Als Servlets werden Java-Klassen bezeichnet, deren Instanzen innerhalb eines Servlet-Containers Anfragen von Clients entgegen nehmen und beantworten. Der Inhalt der Antworten kann dabei dynamisch, also im Moment der Anfrage, erstellt werden und muss nicht bereits statisch (etwa in Form einer HTML-Seite) für den Webserver verfügbar sein. Servlets stellen somit das Java-Pendant zu CGI-Skripten oder anderen Konzepten dar, mit denen dynamisch Web-Inhalte erstellt werden können (PHP, Ruby on Rails usw.).

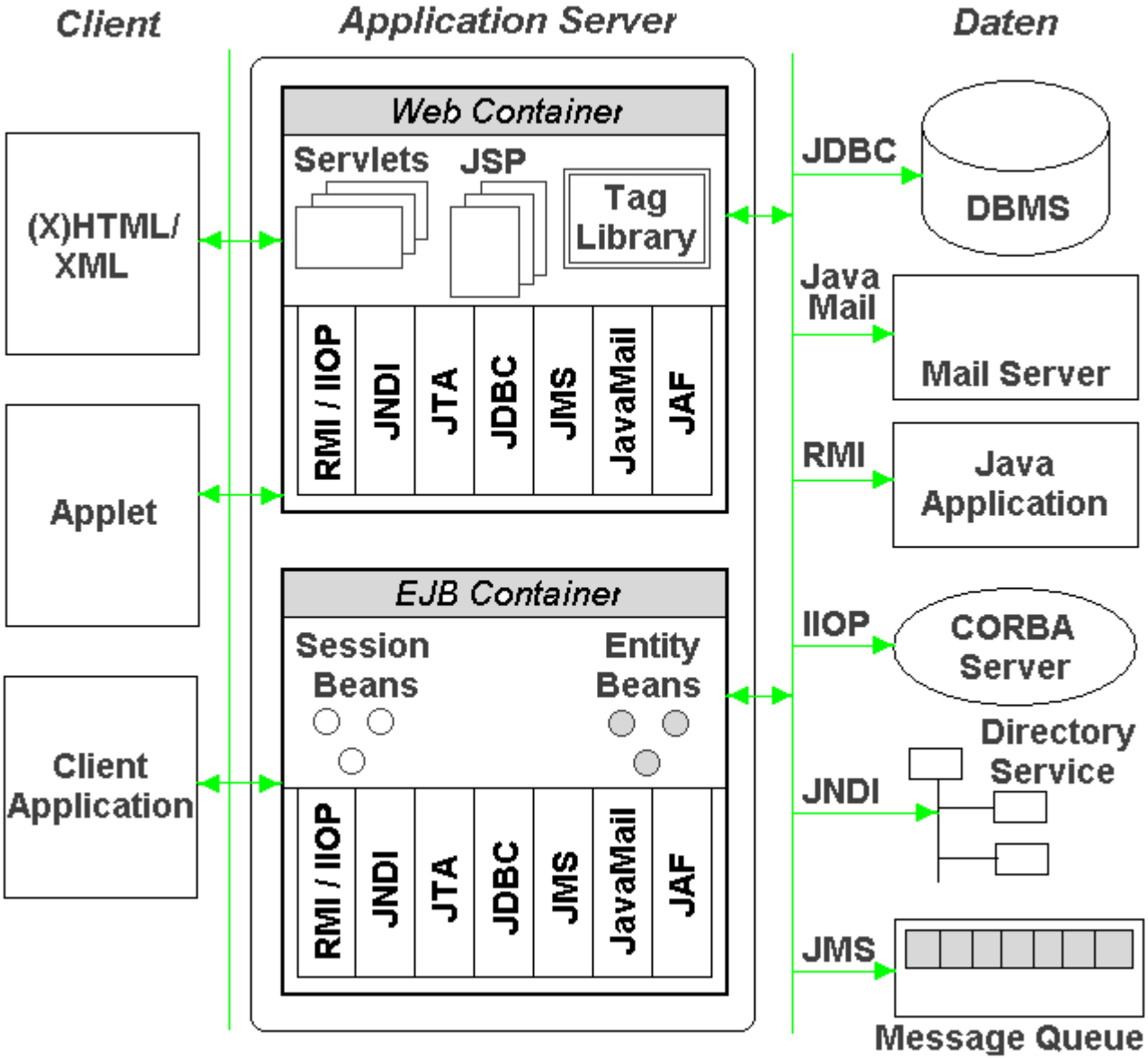
Eine Web Applikation in Java ist eine Sammlung von Servlets, Views, Klassen und weiteren Ressourcen gepackt und ablauffähig in mehreren Web-Containern von verschiedenen Herstellern.

Inhaltsverzeichnis

1. Einleitung

Web Applikationen und im speziellen Servlets sind ein Subset der Java serverseitigen API-Programmierung genannt Java Enterprise Edition 5 (Java EE 5). Die Java EE 5 Technologien sind entwickelt worden, um eine skalierbare und wartbare Infrastruktur für mittlere und grosse Organisationen (Enterprise) zur Verfügung zu stellen. Zu den Servlets kommen noch die JSP, JSF für die Views und Enterprise Java Beans (EJBs) hinzu, die für die Geschäftslogik eingesetzt werden.

Das folgende Diagramm fasst ein Teil der Java EE 5 Applikations-Architektur zusammen:



Die Module Web Applikationen I und II befassen sich mit den gezeigten Komponenten. Ein Tutorial, das die meisten Aspekte abdeckt, ist bei Sun Microsystem zu finden.

1.1 Aufbau einer Web Applikation

In Web Applikationen sind mehrere Komponenten beteiligt. Zu diesen gehören unter anderem:

- HTML-Dokumente
- Bilder, Audio- und Videodaten
- Servlets, Listener und Filter
- JSP-Seiten und Tag-Libraries sowie JavaServer Faces
- Java Beans (POJOs)

Diese Komponenten können auf jedem Server eingesetzt werden, der mit der Servlet-Version 2.2 oder höher kompatibel ist. Sorgfältig entworfene Web Applikationen können von Server zu Server verlagert werden, ohne dass irgendwelche Änderungen an den Komponenten vorgenommen werden müssen.

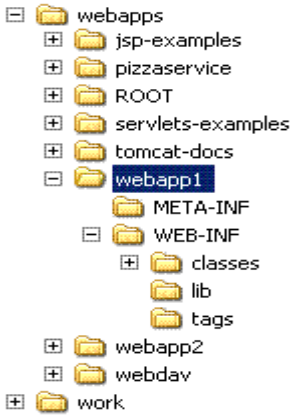
Eigentlich beziehen sich diese Komponenten auf eine Web Applikation, denn Web Applikationen umfassen natürlich auch die Umsetzung einer Web-basierten Applikation in Form von Web-Komponenten im Sinne der EE 5-Spezifikation (vergl. Web Applikationen II).

1.1.1 Deployment Descriptor

Die Servlet-Spezifikation legt neben der physikalischen Organisation der zugehörigen Dateien einer Web Applikation auch die Struktur des sogenannten Deployment Descriptors fest, der in Form einer Datei namens `web.xml` vorliegen muss. Diese Konfigurationsdatei wird für jede Web Applikation benötigt, die in einem Servlet Container eines Servers deployed (verteilt) wird.

Der Deployment Descriptor einer Web Applikation umfasst alle Konfigurationsdaten der Applikation. Er bildet die zentrale Schnittstelle und wird vollständig in der Datei `web.xml` abgebildet und im Verzeichnis `WEB-INF` abgespeichert.

Jeder Web Applikation wird ein logisches Wurzelverzeichnis (Context) zugeordnet. Alle Anfragen, deren Pfadangaben mit diesem Wurzelverzeichnis beginnen, werden an die Web Applikation weitergeleitet. Somit sieht die Verzeichnisstruktur einer einfachen Web Applikation folgendermassen aus (der Context ist `webapp1`):



Verzeichnisstruktur:

- **`/WEB-INF`**: Dieses Verzeichnis muss im Wurzelverzeichnis jeder Web Applikation vorhanden sein.
- **`/META-INF`**: Muss vorhanden sein, falls die Applikation als Archivdatei verteilt werden soll.
- **`/WEB-INF/web.xml`**: Enthält den Deployment Descriptor der Web Applikation.
- **`/WEB-INF/classes`**: In diesem Verzeichnis können beliebige übersetzte Java-Klassen der Web Applikation gespeichert werden.
- **`/WEB-INF/lib`**: Java-Archive der Web Applikation.
- **`/WEB-INF/tags`**: In diesem Verzeichnis können Tag-Dateien abgelegt werden (für JSP's).

Die Dateien im Verzeichnis `WEB-INF` und darunter sind nicht über HTTP erreichbar, daher stehen HTML-Dokumente, Bilder, CSS-Stylesheets usw. nicht in diesem Verzeichnis. Das gilt auch für JSP-Seiten, die über Links erreicht werden müssen.

1.2 Bilden und Verteilen einer Web Applikation

Web Applikationen können in Archiven zusammengefasst werden. Solche Archive werden Web Application ARchives bzw. `WAR`-Archive genannt. Die Dateien haben die Endung `war`. Zur Erzeugung eines `WAR`-Archivs wird das Programm `jar` verwendet.

Für das Bilden und Verteilen einer Web Applikation auf einen Web-Server (build and deploy) stehen verschiedene Werkzeuge zur Verfügung. Das älteste und deshalb wohl auch das meist eingesetzte Werkzeug ist Ant. Ant ist ein in Java geschriebenes Werkzeug zum automatisierten Erzeugen von Programmen aus Quelltext. Damit erfüllt es den gleichen Zweck wie das sehr verbreitete Programm make, nämlich die automatisierte Erstellung von installierbaren Software-Paketen aus existierendem Quelltext, Bibliotheken und sonstigen Dateien.

Ant ist Open Source, startete als Teil des Jakarta-Projekts und ist nun ein Apache-Top-Level-Projekt. Ant ist ein Akronym und steht für »Another Neat Tool« (engl. für »Noch ein hübsches Werkzeug«). Entwickelt wurde die erste Version von James Duncan Davidson, der 1999 ein Werkzeug wie make für Java benötigte, während er die erste J2EE-Referenz-Implementierung entwickelte. Der Name »Ant« steht auch für ein kleines Programm, das wie eine Ameise Grosses leisten kann. Davidson gilt auch als Vater von Apache Tomcat, dem Referenzserver mit integriertem Servlet-Container.

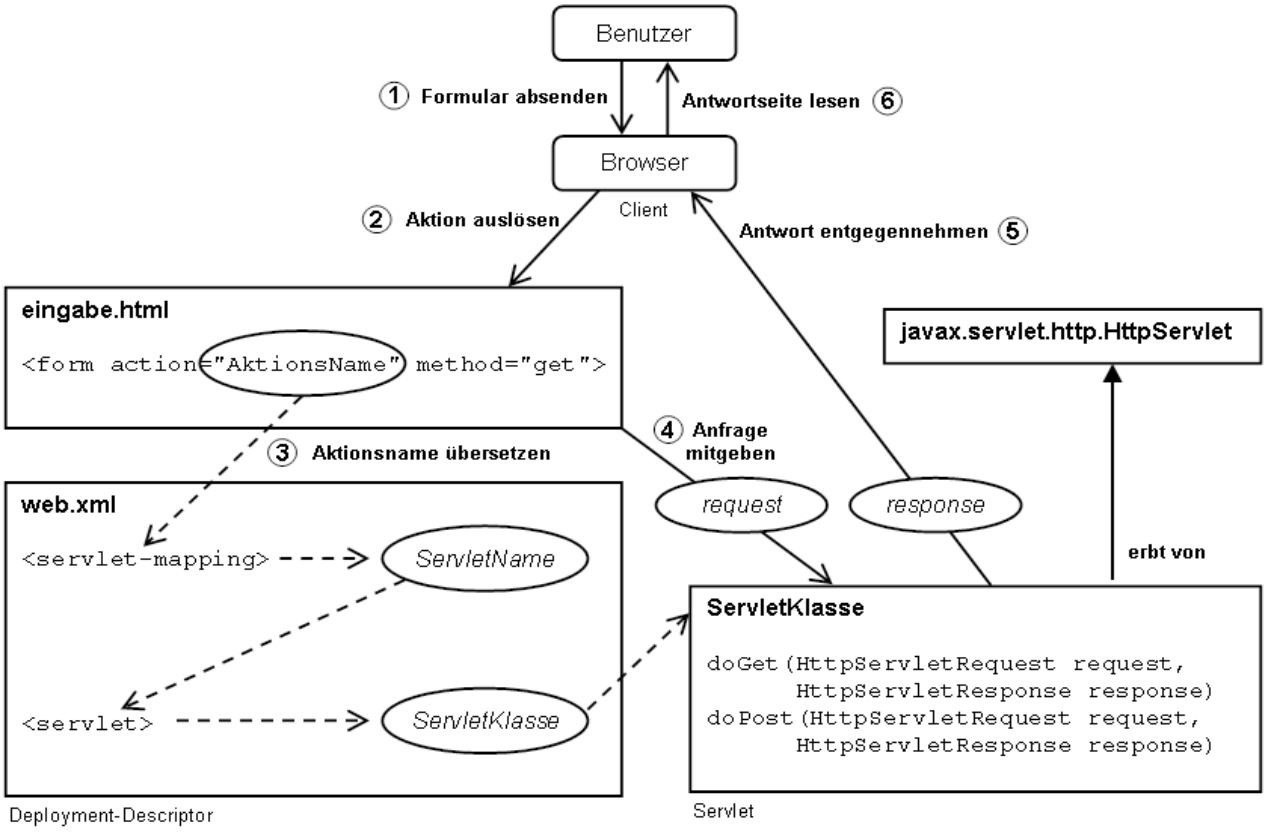
Das neueste Open-Source-Werkzeug auf diesem Markt ist Maven 3, das die klassischen Build-Skripte durch eine Projektkonfigurationsdatei ersetzt und zusätzlich eine Reihe nützlicher Merkmale zur Projektsteuerung anbietet. Maven 3 ist ein populäres Open-Source Build- und Projektmanagement Werkzeug für Java-Projekte. Es übernimmt die harte Arbeit des Build-Prozesses. Dabei benützt Maven 3 den deklarativen Ansatz, bei dem die Projektstruktur und deren Inhalt beschrieben wird im Gegensatz zu Ant oder

