Arbeitsblatt 3 - Servlet Komponenten

Ziele

Sie kennen die verschiedenen Basis-Komponenten für Entwicklung von Java Webapplikation mit Java EE und können diese auch korrekt einsetzen.

Theorie

Servlet-Container

Servlets sind zunächst einmal nichts anderes als ganz normale Java-Klassen, die ein bestimmtes Interface `javax.servlet.Servlet` implementieren. Damit diese Klassen wirklich auf HTTP-Anfragen eines Browsers reagieren und die gewünschte Antwort liefern können, müssen die Servlets in einer bestimmten Umgebung laufen. Diese Umgebung wird vom sogenannten **Servlet-Container** bereitgestellt.

Der Container sorgt zunächst einmal für den korrekten Lebenszyklus der Servlets (siehe später). Zumeist arbeitet der Container im Zusammenspiel mit einem Webserver. Der Webserver bedient z.B. die Anfragen nach statischem HTML-Content, nach Bildern, multimedialen Inhalten oder Download-Angeboten. Kommt hingegen eine Anfrage nach einem Servlet oder einer JSP herein, so leitet der Webserver diese an den Servlet-Container weiter. Dieser ermittelt das zugehörige Servlet und ruft dieses mit den Umgebungsinformationen auf. Ist das Servlet mit seiner Abarbeitung des Requests fertig, wird das Ergebnis zurück an den Webserver geliefert, der dieses dem Browser wie gewöhnlich serviert.

Servlet (HttpServlet)

`javax.servlet.Servlet` ist das Interface, das letztendlich entscheidet, ob eine Java-Klasse überhaupt ein Servlet ist. Jedes Servlet muss dieses Interface direkt oder indirekt implementieren. De facto wird man aber wohl in den meisten Fällen nicht das Interface selber implementieren, sondern auf `javax.servlet.GenericServlet` (falls man kein Servlet für HTTP-Anfragen schreibt), bzw. `javax.servlet.http.HttpServlet` für http-Anfragen zurückgreifen. `HttpServlet` implementiert alle wesentlichen Methoden bis auf die http-doXXX-Methoden wie `doGet` oder `doPost`. Man muss nur noch die Methoden `doGet`, `doPost` oder `doXYZ` überschreiben, je nachdem, welche HTTP-Methoden man unterstützen will.

ServletRequest und ServletResponse

In den Verarbeitungsmethoden von Servlets (`service(), doGet(), doPost()`...) hat man stets Zugriff auf ein Objekt vom Typ `ServletRequest`. Arbeitet man mit `HttpServlet` handelt es sich um das Interface `javax.servlet.http.HttpServletRequest`, ansonsten um das Interface `javax.servlet.ServletRequest`. Diese Interfaces stellen wesentliche Informationen über den Client-Request zur Verfügung.

Man implementiert selber nie diese Interfaces, vielmehr stellt der jeweilig benutzte Container konkrete Implementierungen dieser Interfaces für die Entwickler transparent zur Verfügung. Grundlage des eigenen Codes sollten aber immer die Interfaces selbst sein, niemals die konkrete Implementierung eines Container-Herstellers, da sonst die Portabilität verloren geht.

Letztlich geht es bei Servlets immer darum, auf einen Client-Request mit einer Antwort zu reagieren. Um diese Antwort generieren zu können, benötigt man ein Objekt des Typen `ServletResponse`. Während man aber auf das `ServletRequest`-Objekt sehr häufig zurückgreift, benötigt man das `ServletResponse`-Objekt vergleichsweise selten. Es dient bspw. im Falle eines Fehlers dazu,

einen anderen HTTP-Statuscode als "200" zu setzen. Auch kann man HTTP-Header mit dem Response-Objekt setzen. Ein paar Methoden sind aber doch wichtig:

- `sendRedirect(String)` zur Auslösung eines Redirects,
- `setContentType(String)` zur Setzung des Mime-Types der Antwort
- `setCharacterEncoding(String)` zur Setzung des richtigen Zeichensatz-Encodings (bspw. UTF-8 oder ISO-8859-1).

Schlussendlich muss das Servlet seine Ausgabe auch irgendwohin schreiben. Dazu holt man sich aus dem `Response`-Objekt mit der Methode `getOutputStream()` ein Objekt vom Typ `ServletOutputStream` (für binäre Inhalte wie bspw. generierte PDF-Dateien oder Bilder) oder mit der Methode `getWriter()` ein Objekt vom Typ `PrintWriter`, wenn man als Antwort textuelle Daten generiert (bspw. HTML-Code oder XML-Daten).

