden
- Unterstützung der JSP-Seiten-Erstellung durch eigene tag-Bibliotheken

[1] Struts ist ein Open-Source-Framework für die Präsentations- und Steuerungsschicht von Java-Webanwendungen (siehe auch nächste Folie)

Model View Controller 2 Struts



Model View Controller 2 Struts



Java Server Faces (JSF)

Java-basierte APIs zur Verwaltung von Zustandsinformationen und der Behandlung von Ereignissen in Web-basierten Umgebungen

- übernimmt Konzepte von Struts
- Komplexe Tag-Bibliothek
- nutzt Konzepte von ASP.NET

JSF API und Referenzimplementierung

- muss nicht auf JSP basieren
- Sun RI (Reference Implementation) and MyFaces (Apache)

RESTful Services

REpresentational State Transfer

- Architektur, die den Prinzipien aller webbasierten Systeme folgt, jedoch nicht wirklich als Architekturprinzip beschrieben wurde
- REST ist ausschließlich eine Architektur
  > Web Services sind beides: Standard und Architektur

- Die Architektur beschreibt die Interaktion von Anwendungen gemäß dem webbasierten Interaktionsmodell.
  > Verwendung von HTTP als Transportprotokoll

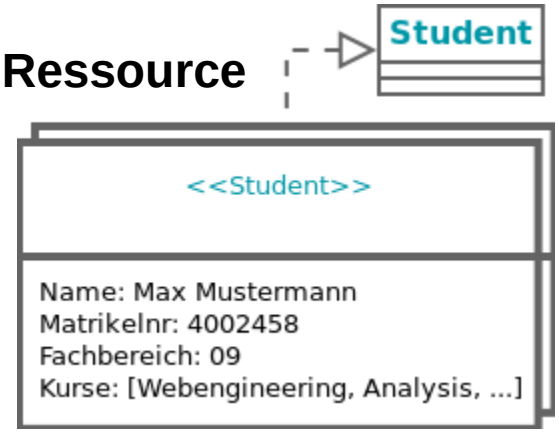
RESTful Services
Konzept

URL

http://localhost/StudentAPI/student/4002458

identifiziert

Ressource



Repräsentation (JSON)

```
{
 „Name“: „Max Mustermann“,
 „Matrikelnr“: 4002458,
 „Fachbereich“: „09“,
 „Kurse“: [„Webengineering“,
             „Analysis“, …]
}
```

repräsentiert

- Ressourcen (HTML-Seiten, Bilder, XML-Dokumente) werden über URI angefordert
- Template URI Matching auf Ressource
- Rückgabe als JSON-Objekt (andere Rückgabeformate möglich)

RESTful Services

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

Roy Fielding: *Architectural Styles and the Design of Network-based Software Architectures*, Dissertation, 2000

RESTful Services
Eigenschaften

Adressierbarkeit

- Jede Ressource ist über eine URI identifizierbar

Zustandslosigkeit

- Keine Anwendungszustände im Server (nur Ressourcenzustände)

Einheitliche Schnittstelle

- CRUD-Operationen (GET, POST, PUT, DELETE)

Entkopplung von Ressourcen und Repräsentation

- Anforderung einer Ressource im gewünschten Format (JSON, XML, HTML, …)

Hypermedia

- Verwendung von Hyperlinks zur Navigation

RESTful Services
CRUD

Die Interaktion lässt sich mittels einer einfachen Abbildung auf die HTTP-Verben beschreiben

(CRUD-Operationen)