make, deren Ansatz Task-basiert ist.

Bei Maven 3 wird davon ausgegangen, dass der Build-Prozess in den meisten Projekten ähnlich ist und nicht jedes Mal neu programmiert werden muss. Die Build-Skripte werden durch mitgelieferte Plug-Ins und die Projektkonfiguration (Project Object Model POM) ersetzt. Die Build-Funktionalität enthalten die Plug-Ins. Die Projektkonfiguration (POM) wird in einer XML-Datei (`pom.xml`) definiert und enthält alle Informationen, die die Plug-Ins zur Build-Zeit benötigen. Zudem verwaltet Maven 3 die Abhängigkeiten eines Projekts von Drittbibliotheken (sog. Dependencies) in einem Repository und generiert Dokumentation und Reports über die Code-Qualität. In den Modulen Web Applikationen I und II wird mit Maven 3 gearbeitet, wobei auch Ant-Targets eingesetzt werden (innerhalb von Maven 3!). Die einführende Beschreibung über Maven 3 finden Sie im Thema 1 dieses Moduls.

2. Servlets

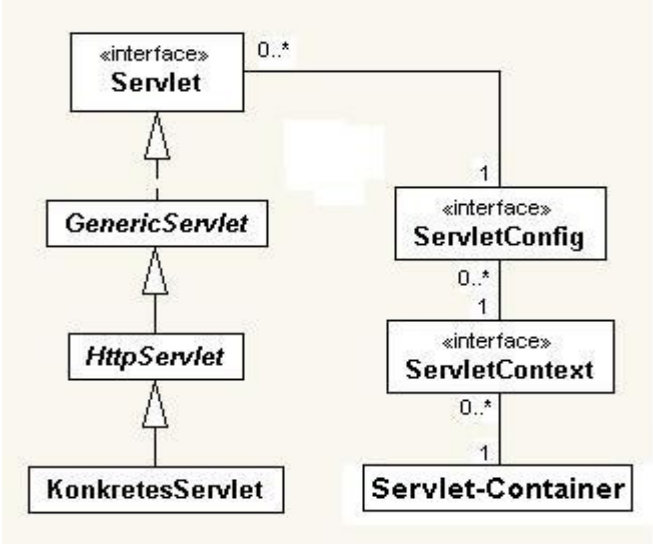
Als Servlet wird im Rahmen der Java Enterprise Edition 5 ein Java-Objekt bezeichnet, an das ein Webserver Anfragen seiner Clients delegiert, um die Antwort an den Client zu erzeugen. Der Inhalt dieser Antwort wird dabei erst im Moment der Anfrage generiert, und ist nicht bereits statisch (etwa in Form einer Hypertext Markup Language HTML-Seite) für den Webserver verfügbar. Servlets sind Instanzen von Java-Klassen, die von der durch die Spezifikation definierten Klasse `javax.servlet.HttpServlet` abgeleitet werden. Diese Instanzen werden bei Bedarf von einer Laufzeitumgebung, dem sogenannten Web-Container erzeugt und von ihm aus angesprochen. Der Servlet-Container (oder Servlet Engine) seinerseits kommuniziert mit dem Webserver. Das folgende Diagramm aus Wikipedia zeigt den Ablau einer Web Applikation mit Servlets:



Die Java Servlet Technologie wird im Java EE 5 Tutorial von Sun Microsystem im Kapitel 4 eingehend beschrieben. Im Java-Kultbuch »Java ist auch eine Insel« beschreibt das Kapitel 19 die Servlet-Technologie.

2.1 Servlet API

Die Servlet-API hat die Aufgabe, eine einfache Programmierschnittstelle für Web Applikationen bereitzustellen und technische Details zu verbergen. Dies wird erreicht, indem Abstraktionen für die wichtigsten Konzepte einer Applikation eingeführt werden. Die Servlet-API definiert unter anderem eine Schnittstelle für Applikationen auf der Basis von HTTP. Die wichtigsten Abstraktionen werden in Form von Interfaces definiert:



Die abstrakte Klasse `HttpServlet` implementiert das Grundgerüst einer Web Applikation. Für jede HTTP-Methode wie beispielsweise GET und POST wird eine Methode mit leerem Rumpf bereitgestellt. Der Servlet-Container ruft bei einer Anfrage die zu der HTTP-Methode gehörende Java-Methode auf (`doGet` oder `doPost`). Nebst diesen beiden Methoden können noch zwei weitere Methoden überschrieben werden, welche am Anfang (`init`) bzw. am Ende (`destroy`) des Lebenszyklus eines Servlets durch den Container aufgerufen werden.

Das Interface `ServletContext` bietet eine Sicht auf den ausführenden Container und die aktuelle Web Applikation. Konfiguriert wird diese durch den Deployment Deskriptor bzw. der Datei `web.xml`.

Das Interface `HttpServletRequest` (nicht dargestellt) bietet eine objektorientierte Sicht auf eine HTTP-Anfrage. Die Methoden in diesem Interface nehmen der Applikation die Aufgabe ab, die HTTP-Anfrage zu parsen und einzelne Teile zu dekodieren. Weiter werden einige höhere Konzepte wie Cookies, Sessions und Locales angeboten.

Das Interface `HttpServletResponse` (nicht dargestellt) ermöglicht den komfortablen Aufbau einer HTTP-Antwort. Die Ausgabe erfolgt gepuffert, d.h. vor dem erstmaligen Leeren des Puffers ist eine Änderung der Daten noch möglich. Die angebotenen Methoden erlauben es, einzelne HTTP-Header und Cookies einzufügen.

Die beschriebene Funktionalität wurde bereits in Version 2.1 der Servlet-API bereitgestellt. Neben diesen Grundlagen gibt es noch weitere Konzepte, welche das Erstellen grösserer

Applikationen erleichtern. Von zentraler Bedeutung ist dabei die Klasse `RequestDispatcher`. Sie ermöglicht Servlets, eingehende Anfragen an andere Ressourcen (z.B. Servlets oder Views) weiterzuleiten.

2.2 Lebenszyklus eines Servlets

Jedes Servlet hat den gleichen Lebenszyklus:

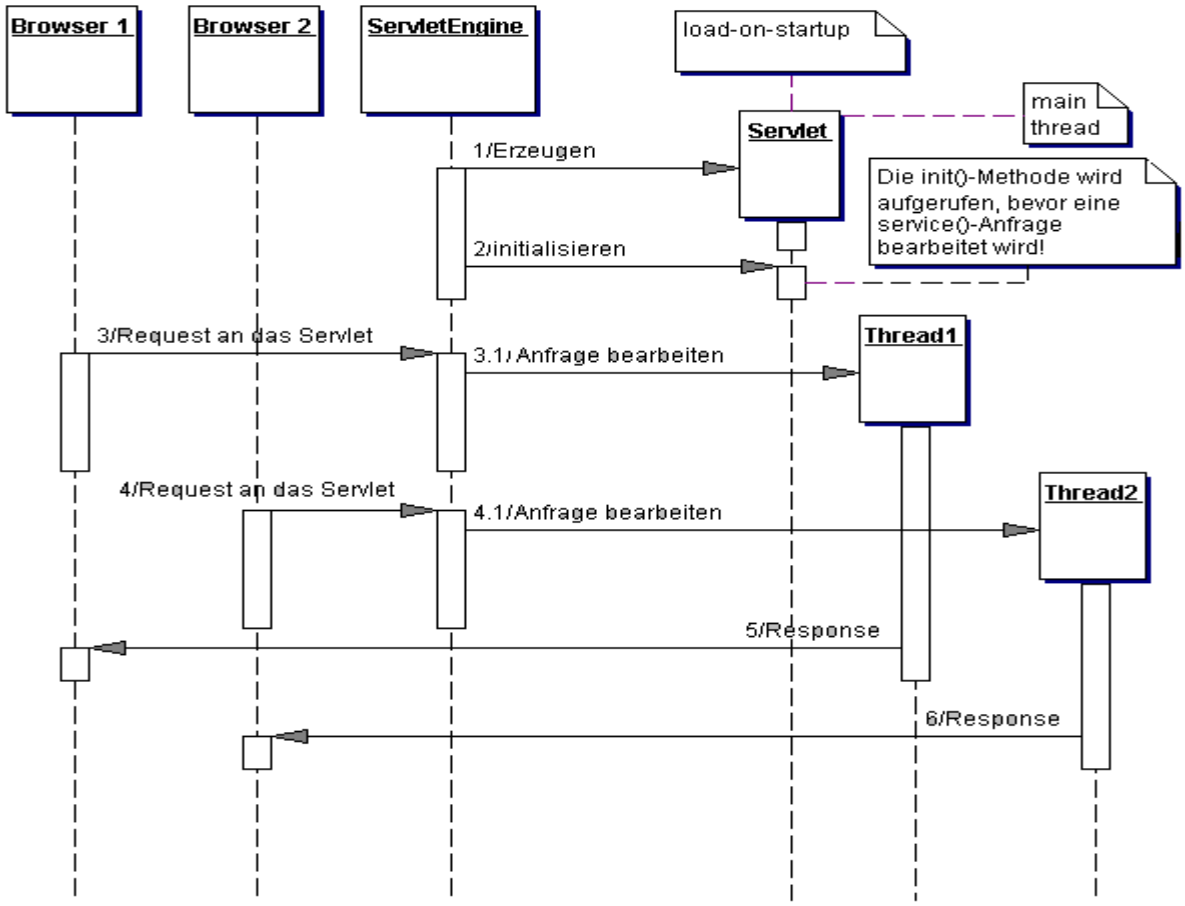
- Der Servletcontainer lädt und initialisiert das Servlet
- Das Servlet bearbeitet eine oder mehrere Anfragen
- Der Servletcontainer zerstört das Servlet

Dazu stehen folgende Methoden für die Implementierung zur Verfügung:

- Initialisierung: `init(ServletConfig config)`
- Bearbeitung: `doGet` bzw. `doPost` mit den formalen Parametern `HttpServletRequest request` und `HttpServletResponse response`
- Zerstörung: `destroy()`