ServletContext
Jedes Servlet wird im Kontext der Webapplikation ausgeführt. Dieser Kontext gilt für alle Servlets der entsprechenden Webapplikation. Deshalb werden Informationen, die im Scope Webapplikation gelten hier abgelegt, so dass jedes Servlet auf diese zugreifen kann.

Kontext-Parameter werden im Deployment-Deskriptor konfiguriert. Zum Beispiel:

```
<webapp ...>
   <context-param>
      <param-name>DBUSER</param-name>
      <param-value>webfr</param-value>
   </context-param>
```

In einem Servlet kann man über das `ServletRequest`-Objekt auf Werte zugreifen

```
protected void doGet(HttpServletRequest request, HttpServletRespones response)... {
   String dbuser = (String) request.getServletContext().getInitParameter("DBUSER");
   ...
```

Ebenfalls kann man zur Laufzeit den ServletContext mit eigenen Attributen füllen und diese wieder auslesen. Hier ein einfaches Beispiel:

```
servletContext.setAttribute("qRepo", QuestionnaireRepository.getInstance());
...
QuestionnaireRepository qr = (QuestionnaireRepository) servletContext.getAttribute("qRepo");
```

Servlet-Lebenszyklus
Ein Servlet wird von einem Servlet-Container verwaltet. Ein Servlet wird über den sogenannten Deployment-Deskriptor (DD) dem Container bekannt gemacht. Der Deployment-Deskriptor ist eine XML-Datei, die stets unter dem in der Servlet-Spezifikation festgelegten Namen "web.xml" im ebenfalls dort festgelegten Verzeichnis "WEB-INF" abgelegt werden muss. Nehmen wir an, wir hätten in einer Webapplikation „basic“ ein Servlet "BasicServlet" im Package "ch.fhnw.edu.basic". Dieses Servlet wollen wir unter der URL "first" aufrufen. Dann würden wir dazu folgenden Deployment-Deskriptor schreiben:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="3.0"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
```

```
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
   <servlet>                                                                    #1
      <servlet-name>BasicServlet</servlet-name>
      <servlet-class>ch.fhnw.webfr.flashcard.BasicServlet</servlet-class>
   </servlet>

   <servlet-mapping>                                                            #2
      <servlet-name>BasicServlet</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

</web-app>
```

#1 Ein Servlet MUSS immer mit einem Namen und einer Klasse konfiguriert werden.

#2 Das URL-Mapping auf ein Servlet. Erst mit diesem Mapping kann das Servlet vom Container angesprochen werden. Mit diesem Mapping werden alle Request an das BasicServlet weitergeleitet.

Seit der Servlet Spec 3.0 kann man für die Konfiguration aller Basis-Komponenten (Servlet, Filter, Listener) auch Annotationen einsetzen. Damit kann man die Grösse des Deployment Descriptor "web.xml" stark verringern. Der obige Eintrag wird dann im `BasicServlet` selber zu:

```
@WebServlet(urlPatterns={"/*"})                                                 #1
public class BasicServlet extends HttpServlet {
...
```

#1 Diese Annotation entspricht der XML-Element <servlet> und <serlvet-mapping> im vorergehenden File web.xml. Falls der Name nicht explizit gesetzt ist, wird als Name der fully-qualified-Classname verwendet.

Der Container reicht aber nicht nur einfach Aufrufe weiter. Er instanziiert zudem das Servlet, ruft die Lebenszyklus-Methoden auf und sorgt für die korrekte Umgebung. Jedes Servlet durchläuft dabei in der Spezifikation genau definierte Phasen:

Laden der Servlet-Klasse Zunächst muss der Classloader des Containers die Servlet-Klasse laden. Wann der Container dies macht, bleibt ihm überlassen, es sei denn, man definiert den entsprechenden Servlet-Eintrag im Deployment-Deskriptor. Mit dem optionalen Eintrag "load-on-startup" wird definiert, dass der Servlet-Container die Klasse bereits beim Start des Containers lädt. Die Zahlen geben dabei die Reihenfolge vor (Servlets mit niedrigeren "load-on-startup"-Werten, werden früher geladen).

Instanziieren Unmittelbar nach dem Laden wird das Servlet instanziiert, d.h. der leere Konstruktor wird aufgerufen.

init(ServletConfig) Bevor der erste Request an das Servlet weiter gereicht wird, wird dieses initialisiert. Dazu wird die Methode `init(ServletConfig)` des Servlets aufgerufen. Diese Methode wird genau einmal im Lebenszyklus eines Servlets und nicht etwa vor jedem Request aufgerufen und dient dazu, grundlegende Konfigurationen vorzunehmen. Dem Servlet wird dabei ein Objekt vom Typ `javax.servlet.ServletConfig` mitgegeben. In diesem `ServletConfig`-Objekt sind die `Init`-Parameter abgelegt. Diese Parameter können im web.xml oder über Annotationen gesetzt werden. Beispiel mit Parameter "email" im web.xml:

```
<servlet>
   <servlet-name>ch.fhnw.edu.basic.BasicServlet</servlet-name>
   <servlet-class>BasicServlet</servlet-class>
   <init-param>
      <param-name>email</param-name>
      <param-value>hugo.tester@fhnw.ch</param-value>
   </init-param>
</servlet>
```

Nach Abarbeiten der `init()`-Methode ist das Servlet bereit, Anfragen entgegen zu nehmen und zu beantworten.

service(ServletRequest, ServletResponse) Für jede Anfrage eines Clients an ein entsprechend definiertes Servlet, wird die Methode `service(ServletRequest, ServletResponse)` aufgerufen. Servlet-Container halten zumeist nur genau eine Instanz pro Servlet. Somit wir jeder Request in einem eigenen Thread aufgerufen. Die daraus entstehenden Threading-Issues müssen unbedingt beachtet werden!

Alle wesentlichen Informationen, die die Client-Anfrage betreffen, befinden sich im `ServletRequest`-Objekt. Und zur Erzeugung der Antwort nutzt man Methoden des `ServletResponse`-Objekts. In eigenen Servlets für HTTP-Anfragen (und das ist sicherlich der Regelfall für selbstgeschriebene Servlets) sollte man allerdings nie die `service()`-Methode selber überschreiben. Vielmehr bietet es sich an, die eigene Klasse von `HttpServlet` abzuleiten und dessen Methoden `doGet(), doPost()` etc. zu überschreiben. Dadurch wird das korrekte Handling der diversen HTTP-Methoden automatisch vom Container übernommen.

destroy() Der Container kann jederzeit eine Servlet-Instanz als überfällig ansehen und diese aus dem Request/Response-Zyklus entfernen. Dazu ruft er am Ende des Lebenszyklus der Servlet-Instanz deren `destroy()`-Methode auf und gibt damit dem Servlet die Möglichkeit z.B: Ressourcen wie Datenbank-Connections freizugeben. Der Container muss zuvor dem Servlet allerdings noch die Gelegenheit geben, seine in der Bearbeitung befindlichen Requests abzuarbeiten, bzw. zumindest einen definierten Timeout abwarten, bevor er die Abarbeitung unterbricht.
Ist einmal die Methode `destroy()` aufgerufen, steht diese Servlet-Instanz nicht mehr für weitere Anfragen zur Verfügung.

Filter
Servlet-Filter bieten eine Möglichkeit auf Request und Response zwischen Client und Servlet zuzugreifen. Dabei können mehrere Filter eine Filterkette bilden. Dabei wird mittels Mapping-Regeln bestimmt, welche Filter für welche Requests wann zuständig sind.

Es gibt zahlreiche Möglichkeiten, bei denen der Einsatz eines Filters sinnvoll sein kann. Der einfachste Anwendungsfall ist das Tracing, um zu sehen welche Ressource angesprochen wurde und wie lange die Bereitstellung der Ressource gedauert hat. Weitere typische Anwendungsfälle umfassen die Security bspw. eine Entschlüsselung des Requests und die Verschlüsselung der Response.

Filter sind im DD deklariert (oder über die Annotation @WebFilter). Sie werden vom Container verwaltet und müssen das Interface `javax.servlet.Filter` implementieren. Es gibt drei Methoden, die den Lifecycle bestimmen:

- `init(FilterConfig)` - wird gerufen, nachdem der Container die Instanz der Filterklasse erzeugt hat

- `doFilter(ServletRequest, ServletResponse, FilterChain)` - wird bei der Abarbeitung der FilterChain gerufen
- `destroy()` - die Filterinstanz wird gleich beseitigt, bitte aufräumen

Ein oder mehrere deklarierte Filter bilden eine Filterkette. Die typische Implementierung von `doFilter()` macht dieses Konzept deutlich:

```
doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
  // Code der etwas vor Aufruf der Kette macht
  chain.doFilter(request, response); // weitere Abarbeitung im Filterstack
  // Code der etwas nach Aufruf der Kette macht
}
```

Damit der Container eine Filterinstanz erzeugen kann, muss jeder Filter einen parameterlosen Konstruktor besitzen.

Im DD werden Filter deklariert und mittels Mapping-Regeln per URL-Pattern oder Servlet-Name bei der Abarbeitung eines Requests berücksichtigt:

```
// Filter deklarieren
<filter>
   <filter-name>My cool filter</filter-name>
   <filter-class>foo.bar.MyFilter</filter-class>
   <init-param> // kann im FilterConfig abgegriffen werden
      <param-name>loglevel</param-name>
      <param-value>10</param-value>
   </init-param>
</filter>

// Filter auf eine URL mappen
<filter-mapping>
   <filter-name>My cool filter</filter-name>
   <url-pattern>*.do</url-pattern>
</filter-mapping>
```

oder das Mapping über <servlet-name>

```
// Filter auf ein Servlet mappen
<filter-mapping>
   <filter-name>My cool filter</filter-name>
   <servlet-name>SomeServlet</servlet-name>
</filter-mapping>
```

Die Reihenfolge mehrerer Filter im Filterstack bestimmt der Container nach den folgenden Regeln:

1. Zunächst werden alle passenden `url-pattern` gesucht und in der Reihenfolge ihres Erscheinens im DD auf den Filterstack gelegt.
2. Nun wird das gleiche mit allen passenden `servlet-name` Filtern gemacht.

Filter werden ähnlich wie Servlets einmal pro Webapplikation instanziiert. Sie sind Webapplikation-Singletons.

Listener
Im Gegensatz zum Filter, der sich in den Request/Response-Pfad zwischen Client und Servlet einbindet, reagiert ein Listener auf verschiedene Ereignisse des Servlet-Containers selber.

Listener sind Klassen, die das Servlet Listener Interface implementieren und die im Deployment Deskriptor (DD) (oder über die Annotation @WebListener) dem Container bekannt gemacht werden. In einer Webanwendung können problemlos mehrere Listener vorhanden sein. Sie werden bei Lifecycle-Ereignissen der Webanwendung vom Container aufgerufen. Listener werden pro Anwendung einmal in der Reihenfolge ihres Erscheinens im DD instanziiert, sie sind Singletons. Eine Listener-Klasse muss unbedingt den Standard-Konstruktor besitzen, damit der Container eine Instanz erzeugen kann.

```
<listener>
  <listener-class>a.b.MyRequestListener</listener-class>
</listener>
```

Die Singletoneigenschaft der Listener-Instanzen erzwingt, dass Zustände der entsprechenden Scopes nicht in den Listener-Klassen gehalten werden dürfen. Die Listener Implementierungen müssen selbst dafür sorgen und es gilt ähnlich wie für Filter: nichttriviale Listener werden synchronisierten Code enthalten und die Synchronisation erfolgt am entsprechenden Scope. Ein typisches Codefragment für das Setzen eines Zustands im Scope der Session eines Nutzers sieht dann so aus:

```
..
HttpSession session = event.getSession();
synchronized (session) {
  session.setAttribute("SOME_STATE", new Integer(1));
}
..
```

Für alle möglichen Ereignisse, welche in einer Webapplikation passieren können, gibt es insgesamt acht Listener-Interfaces. Sie bieten die Möglichkeit auf entsprechende Ereignisse zu reagieren. Folgende Listener Interfaces sind spezifiziert:

`ServletContextListener`: Wird eine Webapplikation deployed, undeployed oder neu gestartet, wird deren `ServletContextListener` benachrichtigt. Typischerweise gibt es in einer Webapplikation eine Klasse, die diesen Listener implementiert. Zum Beispiel hat die Klasse die Aufgabe die Webapplikation zu initialisieren, wie Datenbank-Connections herzustellen, Logger zu konfigurieren etc. Geht dabei irgendwas schief und die Webapplikation ist nicht lebensfähig, kann man eine `RuntimeException` werfen. Tomcat beendet die Webapplikation und ruft `contextDestroyed()` des Listeners auf.

`ServletContextAttributeListener`: Diese Listener werden benachrichtigt, wenn ein Attribut im ServletContext gesetzt, ersetzt oder entfernt wird.

`HttpSessionListener`: Diese Listener werden benachrichtigt, wenn eine HttpSession erzeugt oder zerstört wurde. Damit kann man z.B. die Anzahl der aktiven Sessions (also User der Webapplikation) zählen, um z.B. mehr Ressourcen bereitstellen.

`HttpSessionAttributeListener`: Diese Listener werden benachrichtigt, wenn irgendein Attribut einer Session gesetzt, ersetzt oder entfernt wird.

`HttpSessionBindingListener`: Objekte einer Klasse, welche diesen Listener implementiert, werden vom Container benachrichtigt, wenn sie als Attribut in einer HttpSession gespeichert werden bzw. wenn sie aus der Session entfernt werden.

`HttpSessionActivationListener`: In verteilten WebApps darf es nur genau ein Session-Objekt pro Session-ID geben, egal über wieviel JVMs die WebApp verteilt ist. Durch Load-Balancing des Containers kann es passieren, dass jeder Request bei einer anderen Instanz des gleichen Servlets ankommt. Also muss die Session für diesen Request von einer JVM zu der anderen umziehen.

Der `HttpSessionActivationListener` wird wiederum von den Attributen der Session implementiert, sodass sie benachrichtigt werden bevor und nachdem eine Session-Migration stattfindet. Die Attribute können dann dafür sorgen, dass sie den Trip überleben.

`ServletRequestListener`: Dieser Listener wird benachrichtigt, wenn ein Request die Webapplikation erreicht. Damit kann man z.B. Requests loggen.

`ServletRequestAttributeListener`: Dieser Listener wird benachrichtigt, wenn ein Attribut eines Requests gesetzt, ersetzt oder entfernt wird.

Aufgaben

Aufgabe 1: Servlet-Filter
Implementieren sie den Filter „BasicFilter“, der bei jedem HTTP-Request die entsprechende URL in das Log bzw. auf die Console schreibt, zum Beispiel:

```
Before request [uri=/flashcard-basic/]
Before request [uri=/flashcard-basic/questionnaires]
```

Hinweis:
- Nutzen sie den log4j-Logger für die Ausgabe. Die entsprechende Bibliothek ist bereits im CLASSPATH vorhanden (siehe File "build.gradle").
- Hier ein Beispiel für ein entsprechendes Konfigurationsfile "log4j.properties":

```
log4j.rootLogger=INFO, stdout
log4j.logger.ch.fhnw.webfr.flashcard=debug
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
```

Aufgabe 2: Servlet-Listener und Servlet-Kontext
Momentan gilt folgende Ausgangslage: Das `QuestionnaireRepository` wird in `BasicServlet.init()` initialisiert und dann als Singleton weiterverwendet. Diese Initialisierung ist konzeptuell keine Aufgabe des Servlets. Sie gehört zum Aufbau der Infrastruktur der Webapplikation. Deshalb soll neu beim Hochfahren der Webapplikation auch das Repository erzeugt werden. Das Repository wird anschliessend über den Servlet-Context den Servlets zur Verfügung gestellt.

Aktuell werden ein paar Demo-Fragebögen bei der Initialisierung des Repository erzeugt und im Repository abgelegt. Diese Demo-Fragebögen sollen neu aber nur im Test Modus generiert werden. Dieser Modus soll über den Context-Parameter "mode" gesteuert werden.

Führen sie deshalb im File "web.xml" den Context-Parameter "mode" folgendermassen ein:

```
  <context-param>
    <param-name>mode</param-name>
    <param-value>test</param-value>
  </context-param>
```

Überlegen sie sich zuerst wie der zu entwickelnde Listener "BasicListener" diesen Context-Parameter auslesen kann, um dann das `QuestionnaireRepository` zu initialisieren und zu befüllen - oder eben nicht.

Nun müssen sie einen Weg finden, wie das Servlet auf die im Listener erzeugte Instanz des `QuestionnaireRepository` zugreifen kann. Im Servlet selber soll das `QuestionnaireRepository` zum Beispiel folgendermassen genutzt werden können:

```
Questionnaire questionnaire = questionnaireRepository.findById(id);
```

Implementieren sie den Listener "BasicListener" und passen sie das Servlet "BasicServlet" entsprechend an. Nutzen sie dabei den "ServletContext" und die Attribute so wie sie in Abschnitt ServlerContext erwähnt sind.