Action	Verb
Create	POST
Retrieve	GET
Update	PUT
Delete	DELETE

RESTful Services
CRUD

Create

- POST /resourceName

Retrieve

- GET /resourceName/resourceId

Update

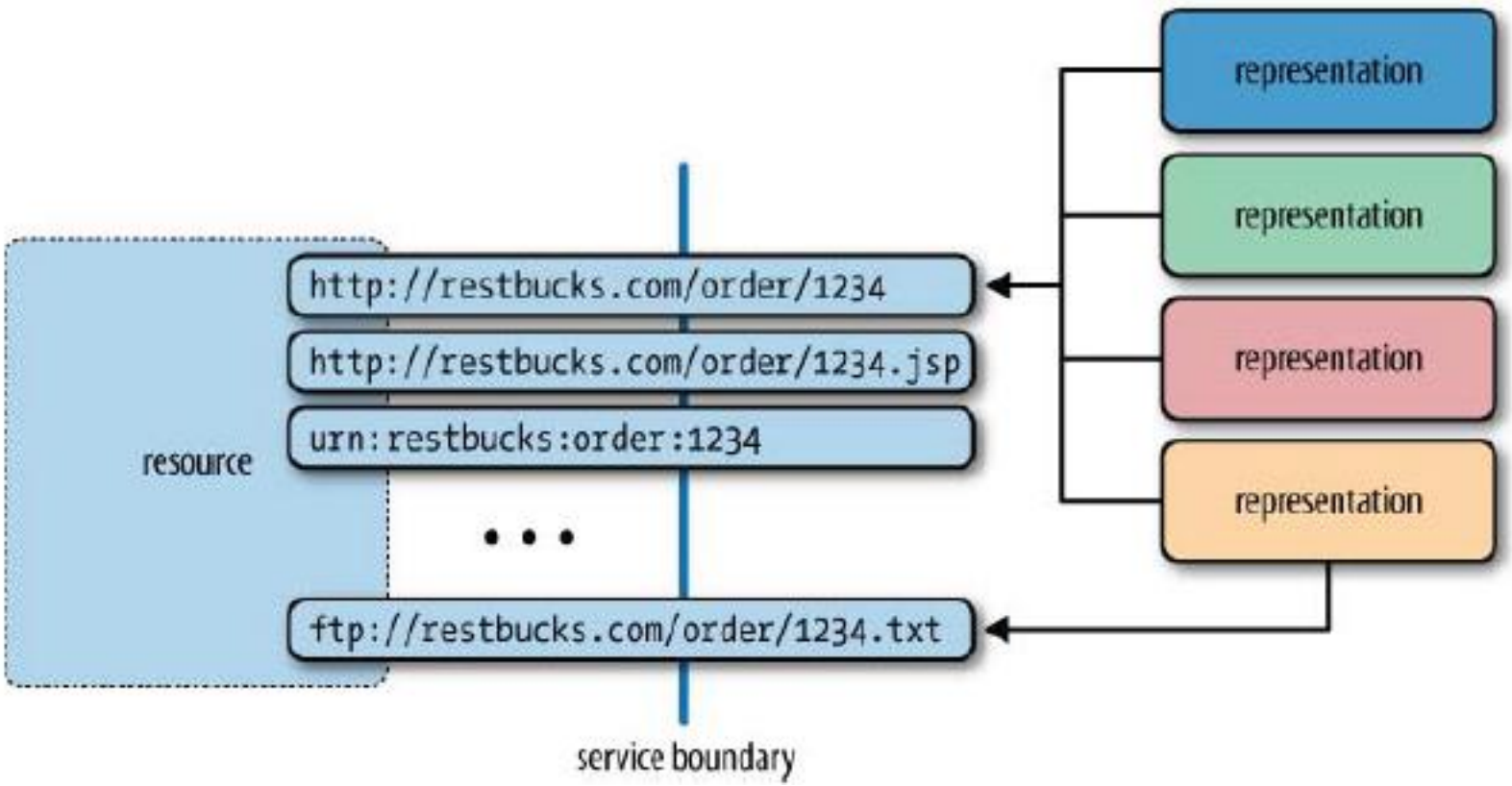
- PUT /resourceName/resourceId

Delete

- DELETE /resourceName/resourceId

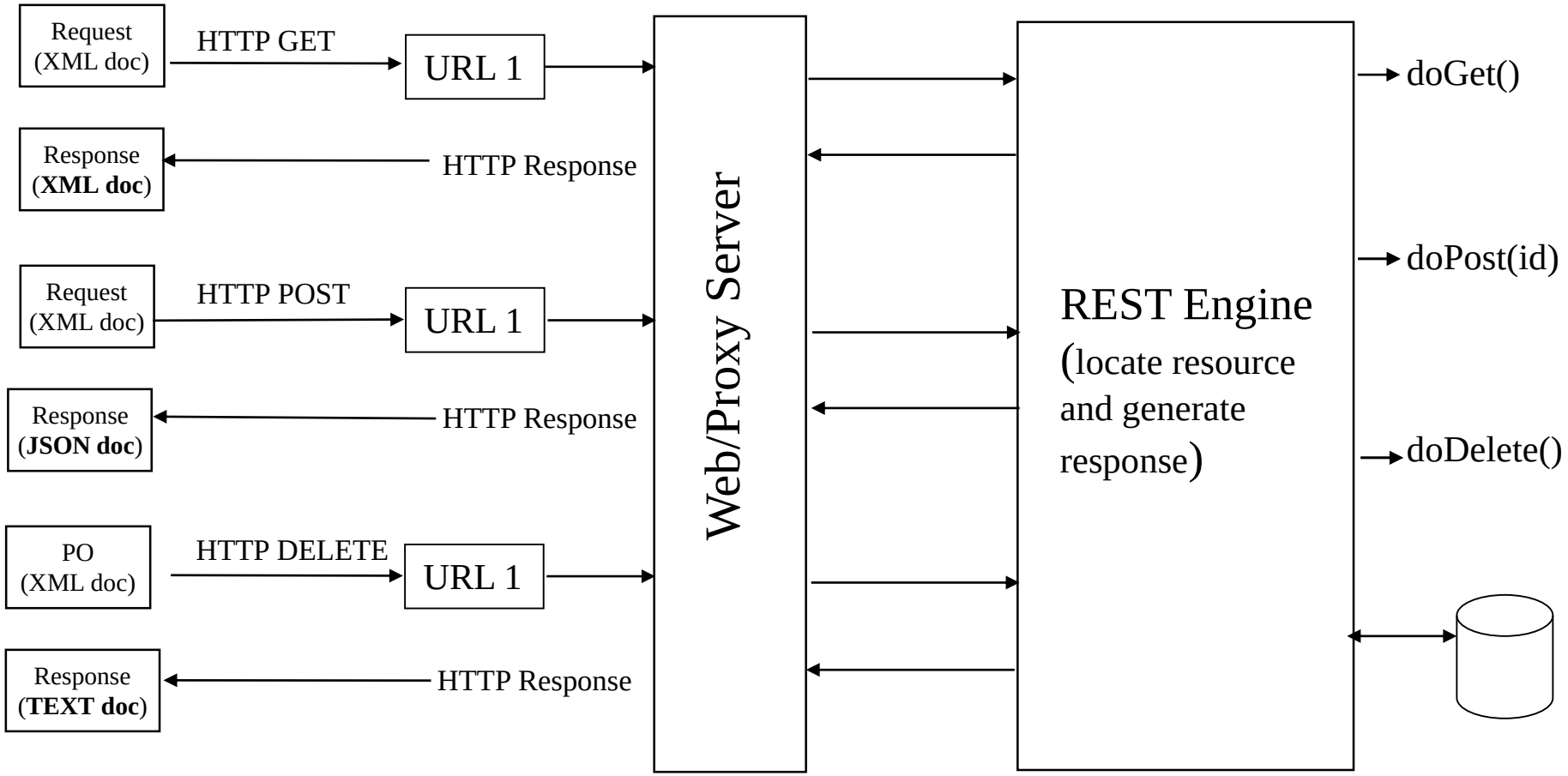
RESTful Services
Representational State Transfer