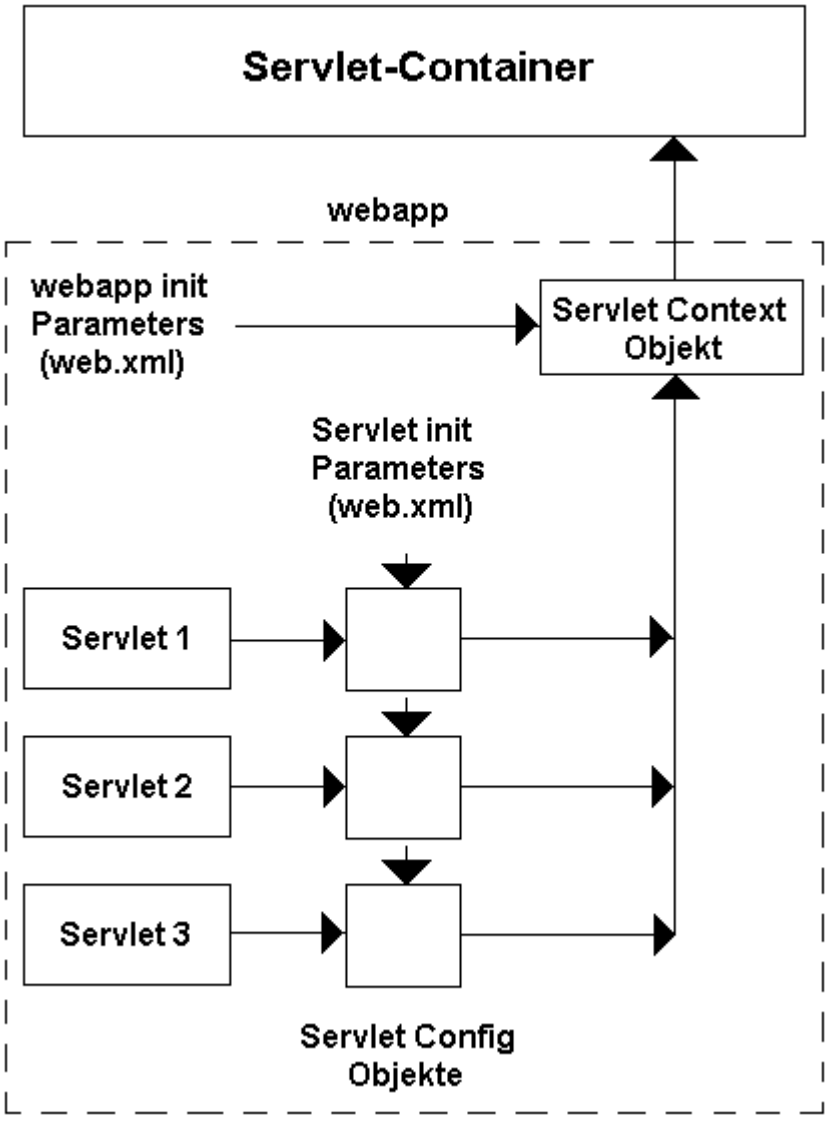
Diese Methoden werden vom Servlet-Container aufgerufen!

Das folgende Diagramm illustriert die Schritte, die ein Servlet durchläuft von der Erzeugung bis hin zu den Anfragen:



2.3 Servlet Context

Eine Applikation setzt sich im Normalfall aus mehreren Servlets zusammen. Es ist also ein Mechanismus notwendig, um gemeinsame Objekte allen Servlets zur Verfügung zu stellen. Dies erfolgt über eine gemeinsame Umgebung (Context), in welcher Objekte abgelegt oder gelesen werden können. Jedes Servlet erbt die Methode `getServletContext()` zum Zugriff auf den aktuellen Context. Über das erhaltene `ServletContext`-Objekt können Methoden zur Verwaltung der gemeinsamen Ressourcen aufgerufen werden. Damit lässt sich die Umgebung eines Servlets folgendermassen darstellen:

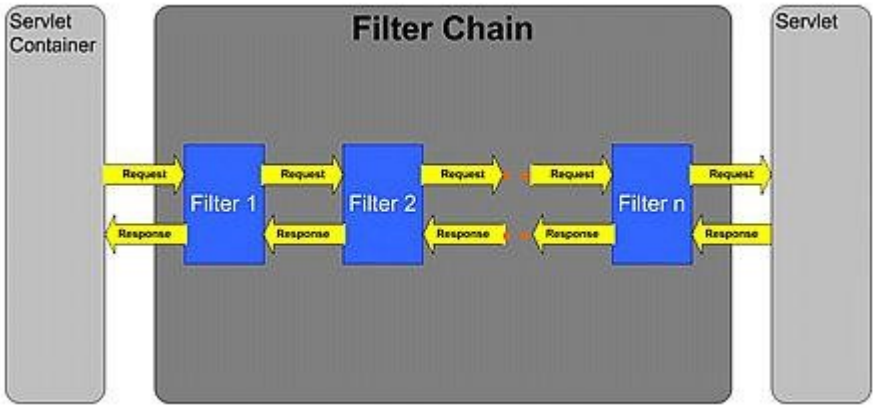


2.4 Sessions

Session werden erzeugt, um Anfragen von Clients (verbindungsloses HTTP-Protokoll) verbindungsorientiert durchzuführen. Sie können entweder von der Servlet-Container inaktiv gesetzt werden oder explizit über Methoden (z.B. `invalidate()`) beeinflusst werden. Das automatische Sessionmanagement ist in die Servlet-API integriert. Dieses wird über Session Ids geregelt. Jede Session bekommt eine im System eindeutige Id. Diese wird versucht über Cookies an den Browser zu schicken. Der Browser schickt bei jeder Anfrage an den Servlet-Container das Cookie wieder mit zurück. Erlaubt der Client keine Cookies, wird die Id an den URL angehängt (URL-Rewriting). Eine weitere Möglichkeit ist die Id in hidden-Felder auf dem Client-Browser zu plazieren.

2.5 Filter und Ereignisse

Zur besseren Aufteilung der Funktionalitäten wurden ab der Version 2.3 der Servlet-API zwei neue Konzepte eingeführt: Filter und Ereignisse. Die Ausführung von Filtern erfolgt automatisch und ist der Ausführung eines angefragten Servlets vorgelagert. Ereignisse für verschieden Komponenten werden vom Servlet-Container generiert. Filter können zu Ketten zusammengefügt werden:



Filter können Anfrage- und Antwortobjekte verändern und an den nächsten Filter oder an das angefragte Servlet weiterleiten. Weiter können sie auch Antwortobjekte erzeugen und direkt zurückgeben. Filter können zu einer Filterkette zusammengesetzt werden. Welche Filter in welcher Reihenfolge vor die Ausführung eines Servlets geschaltet werden, wird im Deployment Descriptor (`web.xml`) festgelegt. Dieser deklarative Ansatz erlaubt es, ohne Neucompilation Filter ein- bzw. auszuschalten. Typische Anwendungsgebiete von Filtern sind Dienste, die nur gelegentlich ausgeführt werden sollen (z.B. Debugging, Laufzeitmessungen) oder globale Dienste wie Verschlüsselung und Authentifizierung.

Vergleichbar zu dem Ereignismodell von Swing gibt es für Servlets sogenannte Lebenszyklus Ereignisse. An den wichtigsten Ausführungsstationen generiert der Servlet-Container Ereignisse und diese werden von Anwendern erstellten Callback-Methoden aufgefangen und verarbeitet. Zu diesem Zweck werden Ereignisklassen und Listener-Interfaces bereitgestellt. Solche Kombinationen gibt es beispielsweise für Context-, Session- und Request-Ereignisse.

3. Praxis anhand eines Gästebuches

Auf vielen Webseiten von Privatpersonen sind Gästebücher zu finden, in denen sich die Besucher eintragen können. Anhand einer solchen einfachen Web Applikation werden die theoretisch beschriebenen Konzepte im folgenden praktisch umgesetzt.

3.1 Gästebuch

Das Gästebuch besteht aus drei Views: »Startseite«, »Anzeigen« und »Eintragen«.

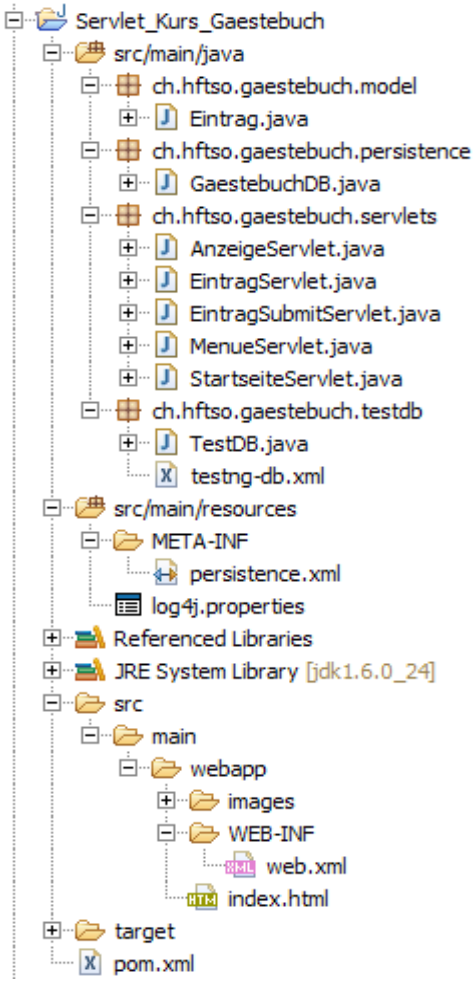
Startseite

Anzeigen

Name	Kommentar
Hans Müller	Es hat mich gefreut
Ruedi Baumann	Alles hat ein Ende!
Kurt Munter	Es lebe Java!

Eintragen

Die Verzeichnisstruktur des Gästebuches sieht folgendermassen aus:



Diese Struktur ist durch die Servlet Spezifikation und durch den Archetyp von Maven vorgegeben.

Das Klassenmodell des Gästebuches besteht aus der Klasse `Eintrag` :

```
@Entity
public class Eintrag implements Serializable {
  private static final long serialVersionUID = -4363176533291352510L;

  private long id;
  private String name;
  private String kommentar;

  public Eintrag() {
    name = "";
    kommentar = "";
  }

  public Eintrag(String name,String kommentar) {
    this.name = name;
    this.kommentar = kommentar;
  }

  @Id
  @GeneratedValue
  public long getId() {
    return id;
  }

  public void setId(long id) {
    this.id = id;
  }

  // Getter- und Setter-Methoden der Instanzvariablen
  // ...
```

Das Gästebuch benötigt eine Datenbank zum Speichern der Einträge. Dazu wird die HSQLDB eingesetzt. Die Anbindung übernimmt die Klasse `GaestebuchDB`, deren Code selbst sprechend im Arbeitsmaterial zu finden ist. Die Parameter für die Anbindung sind in der Persistenzdatei `persistence.xml` eingetragen:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="gaestebuchDatabase"
                    transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.cache.provider_class"
                value="org.hibernate.cache.HashtableCacheProvider" />
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.use_sql_comments" value="true" />
      <property name="hibernate.connection.driver_class"
                value="org.hsqldb.jdbcDriver"/>
      <property name="hibernate.connection.url"
```

```
                value="jdbc:hsqldb:hsql://localhost"/>
      <property name="hibernate.connection.password" value=""/>
      <property name="hibernate.connection.username" value="sa"/>
    </properties>
  </persistence-unit>
</persistence>
```

Für die Implementierung der Funktionalität werden 5 Servlets eingesetzt:

- `MenueServlet.java`
- `StartseiteServlet.java`
- `AnzeigeServlet.java`
- `EintragServlet.java`
- `EintragSubmitServlet.java`

Diese werden im folgenden eingehender betrachtet. Zuerst muss jedoch der Deployment Descriptor (`web.xml`) des Gästebuches implementiert werden:

```
<web-app version="2.5"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>Gaestebuch mit Servlets</display-name>

  <servlet>
   <servlet-name>Menue</servlet-name>
   <servlet-class>ch.hftso.gaestebuch.servlets.MenueServlet</servlet-class>
  </servlet>

  <!-- Weitere 4 Servlet Definitionen
  ... -->
  <servlet-mapping>
    <servlet-name>Menue</servlet-name>
    <url-pattern>/menue</url-pattern>
  </servlet-mapping>
  <!-- Weitere 4 Mapping Definitionen
  ... -->
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Für jede `servlet` Definition muss ein entsprechendes `servlet-mapping` angegeben werden, über welches das Servlet aufgerufen werden kann. Dabei muss das Attribut `servlet-name` übereinstimmen.