Identifikation **einer** Ressource durch **unterschiedliche** URIs



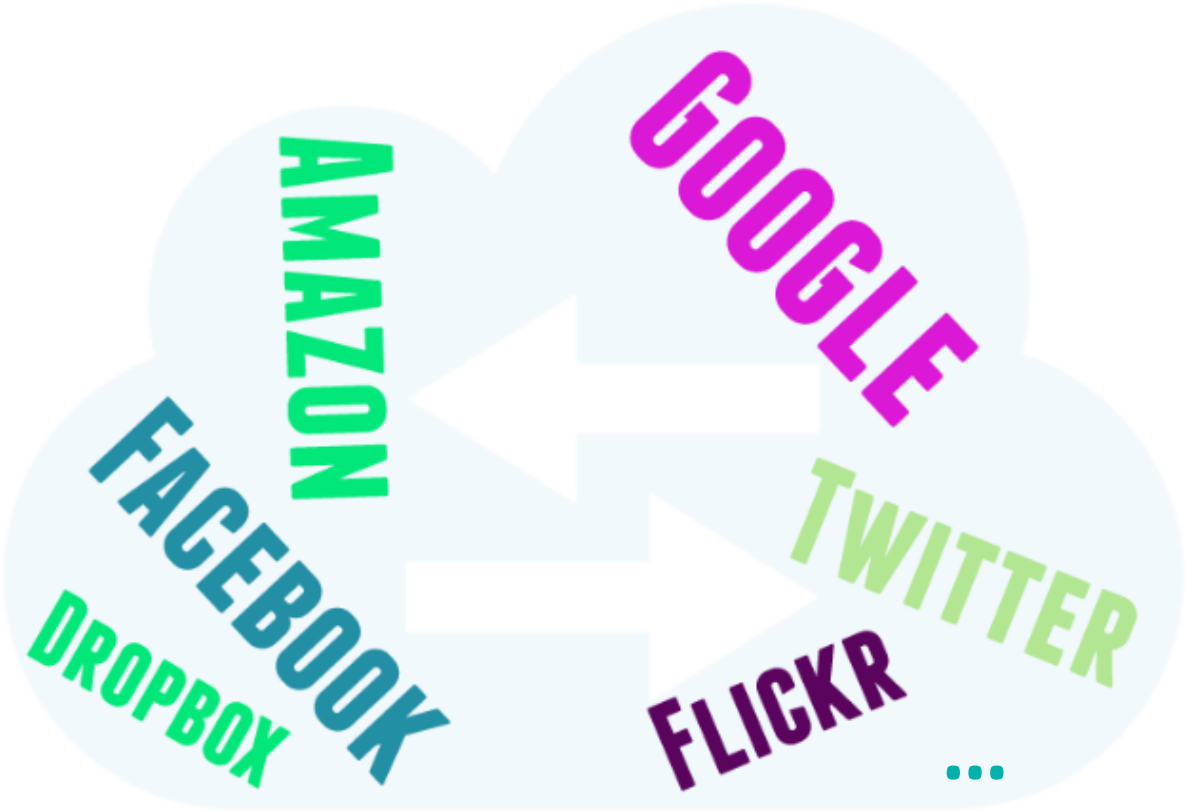
Robinson, Webber, Parastatidis: REST in Practice: Hypermedia and Systems Architecture, O'Reilly Media, 2010

RESTful Services
Eine Übersicht



RESTful Services

Wo finden wir REST?



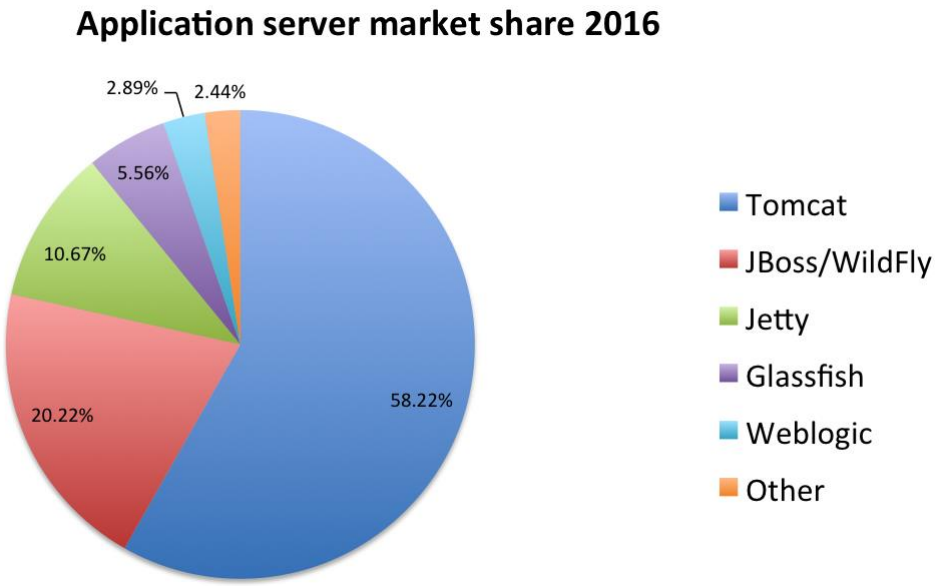
Java API für RESTful Web Services (kurz: JAX-RS)

- Standard zur Bereitstellung von REST Services mit Java
- Referenzimplementierung: **Jersey**
  - > baut auf Servlet auf
    - implementiert init und destroy-Methoden
    - kümmert sich um Lebenszyklus (via Annotationen anpassbar)
      - z.B. `@Singleton`
  - > implementiert REST Spezifikationen
  - > Mapping von URLs auf Methoden via Annotationen
  - > wird intern von Application Servern verwendet