Nun können die Servlets implementiert werden. Sie müssen von der Klasse `HttpServlet` abgeleitet werden! Das Servlet `MenueServlet` beinhaltet den Banner und die 3 Links. Die Implementation sieht folgendermassen aus:

```
public class MenueServlet extends HttpServlet {
  private static final long serialVersionUID = 1075641124581324959L;

  @Override
  public void doGet(HttpServletRequest request,
```

```
                    HttpServletResponse response)
    throws ServletException, IOException{

    // Fuer die GET-Anfrage  gilt die gleiche Verarbeitung
    // wie fuer die POST-Anfrage
    doPost(request,response);
  }

  @Override
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException{
    // MIME-Typ
    response.setContentType("text/html;charset=utf-8");
    // Zeichensatz
    response.setCharacterEncoding("utf-8");
    request.setCharacterEncoding("utf-8");
    // HTML-Menueseite aufbereiten
    StringBuffer buffer = new StringBuffer();
    buffer.append("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
      Transitional//EN\" \"http://www.w3.org/TR/html4/loose.dtd\">");
    buffer.append("<html><head>");
    buffer.append(
    "<meta http-equiv='content-type' content='text/html; charset=utf-8'>");
    buffer.append("<title>Gästebuch</title></head>");
    buffer.append("<body><div align='center'>");
    buffer.append("<table width='850px'><tr><td colspan='3'>");
    buffer.append("<img src='" + request.getContextPath());
    buffer.append("/images/banner.png' alt='Banner' />");
    buffer.append("</td></tr>");
    buffer.append("<td width='25%'><a href='start'>STARTSEITE</a></td>");
    buffer.append("<td width='25%'><a href='anzeige'>ANZEIGEN</a></td>");
    buffer.append("<td width='25%'><a href='eintrag'>EINTRAGEN</a></td>");
    buffer.append("</tr></table><hr width='850px' />");
    buffer.append("</div>");

    // Holen des Schreibers
    PrintWriter out = response.getWriter();
    out.println(buffer.toString());
  }
}
```

Aufrufe über die `doGet`-Methode (z.B. über Links) werden auf die `doPost`-Methode (Aufrufe über Formulare) umgeleitet. An beide Methoden werden zwei Objekte als Parameter übergeben: `request` (`HttpServletRequest`) enthält die Client-Informationen und `response` (`HttpServletResponse`) dient für Informationen, die an den Client zu senden sind. Für die Antwort an den Client muss der MIME-Typ angegeben werden und ein »Schreiber« (`PrintWriter`) muss geholt werden. Danach muss die anzuzeigende Seite aufgebaut werden.

Der Aufruf an das Servlet Menue erfolgt über den URL `http://localhost:8080/gaestebuch/menue`, wobei `localhost` den DNS Namen darstellt (IP Adresse: 127.0.0.1) und `8080` den Port. Der URI `/gaestebuch` ist der Name des WAR-Archivs (`gaestebuch.war`), das beim Tomcat Web-Server in das Verzeichnis `webapps` geladen wird. Letztendlich stellt der URI `/menue` das `url-pattern` dar, mit welchem das Servlet `MenueServlet` aufgerufen wird.

Das Menü ist Teil jeder Seite des Gästebuches und muss in der Implementation entsprechend berücksichtigt werden:

```
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher("/menue");
    if (dispatcher != null) {
      dispatcher.include(request, response);
    }
```

Der `dispatcher` ermöglicht, einen externen Inhalt (`Menue Servlet`) einzubinden. Er kann aber auch dazu verwendet werden, Anfragen weiterzuleiten (Beispiel `EintragSubmitServlet`):

```
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher("/eintrag");
    if (dispatcher != null) {
      dispatcher.forward(request, response);
    }
```

Der `RequestDispatcher` übergibt mit der Methode `forward` die Steuerung vollständig an den zugeordneten URI (`/eintrag`) ab und mit der Methode `include` wird der Inhalt des zugeordneten URI (`/menu`) eingebunden.

Um ein Objekt vom `ServletContext` zu erhalten, kann die Methode `getServletContext()` aufgerufen werden. Dieses Objekt wird von allen Servlets gemeinsam genutzt. Es enthält die Methoden `setAttribute` und `getAttribute`, kann also als Datenspeicher genutzt werden.

Klickt der Benutzer des Gästebuches auf den Menülink »EINTRAGEN«, kann er seinen Namen und seinen Kommentar eingeben. Über den Button »Eintragen« schickt er seine Daten an den Server. Die Implementation (`EintragSubmitServlet`) sieht folgendermassen aus:

```
public class EintragSubmitServlet extends HttpServlet {
  private static final long serialVersionUID = 2696533595306834088L;

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

    // Fuer die GET-Anfrage  gilt die gleiche Verarbeitung
    // wie fuer die POST-Anfrage
    doPost(request,response);
  }

  @Override
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException{

    // Zeichensatz weil Menue nicht eingebunden wird!
    request.setCharacterEncoding("utf-8");

    String name = request.getParameter("name");
```

```
    String kommentar = request.getParameter("kommentar");

    GaestebuchDB.getInstance().eintragen(new Eintrag(name,kommentar));

    // Holen des Dispatcher; anzeigen der Anzeigeseite
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher("/anzeige");
    if (dispatcher != null) {
      dispatcher.forward(request, response);
    }
  }
}
```

Über das `request`-Objekt werden die vom Benutzer eingegebenen Daten über die Methode `getParameter` geholt. Die Parameter `name` und `kommentar` sind die Namen (keys) der Eingabefelder im `EintragServlet`:

```
    buffer.append("<table border='0'><tr><td>Name: </td>");
    buffer.append("<td><input type='text' name='name'></td></tr>");
    buffer.append("<tr><td>Kommentar: </td>");
    buffer.append("<td><input type='text' name='kommentar'></td>");
```

Das Servlet `AnzeigeServlet` zeigt die im Gästebuch enthaltenen Daten an. Ein Ausschnitt aus der Implementation des `AnzeigeServlet`, das für die Darstellung verantwortlich ist, zeigt das folgende Listing:

```
    // Anzeigen
    buffer.append("<table border='1' bordercolor='blue'>");
    buffer.append("<tr><th>Name</th><th>Kommentar</th></tr>");
    // Alle Eintraege
    for (Eintrag eintrag : eintraege) {
      buffer.append("<tr><td>" + eintrag.getName() + "</td>");
      buffer.append("<td>" + eintrag.getKommentar() + "</td></tr>");
    }
    buffer.append("</table></div></body></html>");
```

Die Daten werden aus der Datenbank geholt und iteriert. Auch dieses Beispiel zeigt, wie das Aufbereiten der anzuzeigenden Seite aufwändig zu programmieren ist!

3.2 Gästebuch mit Filter und Listener

Das Gästebuch wird mit einem Filter für die Protokollierung erweitert, der eine Meldung in die Standardausgabe schreibt, wenn das zugehörige Servlet (`AnzeigeServlet`) aufgerufen wird:

```
127.0.0.1 versucht den Zugriff auf http://localhost:8080/gaestebuch/anzeige
um Mon Mar 03 10:48:39 CET 2008 protokolliert von Protokollierung
```

Für diese Protokollierung muss eine neue Klasse `GaestebuchFilter` programmiert werden, die das Interface `Filter` implementiert:

```
public class GaestebuchFilter implements Filter {
  private static Logger logger = Logger.getLogger(GaestebuchFilter.class);
  protected FilterConfig config;
  private ServletContext context;
  private String filterName;
```

```
  /**
   * Einmaliges initialisieren des Filters
   * @param config enthaelt die Konfigurationsdaten
   */
  public void init(FilterConfig config) throws ServletException {
    this.config = config;
    context = config.getServletContext();
    filterName = config.getFilterName();
  }

  /**
   * Enthaelt das Filterungsverhalten.
   * @param request Vollzugriff auf die eintreffenden Informationen
   * @param response Antwortaufbereitung
   * @param chain Aufruf des Servlets oder JSP bzw. des naechsten Filters
   * @throws IOException, ServletException
   */
  public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse res = (HttpServletResponse) response;
    StringBuffer sb = new StringBuffer();
    sb.append(req.getRemoteHost());
    sb.append(" versucht den Zugriff auf ");
    sb.append(req.getRequestURL());
    sb.append(" um ");
    sb.append(new Date());
    sb.append(" protokolliert von ");
    sb.append(filterName);
    context.log(sb.toString());
    logger.debug(sb.toString());
    logger.debug(res.getCharacterEncoding());
    chain.doFilter(request,response);
  }

  /**
   * Verwendung des Filterobjekt wird beendet.
   */
  public void destroy() {
    logger.debug("Filter beendet!!!");
  }
}
```

Wenn ein Servlet aufgerufen wird, ruft der Servlet-Container zuerst die Filter Methode `doFilter` auf, die die gezeigte Protokollierung generiert. Nachdem die Meldung ausgegeben wurde, ruft der Filter die Methode `doFilter` des `FilterChain`-Objekts auf, um das verlangte Servlet aufzurufen. Das einzige filterbezogene Objekt, das eine Methode für den Zugriff auf den ServletContext hat, ist `config` (`FilterConfig`) mit der Methode `getServletContext`. Dieses Objekt wird vom Servlet-Container nur bei der Initialisierung übergeben und muss daher gespeichert werden.

Der Protokollierungsfilter muss im Deployment Descriptors (`web.xml`) definiert werden:

```
  <filter>
    <filter-name>Protokollierung</filter-name>
    <filter-class>
```

```
      ch.hftso.gaestebuch.servlets.filter.GaestebuchFilter
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>Protokollierung</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Mit Filtern können allgemeine Verhaltensweisen von Komponenten auf modulare und wiederverwendbare Art gekapselt werden. Der Zugriff auf den Code einer Komponente kann mittels Filter geschützt werden. Ressourcen können über Filter geändert werden.

Zusätzlich zu Filtern kann das Gästebuch mit Listeners ausgestattet werden. Ein Listener ermöglicht eine globale Steuerung für den Lebenszyklus einer Applikation. Er kann unter anderem auf folgende Ereignisse reagieren:

- `ServletContextListener`: Dieser Listener wird benachrichtigt, wenn der Servlet Context initialisiert und beendet wird.
- `ServletContextAttributeListener`: Dieser Listener wird benachrichtigt, wenn im Servlet Context Attribute hinzugefügt, gelöscht oder ersetzt werden.
- `HttpSessionListener`: Dieser Listener wird benachrichtigt, wenn Session-Objekte erzeugt, vernichtet oder durch Timeout beendet werden.
- `HttpSessionAttributeListener`: Dieser Listener wird benachrichtigt, wenn in irgend einer Session Attribute hinzugefügt, gelöscht oder ersetzt werden.

Im Falle des Gästebuches wird ein `ServletContextListener` eingebaut. Für diese Aufgabe muss eine neue Klasse `GaestebuchListener` eingebaut werden, die das Interface `ServletContextListener` implementiert:

```
public class GaestebuchListener implements ServletContextListener {
  private static Logger logger =
      Logger.getLogger(GaestebuchListener.class);

  /**
   * Die Web-Applikation wurde gerade geladen und
   * der Servlet-Context initialisiert.
   * @param event Servlet-Context initialisiert
   */
  public void contextInitialized(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    context.log("GaestebuchListener: contextInitialized");
    logger.debug("Applikation gestartet: " + context.getContextPath());
  }

  /**
   * Die Web-Applikation wurde gerade geschlossen und
   * der Servlet-Context wird gleich beendet.
   * @param event Servlet-Context beendet
   */
  public void contextDestroyed(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    context.log("GaestebuchListener: contextDestroyed");
    logger.debug("Applikation beendet: " + context.getContextPath());
  }
}
```

Wenn die Gästebuch Web Applikation vom Servlet-Container initialisiert wird (`contextInitialized`), wird das entsprechend geloggt:

```
Applikation gestartet: /gaestebuchfl
```

Wird die Gästebuch Applikation auf dem Server gelöscht (`contextDestroyed`), wird das wiederum geloggt:

```
Applikation beendet: /gaestebuchfl
```

Auch die Listener müssen im DeploymentDescriptor (`web.xml`) definiert werden:

```
  <listener>
    <listener-class>
      ch.hftso.gaestebuch.servlets.listener.GaestebuchListener
    </listener-class>
  </listener>
```

Die Listener ermöglichen, den Lebenszyklus einer Applikation zu überwachen.

4. Zusammenfassung

Zur Erleichterung der Portierung von Web Applikationen wurde in Version 2.2 der Servlet-Spezifikation der Begriff Web Applikation eingeführt. Damit wird eine logische Struktur für das Zusammenspiel aller Komponenten einer Web Applikation festgelegt. Die Spezifikation legt neben der physikalischen Organisation der zugehörigen Dateien auch die Struktur des Deployment Descriptors fest.

Auch das Archivierungsformat `war` auf der Basis von JAR-Archiven wurde definiert. Dieses beschreibt die Archivierung und die Komprimierung aller Dateien einer Web Applikation. Dies ermöglicht die Speicherung und Verteilung einer kompletten Web Applikation in einer einzigen Datei mit der Endung `war`. Es ist die Aufgabe des Servlet-Containers eines Servers, dieses `war`-Archiv entsprechend zu entpacken.

Servlets stellen damit im Rahmen der EE 5-Spezifikation das Pendant zu einem CGI-Skript (Common Gateway Interface) oder anderen Konzepten, mit denen dynamisch Web-Inhalte erstellt werden können (PHP usw.) dar. Die Servlet-API hat die Aufgabe, eine einfache Programmierschnittstelle für eine Applikation bereitzustellen und technische Details zu verbergen. `RequestDispatcher` ermöglicht Servlets, eingehende Anfragen an andere Ressourcen (z.B. Servlets oder JSP's) weiterzuleiten. Das Sitzungskonzept (Sessions) erlaubt trotz dem zustandslosen HTTP-Protokoll, zustandsbehaftete Objekte wie beispielsweise Warenkörbe umzusetzen.

Zur besseren Aufteilung der Funktionalitäten wurden in Version 2.3 der Servlet-API zwei neue Konzepte eingeführt: Filter und Ereignisse. Die Ausführung von Filtern erfolgt automatisch und ist der Ausführung eines angefragten Servlets vorgelagert. Ereignisse für verschieden Komponenten werden vom Servlet-Container generiert.

Literatur

Hall Marty, Brown Larry: *Core Servlets und JavaServer Pages* (deutsch); Markt + Technik Verlag, München 2004

Hall Marty, Brown Larry: Core Servlets and JavaServer Pages (engl); Free Online Version of Second Edition

Hall Marty: More Servlets and JavaServer Pages (engl); Free Online Version in PDF

Ullenboom, Christian: Java ist auch eine Insel; Kapitel 19 . 7. Auflage des Java-Kultbuches

Build a Web Application Using JPA

How to do it...

1. create DB (rechner)
2. copy Project
3. edit pom.xml (Name, Artifactid)
4. add necessary Dependencies
    1. org.hsqldb hsqldb X.Y.Z
    2. org.hibernate hibernate-entitymanager X.Y.Z
5. add JPA 2.0 to Project Facet
6. configure persistence.xml
    1. Name: rechnerDatabase
    2. PersistenceProvider: org.hibernate.ejb.HibernatePersistence
    3. Connection
        1. Transaction Type: Resource local
        2. Driver: org.hsqldb.jdbcDriver
        3. URL: jdbc:hsqldb:hsql://localhost/rechner
        4. User: sa
7. Define Model as Entity
    1. @Entity
    2. import javax.persistence.Entity
    3. add id ‚private long id‘ @Id @GeneratedValue
    4. .....
8. Add Entity (Class) to persistence.xml
    1. Managed Classes
        1. ch.hftm.rechner.Rechner
9. 9. ....

Zuletzt geändert: Donnerstag, 7. Mai 2015, 13:10

4 Understanding and Using Servlet Filters

When the servlet container calls a method in a servlet on behalf of the client, the HTTP request that the client sent is, by default, passed directly to the servlet. The response that the servlet generates is, by default, passed directly back to the client, with its content unmodified by the container.

Alternatively, you can use servlet filters to preprocess Web application requests and postprocess server responses. Filters were introduced in "When to Use Filters for Pre-Processing and Post-Processing", and are described in the following sections:

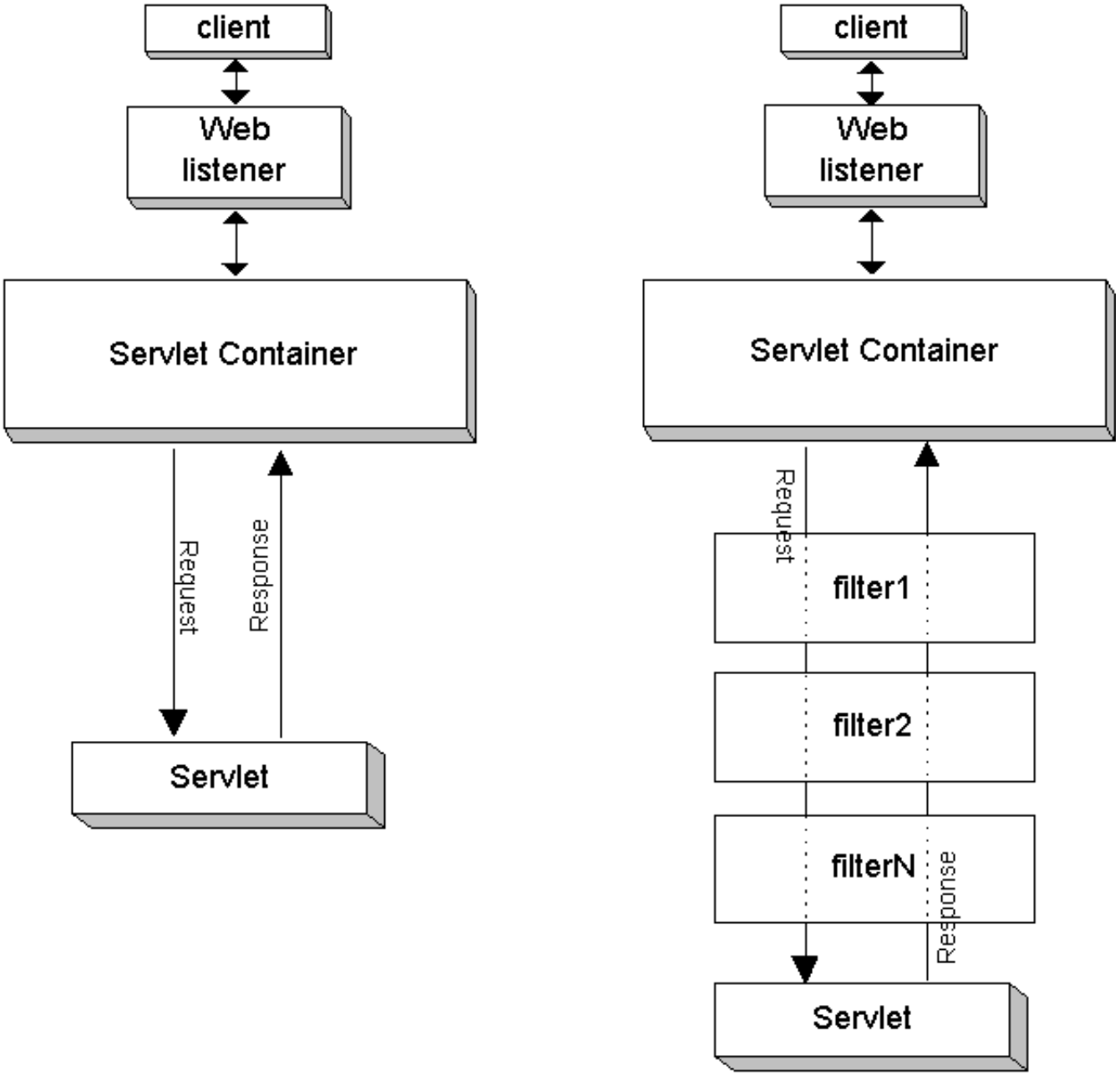
Overview of How Filters Work

This section provides an overview of the following topics:

How the Servlet Container Invokes Filters

Figure 4-1 shows, on the left, a scenario in which no filters are configured for the servlet being requested. On the right, several filters (1, 2,..., N) have been configured.

Figure 4-1 Servlet Invocation with and without Filters



Description of "Figure 4-1 Servlet Invocation with and without Filters"

Each filter implements the `javax.servlet.Filter` interface, which includes a `doFilter()` method that takes as input a request and response pair along with a filter chain, which is an instance of a class (provided by the servlet container) that implements the `javax.servlet.FilterChain` interface. The filter chain reflects the order of the filters. The servlet container, based on the configuration order in the `web.xml` file, constructs the chain of filters for any servlet or other resource that has filters mapped to it. For each filter in the chain, the filter chain object passed to it represents the remaining filters to be called, in order, followed by the target servlet.

The `FilterChain` interface also specifies a `doFilter()` method, which takes a request and response pair as input and is used by each filter to invoke the next entity in the chain.

Also see "Standard Filter Interfaces".

If there are two filters, for example, the key steps of this mechanism would be as follows:

1. The target servlet is requested. The container detects that there are two filters and creates the filter chain.
2. The first filter in the chain is invoked by its `doFilter()` method.
3. The first filter completes any preprocessing, then calls the `doFilter()` method of the filter chain. This results in the second filter being invoked by its `doFilter()` method.
4. The second filter completes any preprocessing, then calls the `doFilter()` method of the filter chain. This

results in the target servlet being invoked by its `service()` method.

5. When the target servlet is finished, the chain `doFilter()` call in the second filter returns, and the second filter can do any postprocessing.
6. When the second filter is finished, the chain `doFilter()` call in the first filter returns, and the first filter can do any postprocessing.
7. When the first filter is finished, execution is complete.

None of the filters are aware of their order. Ordering is handled entirely through the filter chain, according to the order in which filters are configured in `web.xml`.

Typical Filter Actions

The following are among the possible actions of the `doFilter()` method of a filter:

- Create a wrapper for the request object to allow input filtering. Process the content or headers of the request wrapper as desired.
- Create a wrapper for the response object to allow output filtering. Process the content or headers of the response wrapper as desired.
- Pass the request and response pair (or wrappers) to the next entity in the chain, using the chain `doFilter()` method. Alternatively, to block request processing, do *not* call the chain `doFilter()` method.

Any processing you want to occur before the target resource is invoked must be prior to the chain `doFilter()` call. Any processing you want to occur after the completion of the target resource must be after the chain `doFilter()` call. This can include directly setting headers on the response.

Note that if you want to preprocess the request object or postprocess the response object, you cannot directly manipulate the original request or response object. You must use wrappers. When postprocessing a response, for example, the target servlet has already completed and the response could already be committed by the time a filter would have a chain to do anything with the response. You must pass a response wrapper instead of the original response in the chain `doFilter()` call. See "Using a Filter to Wrap and Alter the Request or Response".

Standard Filter Interfaces

A servlet filter implements the `javax.servlet.Filter` interface. The main method of this interface, `doFilter()`, takes a `javax.servlet.FilterChain` instance, created by the servlet container to represent the entire chain of filters, as input. The initialization method of the `Filter` interface, `init()`, takes a filter configuration object, which is an instance of `javax.servlet.FilterConfig`, as input. This section briefly describes the methods specified in these interfaces.

For additional information about the interfaces and methods discussed here, refer to the Sun Microsystems Javadoc for the `javax.servlet` package, at:

http://java.sun.com/j2ee/1.4/docs/api/index.html

Methods of the Filter Interface

The `Filter` interface specifies the following methods to implement in your filters:

- `void init(FilterConfig filterConfig)`

  The servlet container calls `init()` as a filter is first instantiated and placed into service. This method takes a `javax.servlet.FilterConfig` instance as input, which the servlet container uses to pass information to the filter during the initialization. Include any special initialization requirements in your implementation. Also see "Methods of the FilterConfig Interface".

- `void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`

This is where your filter processing occurs. Each time a target resource (such as a servlet or JSP page) is requested, where the target resource is mapped to a chain of one or more filters, the servlet container calls the `doFilter()` method of each filter in the chain, in order according to `web.xml` filter configurations. (See "Construction of the Filter Chain".) Within the `doFilter()` processing of a filter, invoke the `doFilter()` method on the filter chain object that is passed in to the `doFilter()` method of the filter. (An exception to this is if you want to block request processing.) This is what leads to invocation of the next entity in the chain (either the next filter, or the target servlet if this is the last filter in the chain) after a filter has completed.

- `destroy()`: The servlet container calls `destroy()` after all execution of the filter has completed (all threads of the `doFilter()` method have completed, or a timeout has occurred) and the filter is being taken out of service. Include any special cleanup requirements in your implementation.

Method of the FilterChain Interface

The `FilterChain` interface specifies one method:

- `void doFilter(ServletRequest request, ServletResponse response)`

  Invoking this method, which you do from the `doFilter()` method of a filter, causes the next entity in the chain to be invoked—either the next filter, or the target resource (such as a servlet or JSP page) if this method is called from the last filter in the chain.

Methods of the FilterConfig Interface

The `FilterConfig` interface specifies the following methods, available for your optional use:

- `java.util.Enumeration getInitParameterNames()`

  You can set initialization parameters for a filter through `<init-param>` elements under the `<filter>` element in the `web.xml` file. (See "Configure the Filter".) Then, in your filter, you can use the `getInitParameterNames()` method of the `FilterConfig` object, which is passed in through the `init()` method, to retrieve an `Enumeration` object of Java strings containing the names of the initialization parameters. (The `Enumeration` object is empty if there are no initialization parameters for the filter.)

- `String getInitParameter(String paramname)`

  After retrieving initialization parameter names, use `getInitParameter()` to retrieve the value of a specified parameter.

- `ServletContext getServletContext()`

  You can use this method to retrieve the servlet context associated with the requested servlet (which the filter is filtering).

- `String getFilterName()`

  You can use this method to retrieve the name of the filter, according to the`<filter-name>` element in the `web.xml` file.

Implementing and Configuring Filters

This section shows the basic steps of implementing and configuring a filter. Steps such as these are included in a complete sample in "Simple Filter Example".

There is a subsection describing construction of the filter chain, based on your filter configuration order in `web.xml`.

Implement the Filter Code

This section lists steps in implementing code for a servlet filter.

1. Create a class that implements the `javax.servlet.Filter` interface. For example:

```
public class TimerFilter implements javax.servlet.Filter { }
```

2. For initialization of your filter, implement the `init()` method, specified in the `Filter` interface. First, create or retrieve a `javax.servlet.FilterConfig` object, which `init()` takes as input. For example:

```
private FilterConfig filterConfig;
...
public void init(final FilterConfig filterConfig)
{
    this.filterConfig = filterConfig;
}
```

   In case you want any special initialization processing, see "Methods of the FilterConfig Interface".

3. For your filter processing, implement the `doFilter()` method, specified in the `Filter` interface. This method takes a request object, a response object, and a `javax.servlet.FilterChain` object created by the servlet container. Implement whatever processing you want, and (typically) call the `doFilter()` method of the filter chain object to invoke the next entity in the chain. For example:

```
public void doFilter(ServletRequest request, ServletResponse response,
       FilterChain chain)
       throws java.io.IOException, javax.servlet.ServletException
{
    long start = System.currentTimeMillis();
    System.out.println("Milliseconds in: " + start);
    chain.doFilter(request, response);
    long end = System.currentTimeMillis();
    System.out.println("Milliseconds out: " + end);
}
```

   The first `println()` call is executed before the rest of the chain is invoked; the second `println()` call is executed afterward, when `chain.doFilter()` returns.

4. Implement the `destroy()` method, specified in the `Filter` interface, to clean up resources or do anything special before the filter is taken out of service. For example:

```
public void destroy()
{
    filterConfig = null;
}
```

> **Note:**
>
> There are additional considerations in implementing a filter to alter the HTTP request or response. See "Using a Filter to Wrap and Alter the Request or Response".

Configure the Filter

This section lists the steps in configuring a servlet filter. Do the following in `web.xml` for each filter:

1. Declare the filter through a `<filter>` element and its subelements, which maps the filter class (including package) to a filter name. For example:

```
<filter>
    <filter-name>timer</filter-name>
    <filter-class>filter.TimerFilter</filter-class>
</filter>
```

   You can optionally specify initialization parameters here, similarly to how you would for a servlet:

```
<filter>
    <filter-name>timer</filter-name>
    <filter-class>filter.TimerFilter</filter-class>
    <init-param>
        <param-name>name</param-name>
        <param-value>value</param-value>
    <init-param>
</filter>
```

2. Using a `<filter-mapping>` element and its subelements, map the filter name to a servlet name or URL pattern to associate the filter with the corresponding resource (such as a servlet or JSP page) or resources. For example, to have the filter invoked whenever the servlet of name `myservlet` is invoked:

   ```
   <filter-mapping>
       <filter-name>timer</filter-name>
       <servlet-name>myservlet</servlet-name>
   </filter-mapping>
   ```

   Or, to have the filter invoked whenever `sleepy.jsp` is requested, according to URL pattern:

   ```
   <filter-mapping>
       <filter-name>timer</filter-name>
       <url-pattern>/sleepy.jsp</url-pattern>
   </filter-mapping>
   ```

   Note that instead of specifying a particular resource in the `<url-pattern>`element, you can use wild card characters to match multiple resources, such as in the following example:

   ```
   <url-pattern>/mypath/*</url-pattern>
   ```

The filter name can be arbitrary, but preferably is meaningful. It is simply used as the linkage in mapping a filter class to a servlet name or URL pattern.

If you configure multiple filters that apply to a resource, they will be entered in the servlet chain according to their declaration order in `web.xml`, and they will be invoked in that order when the target servlet is requested. See the next section, "Construction of the Filter Chain".

Note:

There are additional steps to configure a filter for a forward or include target.
See "Filtering Forward or Include Targets".

Construction of the Filter Chain

When you declare and map filters in `web.xml`, the servlet container determines which filters apply to each servlet or other resource (such as a JSP page or static page) in the Web application. Then, for each servlet or resource, the servlet container builds a chain of applicable filters, according to your `web.xml` configuration order, as follows:

1. First, any filters that match a servlet or resource according to a `<url-pattern>`element are placed in the chain, in the order in which the filters are declared in`web.xml`.
2. Next, any filters that match a servlet or resource according to a `<servlet-name>` element are placed in the chain, with the first `<servlet-name>` match following the last `<url-pattern>` match.
3. Finally, the target servlet or other resource is placed at the end of the chain, following the last filter with a `<servlet-name>` match.

Simple Filter Example

This example shows a filter that is invoked when a JSP page is requested. The JSP page writes a line to the browser. The filter writes two lines to the OC4J console—one line before the JSP page runs, and one after.

Write the Simple Filter Code

Here is the code for the simple filter, `TimerFilter`. The `doFilter()` method writes two lines to the OC4J console, one before the target JSP page is executed and one after.

```
package filter;

import javax.servlet.*;

public class TimerFilter implements javax.servlet.Filter
{
    private FilterConfig filterConfig;

    public void doFilter(ServletRequest request, ServletResponse response,
           FilterChain chain)
           throws java.io.IOException, javax.servlet.ServletException
    {
        long start = System.currentTimeMillis();
        System.out.println("Milliseconds in: " + start);
        chain.doFilter(request, response);
        long end = System.currentTimeMillis();
        System.out.println("Milliseconds out: " + end);
    }

    public void init(final FilterConfig filterConfig)
    {
        this.filterConfig = filterConfig;
    }

    public void destroy()
    {
        filterConfig = null;
    }
}
```

Write the Target JSP Page

Here is the target JSP page, `sleepy.jsp`, which has a wait interval then outputs the wait time to the browser.

```
<%
  int sleeptime = 1000;
  Thread.sleep(sleeptime);
%>
<HTML>
<HEAD>
<TITLE>Sleepy JSP</TITLE>
</HEAD>
<BODY>
<HR>
<P><CENTER>Sleepy JSP slept for <%= sleeptime %> milliseconds!</CENTER></P>
<HR>
</BODY>
</HTML>
```

Configure the Simple Filter

Here is the configuration in `web.xml` that declares the simple filter and maps it to requests for `sleepy.jsp`:

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <filter>
```

```
      <filter-name>timer</filter-name>
      <filter-class>filter.TimerFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>timer</filter-name>
      <url-pattern>/sleepy.jsp</url-pattern>
   </filter-mapping>
</web-app>
```

Package the Simple Filter Example

The WAR file for this example, which we name `filter.war`, has the following contents and structure:

```
sleepy.jsp
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/filter/TimerFilter.class
WEB-INF/classes/filter/TimerFilter.java
```

And the EAR file is as follows:

```
filter.war
META-INF/Manifest.mf
META-INF/application.xml
```

(The `Manifest.mf` files are created automatically by the JAR utility.)

Invoke the Simple Filter Example

For this example, assume that `application.xml` maps the context path `/myfilter`to `filter.war`. In this case, after deployment, you invoke `sleepy.jsp` as follows, and the filter is executed as a result:

```
http://host:port/myfilter/sleepy.jsp
```

The following is written to the browser:

```
Sleepy JSP slept for 1000 milliseconds!
```

In a sample execution, the following is written to the OC4J console (where you started OC4J, for example, if you are running OC4J in a standalone environment):

```
04/04/30 13:01:19 Milliseconds in: 1083355279136
04/04/30 13:01:20 Milliseconds out: 1083355280152
```

The "Milliseconds in" line is written before the JSP page is invoked. The "Milliseconds out" line is written after the JSP page is done and execution returns to the filter. In this example, there is a difference of 1016 milliseconds, mostly due to the 1000-millisecond wait in the JSP page.

Filtering Forward or Include Targets

You can configure a filter to act on forward or include targets in addition to, or instead of, acting on direct request targets. This is described in the following subsections:

- The web.xml <dispatcher> Element
- Configuring Filters for Forward or Include Targets

Note:

See "Dispatching to Other Servlets Through Includes and Forwards"for general information about including and forwarding.

The web.xml <dispatcher> Element

Use the `<dispatcher>` subelement of `<filter-mapping>` in `web.xml` if you want to configure filters for forward or include targets. This element has four supported values:

- `INCLUDE`: Use this for the filter to be applied to any include targets matching a specified servlet name or with URLs matching a specified pattern.
- `FORWARD`: Use this for the filter to be applied to any forward targets matching a specified servlet name or with URLs matching a specified pattern.
- `REQUEST`: Use this in addition to an `INCLUDE` or `FORWARD` setting (one`<dispatcher>` element for each setting) for the filter to also be applied to direct request targets matching a specified servlet name or with URLs matching a specified pattern. (It is nonsensical to use the `REQUEST` value only. If you want the filter to apply only to direct requests, there is no need to use the`<dispatcher>` element.)
- `ERROR`: Use this for the filter to be applied under the error page mechanism.

See the following section, "Configuring Filters for Forward or Include Targets", for examples.

Configuring Filters for Forward or Include Targets

This section provides a few sample configurations to have a filter act on forward or include targets. We start with the filter declaration, followed by alternative filter mapping configurations:

```
<filter>
    <filter-name>myfilter</filter-name>
    <filter-class>mypackage.MyFilter</filter-class>
</filter>
```

To execute `MyFilter` to filter an include target named `includedservlet`:

```
<filter-mapping>
   <filter-name>myfilter</filter-name>
   <servlet-name>includedservlet</servlet-name>
   <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Note that the `include()` call can come from any servlet (or other resource) in the application. Also note that `MyFilter` would not execute for a direct request of`includedservlet`, unless you have another `<dispatcher>` element with the value`REQUEST`.

To execute `MyFilter` to filter any servlet directly requested through a URL pattern "`/mypath/`", or to execute it to filter any forward target that is invoked through a URL pattern starting with "`/mypath/`":

```
<filter-mapping>
   <filter-name>myfilter</filter-name>
   <url-pattern>/mypath/*</url-pattern>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Using a Filter to Wrap and Alter the Request or Response

Particularly useful functions for a filter are to manipulate a request, or manipulate the response to a request. To manipulate a request or response, you must create a wrapper. You can use the following general steps:

1. To manipulate requests, create a class that extends the

standard`javax.servlet.http.HttpServletRequestWrapper` class. This class will be your request wrapper, allowing you to modify a request as desired.

2. To manipulate responses, create a class that extends the standard`javax.servlet.http.HttpServletResponseWrapper` class. This class will be your response wrapper, allowing you to modify a response after the target servlet or other resource has delivered and possibly committed it.

3. Optionally create a class that extends the standard`javax.servlet.ServletOutputStream` class, if you want to add custom functionality to an output stream for the response.

4. Create a filter that uses instances of your custom classes to alter the request or response as desired.

The next section, "Response Filter Example", provides an example of a filter that alters the response.

Response Filter Example

This example employs an HTTP servlet response wrapper that uses a custom servlet output stream. This functionality allows the wrapper to manipulate the response data after the target HTML page is finished writing it out. Without using a wrapper, you cannot change the response data after the servlet output stream has been closed (essentially, after the servlet has committed the response). That is the reason for implementing a filter-specific extension to the `ServletOutputStream` class in this example.

This example uses the following custom classes:

- `GenericResponseWrapper`: Extends `HttpServletResponseWrapper` for custom functionality in manipulating an HTTP response.
- `FilterServletOutputStream`: Extends `ServletOutputStream` to provide custom functionality for use in the response wrapper.
- `MyGenericFilter`: This class is for a generic, empty ("pass-through") filter that is used as a base class.
- `PrePostFilter`: Extends `MyGenericFilter` and implements `doFilter()` code to alter the HTTP response, inserting a line before the HTML page output and a line after the HTML page output.

Write the Custom Output Stream Code

This class, `FilterServletOutputStream`, extends the standard`ServletOutputStream` class to implement special functionality that the response wrapper will use.

```
package mypkg;

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

public class FilterServletOutputStream extends ServletOutputStream {

  private DataOutputStream stream;

  public FilterServletOutputStream(OutputStream output) {
    stream = new DataOutputStream(output);
  }

  public void write(int b) throws IOException  {
    stream.write(b);
  }

  public void write(byte[] b) throws IOException  {
    stream.write(b);
  }

  public void write(byte[] b, int off, int len) throws IOException  {
```

```
    stream.write(b,off,len);
  }

}
```

Write the Response Wrapper Code

This class, `GenericResponseWrapper`, extends the standard`HttpServletResponseWrapper` class to implement custom functionality to manipulate an HTTP response.

```
package mypkg;

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

public class GenericResponseWrapper extends HttpServletResponseWrapper {
  private ByteArrayOutputStream output;
  private int contentLength;
  private String contentType;

  public GenericResponseWrapper(HttpServletResponse response) {
    super(response);
    output=new ByteArrayOutputStream();
  }

  public byte[] getData() {
    return output.toByteArray();
  }

  public ServletOutputStream getOutputStream() {
    return new FilterServletOutputStream(output);
  }

  public PrintWriter getWriter() {
    return new PrintWriter(getOutputStream(),true);
  }

  public void setContentLength(int length) {
    this.contentLength = length;
    super.setContentLength(length);
  }

  public int getContentLength() {
    return contentLength;
  }

  public void setContentType(String type) {
    this.contentType = type;
    super.setContentType(type);
  }

  public String getContentType() {
    return contentType;
  }
}
```

Write the Base Filter Code

This class, `MyGenericFilter`, is an empty filter, providing a template that is extended by the response filter of this example.

```
package mypkg;

import javax.servlet.*;
```

```
public class MyGenericFilter implements javax.servlet.Filter {
  public FilterConfig filterConfig;

  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
      throws java.io.IOException, javax.servlet.ServletException {
    chain.doFilter(request,response);
  }

  public void init(final FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
  }

  public void destroy() {
  }
}
```

Write the Response Filter Code

This class, `PrePostFilter`, which extends `MyGenericFilter`, is a filter that alters the response of the target HTML page, prepending a line of output before the HTML page output, and appending a line of output after the HTML page output.

```
package mypkg;

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

public class PrePostFilter extends MyGenericFilter {

  public void doFilter(final ServletRequest request,
                       final ServletResponse response,
                       FilterChain chain)
       throws IOException, ServletException {
  OutputStream out = response.getOutputStream();
  out.write(new String("<HR>PRE<HR>").getBytes());
  GenericResponseWrapper wrapper =
         new GenericResponseWrapper((HttpServletResponse) response);
  chain.doFilter(request,wrapper);
  out.write(wrapper.getData());
  out.write(new String("<HR>POST<HR>").getBytes());
  out.close();
  }
}
```

Write the Target HTML Page

This HTML page, `prepostfilter.html`, is requested by the user in this example. The filter inserts output before and after the output of this page.

```
<HTML>
<HEAD>
<TITLE>PrePostFilter Example</TITLE>
</HEAD>
<BODY>
This is a testpage. You should see<br>
this text when you invoke prepostfilter.html, <br>
as well as the additional material added<br>
by the PrePostFilter class.
<br>
</BODY>
</HTML>
```

Configure the Response Filter

Here is the configuration in `web.xml` that declares the response filter and maps it to requests for `prepostfilter.html`:

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (doctype...)>
<web-app>
   <filter>
      <filter-name>prepost</filter-name>
      <filter-class>mypkg.PrePostFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>prepost</filter-name>
      <url-pattern>/prepostfilter.html</url-pattern>
   </filter-mapping>
</web-app>
```

Package the Response Filter Example

The WAR file for this example, which we name `myresponsewrapper.war`, has the following contents and structure:

```
prepostfilter.html
META-INF/Manifest.mf
WEB-INF/web.xml
WEB-INF/classes/mypkg/FilterServletOutputStream.class
WEB-INF/classes/mypkg/FilterServletOutputStream.java
WEB-INF/classes/mypkg/GenericResponseWrapper.class
WEB-INF/classes/mypkg/GenericResponseWrapper.java
WEB-INF/classes/mypkg/MyGenericFilter.class
WEB-INF/classes/mypkg/MyGenericFilter.java
WEB-INF/classes/mypkg/PrePostFilter.class
WEB-INF/classes/mypkg/PrePostFilter.java
```

And the EAR file is as follows:

```
myresponsewrapper.war
META-INF/application.xml
META-INF/Manifest.mf
```

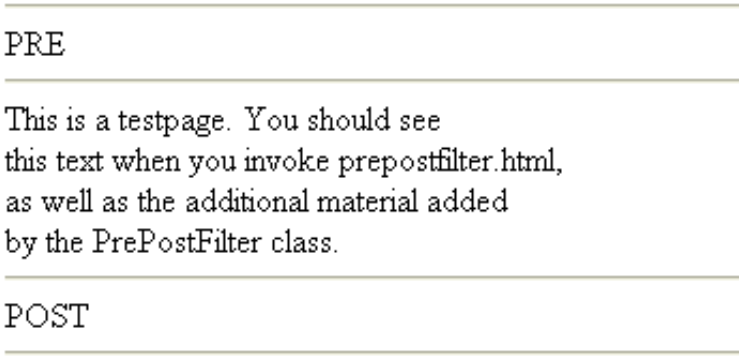
(The `Manifest.mf` files are created automatically by the JAR utility.)

Invoke the Response Filter Example

For this example, assume that `application.xml` maps the context path `/mywrapper`to `myresponsewrapper.war`. In this case, after deployment, you invoke`prepostfilter.html` as follows, and the filter is executed as a result:

```
http://host:port/mywrapper/prepostfilter.html
```

The following is output to the browser, with "`PRE`", "`POST`", and the horizontal rules coming from the filter:



Description of the illustration prepost.gif

Form Authentication Filter

An OC4J proprietary filter dispatcher was introduced in 10.1.3 that enables a filter to access the username/password credentials passed in to OC4J through Form Authentication. For example, you would do this if you want to perform further authentication on external resources.

To enable this feature, specify the `FORMAUTH` value in the `<dispatcher>` element for the filter in the `orion-web.xml` file.

The following examples show the code that declares the filter and the code that specifies the `FORMAUTH` feature:

> **Note:**
>
> The `<filter>` element can be declared in either the `orion-web.xml` file or the `web.xml` file.
>
> The `<filter-mapping>` element must be declared in the `orion-web.xml` file.

In `orion-web.xml` or `web.xml`:

```
<filter>
   <filter-name>MyFilter</filter-name>
   <filter-class>myFilterClass</filter-class>
</filter>
```

In orion-web.xml:

```
<orion-web-app>
...
   <web-app>
   ...
      <filter-mapping>
         <filter-name>MyFilter</filter-name>
         <dispatcher>FORMAUTH</dispatcher>
      </filter-mapping>
   </web-app>
</orion-web-app>
```

Any filter declared this way will be executed after the authentication form is submitted, but before authentication is performed in OC4J.

The filter can call `request.getParameters()` to get the `j_username` and`j_password` parameters from the request object.

Here is an example of the calling code:

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

public class MyFilter implements Filter {

    public MyFilter() {
        super();
    }

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain filterChain)
        throws IOException, ServletExxception {
        HttpServletRequest req = (HttpServletRequest) request;

String username = req.getParameter("j_username");
String password = req.getParameter("j_password");
// use these credentials to access a remote DB
....
filterChain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {
    }

    public void destroy() {
    }
}
```

- Die Struktur einer Web-Applikation kennen.

- Den Lebenszyklus von Servlets kennen und sie entwickeln und implementieren können.

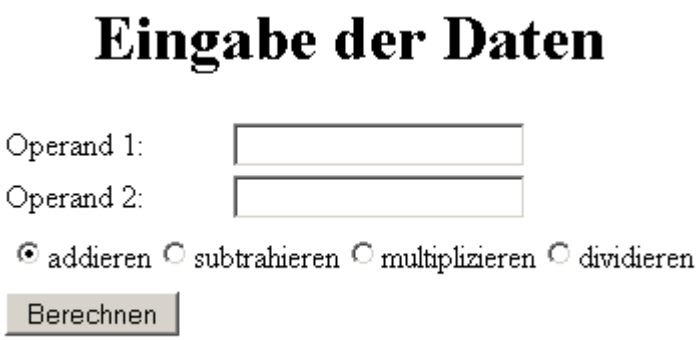
Inhaltsverzeichnis

1. Rechner

Mit der Servlet-Rechner Applikation sollen Sie Schritt für Schritt zeigen, wie die Servlet-Technologie eingesetzt wird:

- Wie wird ein Servlet implementiert
- Wie wird ein Servlet compiliert
- Wie werden die HTML-Form-Parameter vom Request extrahiert
- Wie wird die `web.xml` Datei konfiguriert
- Wie wird die `war`-Datei erzeugt und deployed

Die Applikation soll dem Benutzer ermöglichen, zwei Zahlen einzugeben und diese miteinander zu addieren, subtrahieren, multiplizieren oder zu dividieren:



Das Berechnungsresultat wird ihm auf einer neuen Seite angezeigt (66 / 8 = 8):

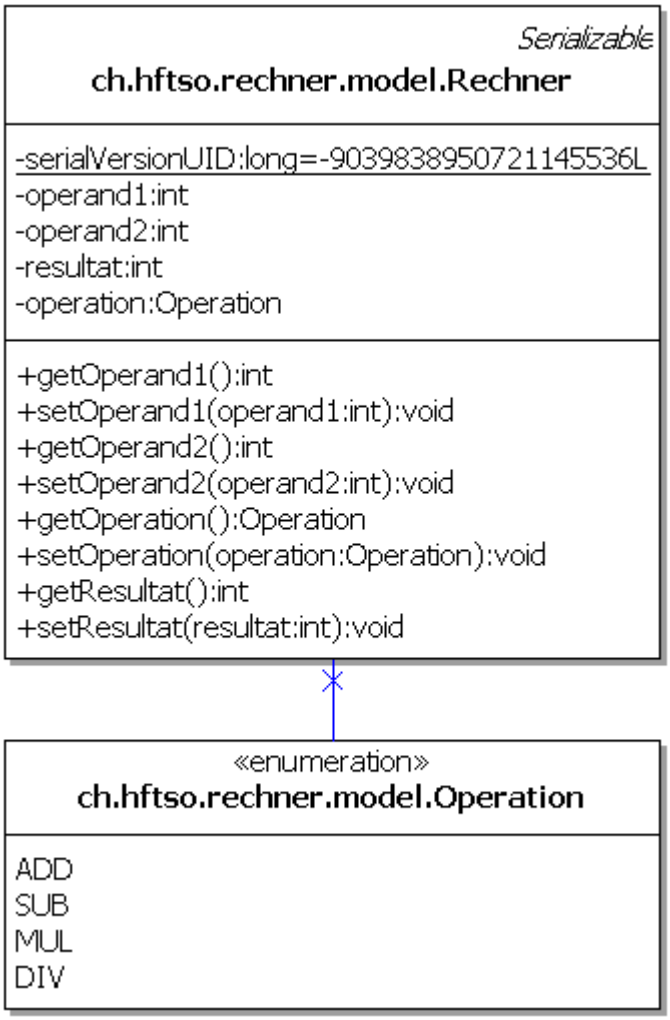
Berechnungsresultat

Operand 1	Operand 2	Operator	Resultat
66	8	DIV	8

Zurück

1.1 Modell-Klassen

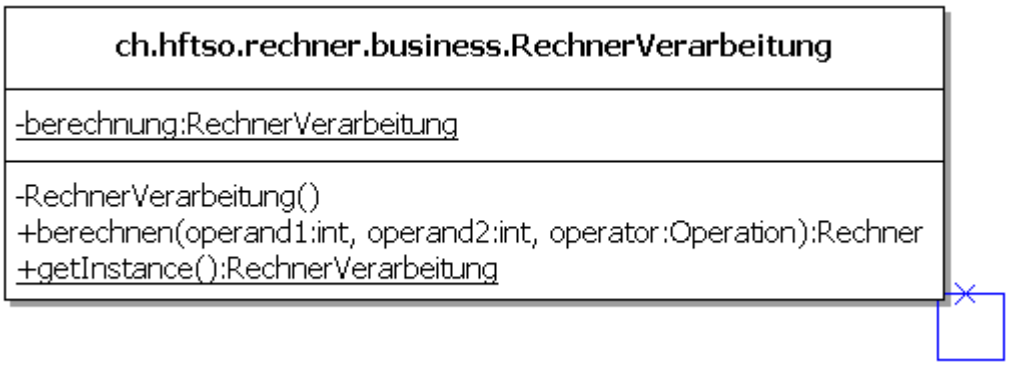
Das Klassendiagramm der Modell-Klassen sieht folgendermassen aus:



Für die Identifizierung des Operanden ist ein `enum` Typ `Operation` gefordert.