Application Server

Application Server

- Spezialisierter Webserver
  - > integriert Frameworks für Java Web Apps (z.B. Servlets)
- dienen als Schnittstelle zwischen HTTP und Java-Klassen
- Beispiele
  - > Tomcat
  - > JBoss
  - > Wildfly
  - > Glassfish
  - > …
- Konfiguration mittels web.xml



https://plumbr.eu/uncategorized/most-popular-java-ee-servers-2016-edition

Beispiel web.xml (Deployment Konfigurationsdatei)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version=„3.1"
          xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
          http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
          <display-name>Restful Web Application</display-name>

          <servlet>
                    <servlet-name>FH-Aachen StudentAPI</servlet-name>
                    <servlet-class> org.glassfish.jersey.servlet.ServletContainer
                              </servlet-class>
                    <init-param>
                         <param-name>org.glassfish.jersey.spi.container.ContainerResponseFilters
                              </param-name>
                         <param-value>de.aachen.fh.studentAPI</param-value>
                    </init-param>
                    <load-on-startup>1</load-on-startup>
          </servlet>

          <servlet-mapping>
                    <servlet-name>FH-Aachen StudentAPI</servlet-name>
                    <url-pattern>/rest/*</url-pattern>
          </servlet-mapping>

</web-app>
```

Verweis auf verwendete Jersey-Implementierung (abhängig vom verwendeten Application Server)

Package Name mit Ressourcen Klassen

EntryPoint

Jersey

Entwicklungsphase

- Definition von Ressourcen und zugehörigen Aktionen über JAX-RS Annotationen (@Path, @GET, …)

Entwickler

Bei Start des Servlets:

- Auslesen von Annotationen mit Hilfe von Reflections
- Generierung von sog. Path-Templates aus Annotationen

Für jeden Request:

- Matching von HTTP-Methode + REST-URL mit Path-Template

Jersey

GET *http://localhost/MyProject/MyAPI/MyClass/MyMethod/MyParam*

Projektname

Entrypoint

```
@Path(“”“)
class MyClass {
}
```

```
@Path(“MyMethod“)
public void method(@PathParam(„StringParam“)
String param){
}
```

Java API für RESTful Web Services

Nutzung von JAX-RS auch ohne Applikationsserver möglich (Java 7 implementiert die JAX-RS-API nicht)

- Folgende Java-Archive müssen gedownloadet und in den Klassenpfad aufgenommen werden:
  - > jersey-bundle.jar
  - > jsr311-api.jar
  - > asm.jar

Beispiel Service Code

```
import javax.ws.rs.*;

@Path("message")
public class MessageResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String message() {
        return "Yea!";
    }
}
```

`@Path:` Pfadangabe, die auf den Basispfad (EntryPoint) gesetzt wird
`@GET:` HTTP-Methode
`@Produces:` Rückgabeformate (MIME-Type)

Beispiel Server Code

```
HttpServer server =
        HttpServerFactory.create("http://localhost:8080/rest");
server.start();
JOptionPane.showMessageDialog(null, "Ende"); // Programm anhalten
server.stop(0);
```

Pfad für die Ressource: http://localhost:8080/rest/message

EntryPoint

Die Webservices werden automatisch gefunden und müssen nicht beim Server registriert werden.

Service Endpunkte festlegen (Service Code)

```
import javax.ws.rs.*;

@Path("message")
public class MessageResource {

    @GET                    // http://localhost:8080/rest/message
    @Produces(MediaType.TEXT_PLAIN)
    public String message() {
        return "Yea!";
    }

    @GET
    @Path("hello")          // http://localhost:8080/rest/message/hello
    @Produces(MediaType.TEXT_PLAIN)
    public String helloWorld() {
        return "HelloWorld!";
    }
}
```

Rückgabe von nicht-primitiven Datentypen MediaTypes (Service Code)

```
@XmlRootElement
public class ServerInfo {
    public String server;
}


@Path("message")
public class MessageResource {
    @GET
    @Path("serverinfo")
    @Produces(MediaType.TEXT_XML)
    public ServerInfo serverinfo() {
        ServerInfo info = new ServerInfo();
        info.server = System.getProperty("os.name");
        return info;
    }
}
```

Das Objekt info vom Typ ServerInfo wird automatisch mit JAXB in XML serialisiert.

Aufruf über Basispfad + "message" + "/" + "serverinfo“

http://localhost:8080/rest/message/serverinfo

Mehrere Rückgabetypen MediaTypes (Service Code)

```
@Path("message")
public class MessageResource {


    @GET
    @Path("serverinfo")
    @Produces({ MediaType.TEXT_XML, MediaType.APPLICATION_JSON })
    public ServerInfo serverinfo(){

        ...

    }
}
```

Client teilt den Wunschtyp mit (*Content Negotiation*)

Client: Testen des REST Services

- Test via Browser
- Insomnia
- curl
- selbst programmierter Client
  - > JavaScript (AJAX)
  - > Java
  - > …

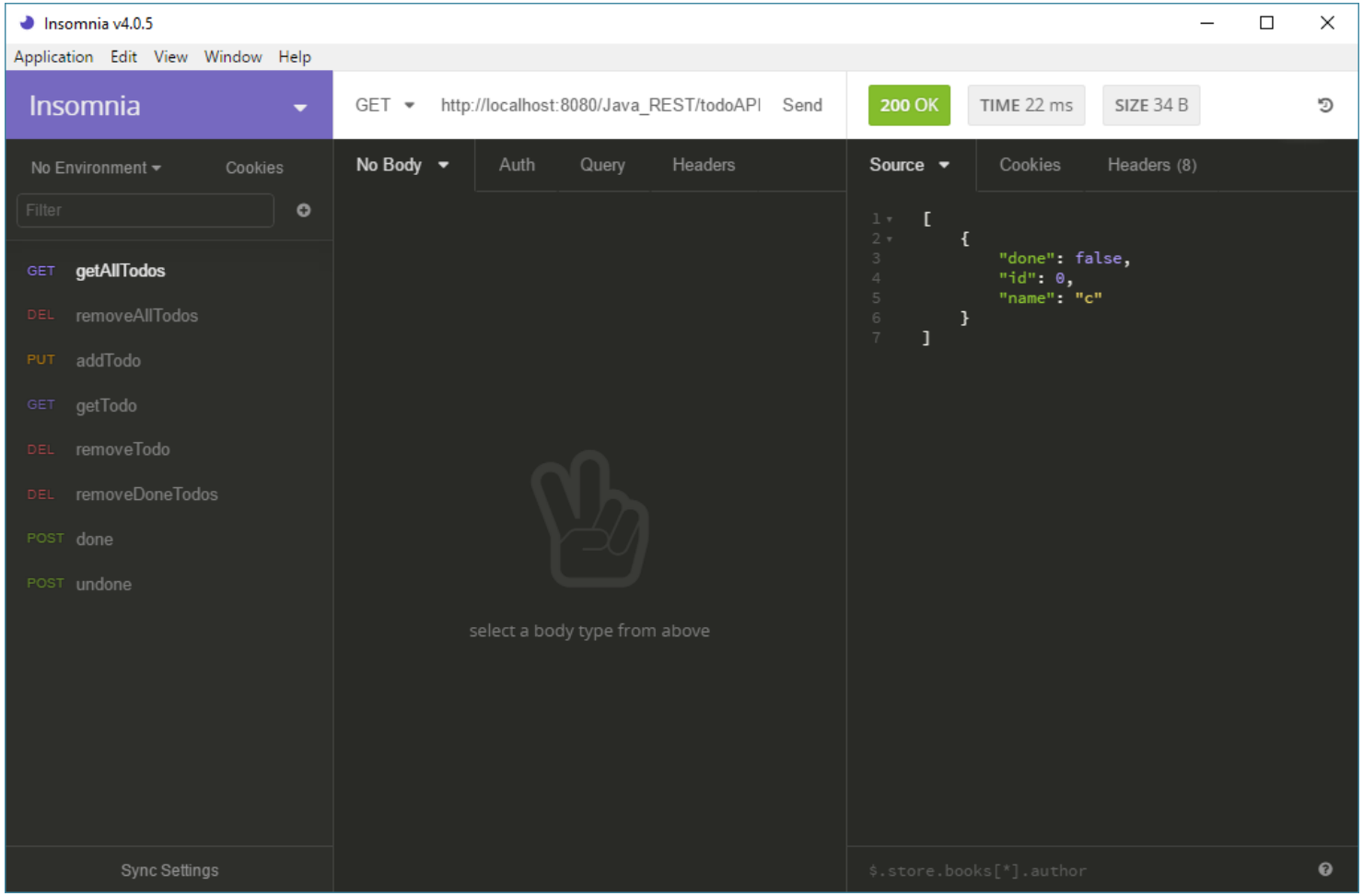
Client: Test via Browser

Nachteile:

- nur GET testbar!
- fordert immer HTML an

http://openbook.galileocomputing.de/java7/1507_13_002.html

Client: Insomnia



https://insomnia.rest/

Client: curl

Konsolentool

- Beispiel:

```
curl 'http://localhost:8080/Java_REST/todoAPI/todo' -X PUT -u
myUser:Secret -d "MyNewTask“
```

> gängige Optionen:
> -X: HTTP-Methode
> -u: Authentifizierung
> -d: zu übermittelnde Daten

Client: Java

```
import javax.ws.rs.core.*;
import com.sun.jersey.api.client.*;
import com.sun.jersey.api.client.WebResource.*;

WebResource resource =
          Client.create().resource("http://localhost:8080/rest");

Builder sb1 =
        resource.path("message").accept(MediaType.TEXT_PLAIN);

String yea = sb1.get(String.class);
System.out.println(yea);

Builder sb1 = resource.path("message").path("hello")
                      .accept(MediaType.TEXT_PLAIN);

String hello = sb1.get(String.class);
System.out.println(hello);
```

Rückgabetypen im Client

```
Builder sb1 = resource.path("message").path("serverinfo")
                                .accept(MediaType.APPLICATION_JSON);

ServerInfo info1 = sb1.get(ServerInfo.class);
System.out.println(info1.server);


Builder sb2 = resource.path("message").path("serverinfo")
                                    .accept(MediaType.TEXT_XML);

ServerInfo info2 = sb2.get(ServerInfo.class);
System.out.println(info2.server);
```

Rückgabetypen im Client (Exception bei nicht verfügbarem MediaType)

```
try {
    Builder sb3 = resource.path("message").path("serverinfo")
                          .accept(MediaType.TEXT_PLAIN);
    // PLAIN wird vom Server nicht angeboten -> Exception
    ServerInfo info3 = sb3.get(ServerInfo.class);
    System.out.println(info3.server);
} catch (UniformInterfaceException ex) {
    System.out.println(ex.getMessage());
}
// besser: Prüfen!
ClientResponse cr = sb2.get(ClientResponse.class);
if (cr.hasEntity()) {
    ServerInfo info4 = cr.getEntity(ServerInfo.class);
    System.out.println(info4.server);
} else {
    System.out.println("No Entity");
}
```

Parameterübergabe mit GET – Methodenüberladung (Client Code)

```
@GET @Produces(MediaType.TEXT_PLAIN)
public String message()
{ ... }
```

/rest/message
message();

```
@GET @Produces(MediaType.TEXT_PLAIN)
@Path("user/{user}")
public String message(
@PathParam("user") String user)
{ ... }
```

/rest/message/user/chris
message(„chris“);

```
@GET @Produces(MediaType.TEXT_PLAIN)
@Path("user/{user}/search/{search}")
public String message(
@PathParam("user") String user,
@PathParam("search") String search)
{ ... }
```

/rest/message/user/chris/search/kitesurfing
message(„chris“, „kitesurfing“);

Parameterübergabe mit GET – Methodenüberladung (Client Code)

```
// URLs dieses Formats werden durch Kaskaden von path() abgebildet.
Builder sb1 = resource.path("message").path("user").path("chris")
.accept(MediaType.TEXT_PLAIN);
System.out.println(sb1.get(String.class));
Builder sb2 = resource.path("message").path("user").path("chris")
.path("search").path("kitesurfing")
.accept(MediaType.TEXT_PLAIN);

System.out.println(sb2.get(String.class));

// Unschoene Alternative
Builder sb3 = Client
.create().resource(
"http://localhost:8080/rest/message/user/chris/search/kitesurfing")
.accept(MediaType.TEXT_PLAIN);


System.out.println(sb3.get(String.class));
```

Parameterübergabe mit PUT (Service & Client Code)

```
@PUT
@Path("userput/{user}")
@Consumes(MediaType.TEXT_PLAIN)
@Produces(MediaType.TEXT_PLAIN)
public String postMessage(
@PathParam("user") String user, String message) {
    return String.format("%s sendet '%s'%n", user, message);
}

-----------------------------------------------------------

Builder sb1 = resource.path("message").path("userput")
.path("chris").type(MediaType.TEXT_PLAIN).accept(MediaType.TEXT_PLAI
N);

System.out.println(sb1.put(String.class, "Hey Chris"));
```