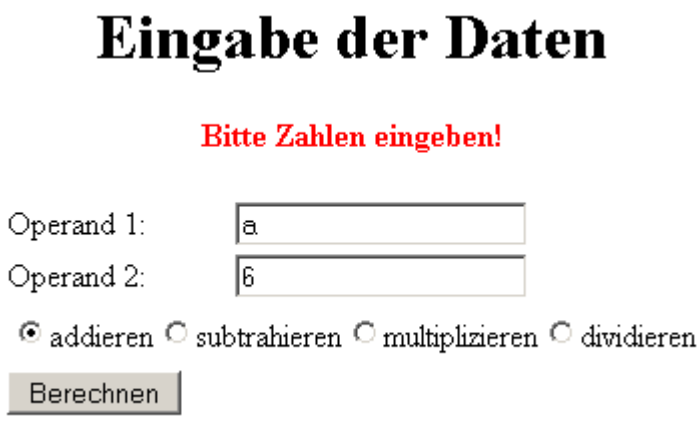
1.2 Business Klasse

Für die Berechnung der Resultate aus den Operanden und der dazu gehörenden Operation dient die Klasse `RechnerVerarbeitung`, deren Klassendiagramm folgendermassen aussieht:

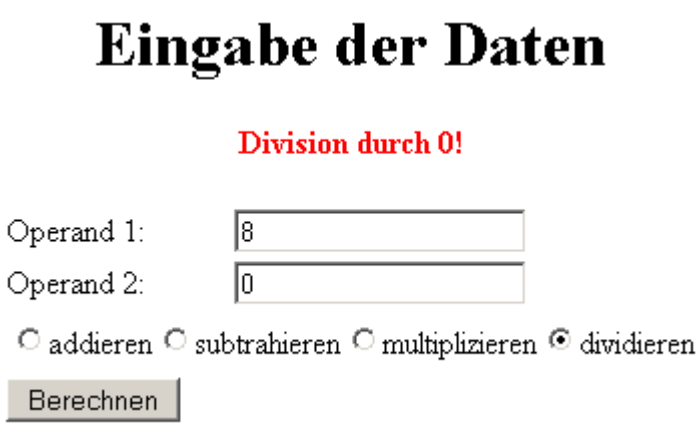


1.3 Servlets

Für die Eingabe der Daten ist das `EingabeServlet` verantwortlich, dessen Mapping `/eingabe` ist. Für die Anzeige des Berechnungsresultates ist das Servlet `AnzeigeServlet` zu implementieren, dessen Mapping `/anzeige` sein muss. Das `EingabeServlet` übergibt die vom Benutzer eingegebenen Daten an das `AnzeigeServlet`, das diese Daten prüft und an das Business-Objekt (Singleton) zur Berechnung weiter gibt. Die Weitergabe erfolgt nur bei korrekt eingegebenen Benutzerdaten, ansonsten wird dem Benutzer über das `EingabeServlet` die Fehleingabe angezeigt:



Falls das Business-Objekt eine Division durch 0 berechnet, informiert es das `AnzeigeServlet`, dieses informiert das `EingabeServlet` und dieses zeigt dem Benutzer die Meldung:

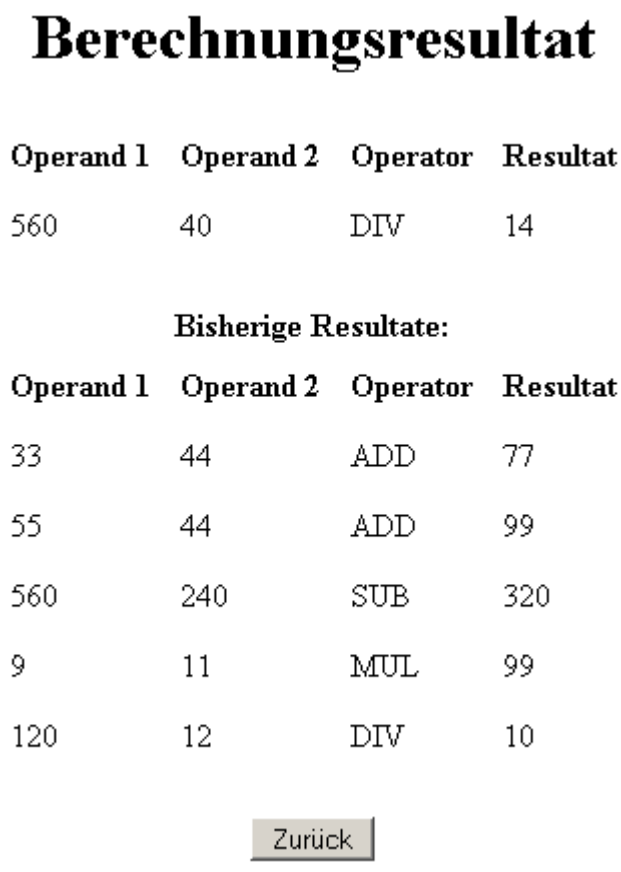


1.4 Aufgabe

Implementieren Sie aufgrund der dargestellten Fakten die Rechner-Applikation. Legen Sie Wert auf eine aufschlussreiche Dokumentation (JavaDoc).

2. Rechner mit JPA

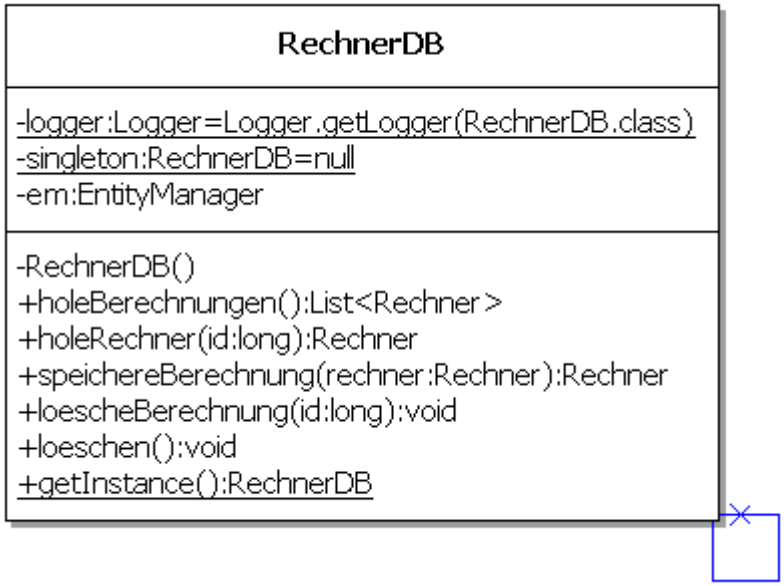
Die Rechner-Applikation aus Kapitel 1 muss die errechneten Resultate in eine Datenbank (HSQL) speichern und entsprechend dem Benutzer anzeigen:



Die aktuelle Berechnung darf in der Liste nicht erscheinen! Die Liste muss aufsteigend nach dem Operator sortiert angezeigt werden. Falls die Liste noch leer ist, darf sie nicht angezeigt werden.

2.1 Persistence Klasse

Für die Speicherung der Resultate ist die Klasse `RechnerDB` zu implementieren (mittels JPA und HSQLDB), das Klassendiagramm sieht folgendermassen aus:



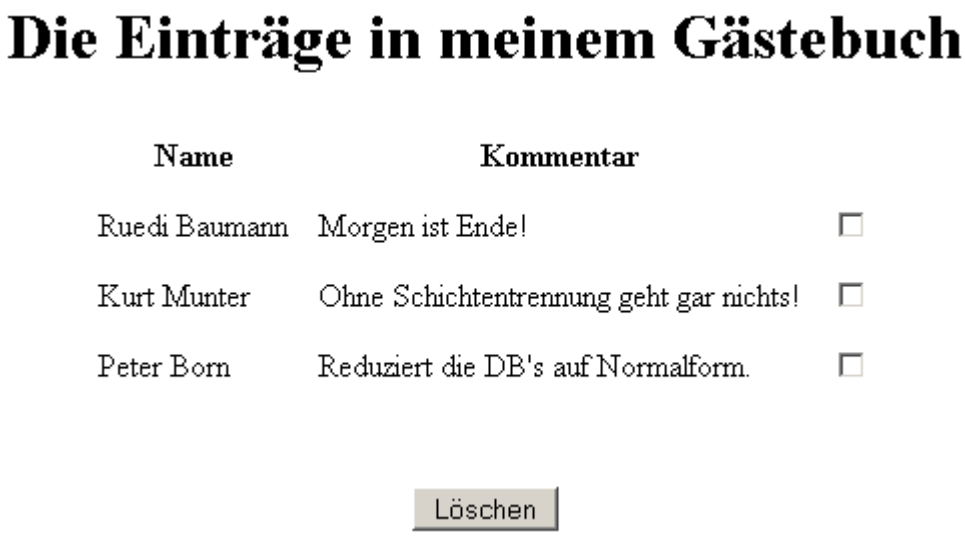
Die Signaturen der Methoden sind zu übernehmen und entsprechend zu implementieren. Die Businessklasse `Rechnerverarbeitung` aus Kapitel 1 muss für die Datenspeicherung angepasst werden.

2.2 Testklasse

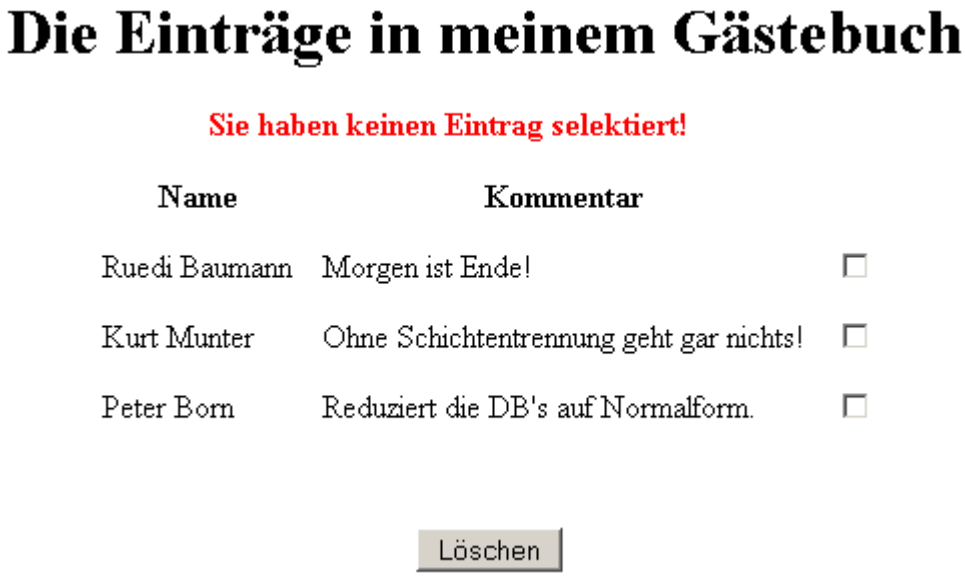
Die Testklasse ist mittels TestNG zu implementieren. Sie muss sämtliche Operationen der Businessklasse testen. Weiter muss sie nachweisen, dass alle Methoden der Persistenzklasse einwandfrei funktionieren. Die Methode `löschen` der Persistenzklasse löscht alle Berechnungen in der Datenbank!

3. Gästebuch

Die aus dem Unterricht bekannte Gästebuch-Applikation ist mit der Funktionalität »Löschen« zu erweitern, d.h. der Benutzer muss die einzelnen Einträge löschen können. Die Benutzeroberfläche dazu sollte folgendes aussehen haben (der Banner und die Menüführung ist weggelassen):



Selektiert der Benutzer eine oder mehrere Checkboxen und klickt auf den Button »Löschen«, werden die markierten Einträge gelöscht. Vergisst der Benutzer eine Checkbox zu selektieren und klickt auf den Button »Löschen«, wird ihm das gemeldet:



Falls alle Einträge gelöscht worden sind, darf die Tabelle nicht mehr angezeigt werden, sondern es muss dem Benutzer gemeldet werden: