Web Frameworks

Roland Hediger

5. November 2013

Inhaltsverzeichnis

1 Das Servlet

Listing 1.1: SServlet Example

```
1  WebServlet(urlPatterns={"/first/*"}) #1
   public class BasicServlet extends HttpServlet { #2
   private List<Questionnaire> questionnaires;
   private Logger logger = Logger.getLogger(this.getClass());
   protected void doGet(HttpServletRequest request,
6  HttpServletResponse response) throws ServletException, IOException {
   response.setContentType("text/html;charset=ISO-8859-1");
   #3
   #4
   // get parameters from the request
11 String answerRequest = request.getParameter("answer");
   #5
   String questionnaireRequest = request.getParameter("questionnaire");
   if ((questionnaireRequest != null) && (answerRequest == null)) {
   ...
16 } else {
   handleIndexRequest(request, response);
   }
   }
   private void handleIndexRequest(HttpServletRequest request,
21 HttpServletResponse response) throws IOException {
   // create html response
   PrintWriter writer = response.getWriter();
   #6
   writer.append("<html><head><title>Example</title></head><body>");
26 writer.append("<h3>Frageb gen</h3>");
   for (Questionnaire questionnaire : questionnaires) {
   String url = "?questionnaire=" + questionnaire.getSubject();
   writer.append("<a href=" + response.encodeURL(url) + ">"
   + questionnaire.getTitle() + "</a><br/>");
31 }
   writer.append("</body></html>");
   }
   ...
   @Override
36 public void init(ServletConfig config) throws ServletException {
   #7
   super.init(config);
   questionnaires = QuestionnaireInitializer.createQuestionnaires();
   #8
41 }
   }
```

1.1 Servlet Allgemein

Passt mi Build Path in Java Eclipse zusammen.

Function von Web XML Vor 3.0 notwendig nicht mehr weil Servlets jetzt Annotationsgesteurt sind. Container verwaltet die Application und muss notwendigen Informationen über die Komponenten davon wissen. Man kann auf web.xml verzichten falls man jede xml Element in eine entsprechende Annotation umwandelt. **Eine mischung ist empfehlenswert.**

URL Pattrns in web.xml Servlet Mappings stellen die verschiedene Pfade dar worauf der Servlet wartet dar. **Welcome file ist eine liste von gemappte URL Patterns für den Index Fall**

Servlet Context ist alles unter app verzeichnis im webapps. Konkret: webapps/basic.

Zeilen Erklärung

1. Annotation
2. Http Ausprägung von Servlet - kann auch andere Protokolle.
3. Override von entsprechende methode doGet gibt HTTPResponse aus.

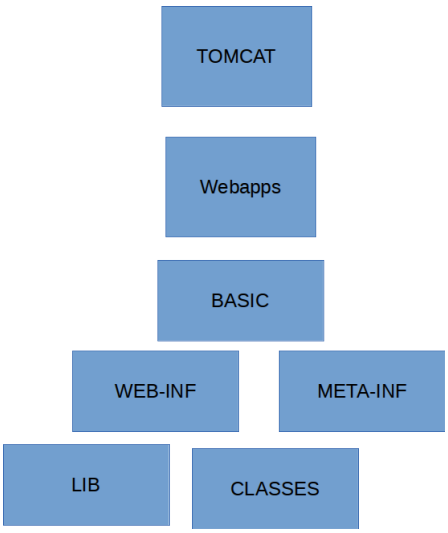

Abbildung 1.1: figure

4. Erste Zugriff auf Response muss sagen was für ein Typ der Content ist.
5. Hohle get Parameter answer=bla, optional.
6. Behandelt Index Request, Writer hohlen, html schreiben. Buffer/ZeichenOrientierte Antwort.
7. LifeCycle Start, einmal aufgerufen.
8. Custom Model Klasse.

1.2 Servlet Komponenten

1.2.1 Servlet Definitionen

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: 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 (HttpServletRequest und HttpServletResponse) 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 vom Typ 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 "200ßu setzen. Des Weiteren 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>

protected void doGet(HttpServletRequest request, HttpServlet Respones response)... {
String dbuser = (String) request.getServletContext().getAttribute("DBUSER");
...
```

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.xmlïm ebenfalls dort festgelegten Verzeichnis "WEB-INFäbgelegt werden muss. Nehmen wir an, wir hätten in einer Webapplikation „basic“ ein Servlet "BasicServletïm Package ch.fhnw.edu.basic". Dieses Servlet wollen wir unter der URL firstäufrufen. 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.edu.basic.BasicServlet</servlet-class>
</servlet>
<servlet-mapping>
#2
<servlet-name>BasicServlet</servlet-name>
<url-pattern>/first/*</url-pattern>
</servlet-mapping>
</web-app>

 //Seit 3.0
 @WebServlet(urlPatterns={"/first/*"})
#1
public class BasicServlet extends HttpServlet {
...
```

- Name & Klasse konfiguration (muss) @Annotation nimmt als Default FQCN [1]
- Mapping von URL auf Servlet. Annotation @WebServlet.

[1] Fully Qualified Class Name

1.3 Servlet Lebenszyklus

Laden der Servlet-Klasse unächst 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-startupWerten, werden früher geladen).

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

init(ServletConfig) evor 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 ëmailïm 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.

1.4 Filter

Servlet-Filter bieten eine Möglichkeit auf Request und Response zwischen Client und Servlet zuzugreifen (Abbildung 2). 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 Logging, 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. 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 Mappingregeln per URL-Pattern oder Servlet-Name bei der Abarbeitung eines Requests berücksichtigt:

Listing 1.2: Filter Beispiel

```
    // Filter deklarieren
   <filter>
 3 <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>
 8 </init-param>
   </filter>
   // Filter auf eine URL mappen
   <filter-mapping>
   <filter-name>My cool filter</filter-name>
13 <url-pattern>*.do</url-pattern>
   </filter-mapping>
   // Filter auf ein Servlet mappen
   <filter-mapping>
   <filter-name>My cool filter</filter-name>
18 <servlet-name>SomeServlet</servlet-name>
   </filter-mapping>
```

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

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

Filter werden, ähnlich wie Servlets, einmal pro Webapplikation instanziiert, sie sind Webapplikation- Singletons. Die Spezifikation sagt zwar, dass per JVM eine Instanz erzeugt wird, aber das wird nicht stimmen. Dann könnte man ja Informationen zwischen verschiedenen Webanwendungen, die in der gleichen JVM laufen, austauschen. Das klingt interessant, brächte aber erhebliche konzeptionelle Probleme mit sich, von denen in der Spezifikation keine Rede ist. Filter müssen für den nebenläufigen Einsatz designed werden, mit anderen Worten, nichttriviale Filter enthalten synchronisieren Code.

1.5 Listener

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:

1.5.1 Listener Interfaces

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: n 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 selben 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 Attribute eines Requests gesetzt, ersetzt oder entfernt wird.

2 MVC Design Pattern

Model-View-Controller ist ein Architekturmuster zur Strukturierung der Software in die drei Einheiten Datenmodell (engl. model), Präsentation (engl. view) und Programmsteuerung (engl. controller) (siehe Abbildung 1). Ziel des Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht

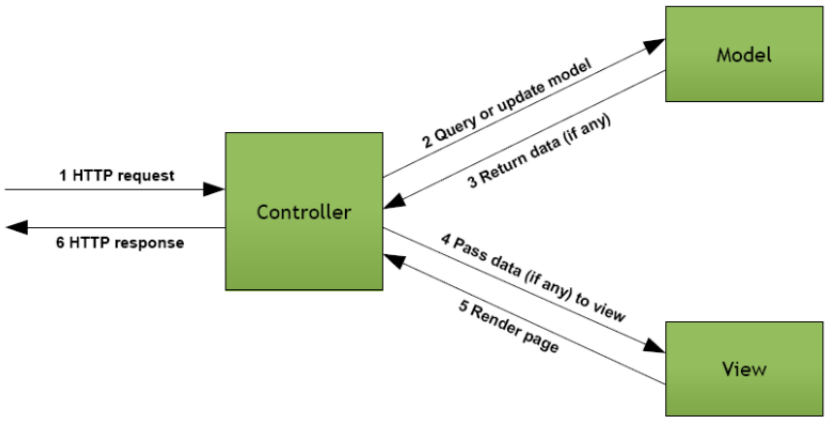

Abbildung 2.1: Spring MVC Controller

2.1 Delegation von Arbeit

Der Begriff **FrontController** bezeichnet ein Architekturmuster in der Softwaretechnik zur Erweiterung des Model-View-Controller-Architekturmusters. Ein FrontController dient in einer Webanwendung als Router. Alle Anfragen an die Webanwendung werden deshalb vom Front-Controller empfangen und er wird den Request zur Weiterbearbeitung an andere Controller (PageController) weiterleiten (siehe Abbildung 2). Der FrontController kann vor der Delegierung allgemeine Funktionen durchführen, wie zum Beispiel das Auslesen der Request- Parameter.

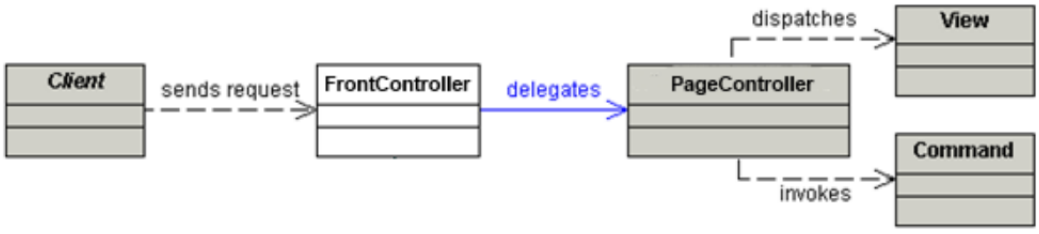

Abbildung 2.2: Spring FrontController

PageController Der PageController ist ein Objekt, das eine spezifische Anfrage für eine bestimmte Seite oder Aktion einer Website bearbeitet. Er wird normalerweise vom FrontController aufgerufen.

FrontController
Der FrontController in SpringMVC Im File "web.xmlfinden sie den Eintrag für den FrontController. Es ist das Servlet DispatcherServlet. Die Hauptaufgabe des FrontController ist es den Request entgegen zu nehmen (1), die Parameter zu verarbeiten und die Anfrage aufgrund einer entsprechenden Mapping Strategie (2) an eine dezidierte Klasse weiter zu leiten (3). Diese Klasse entspricht dem PageController. Der PageController wird schlussendlich mit dem Model und einem View Namen antworten (4).

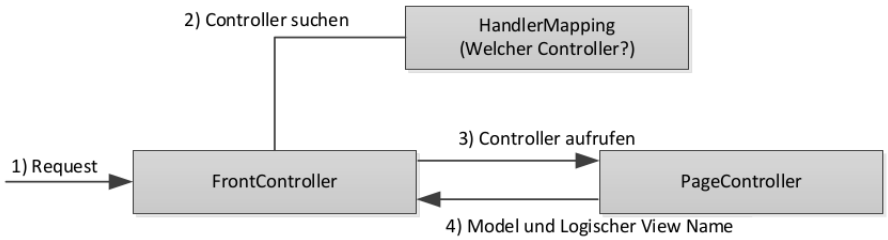

Abbildung 2.3: figure

PageController Der PageController übernimmt grundsätzlich 3 Arbeitschritte:

1. Mapping der Request URLs auf die Controller-Methoden Die Controller enthalten verschiedene Methoden, welche mit der Annotation @RequestMapping (mehr in Listing 1) einen entsprechenden Request entgegennehmen und diesen verarbeiten können.
2. Businesslogik aufrufen oder an Business-Services übergeben In der Regel wird nun die Verarbeitung an die Businesslogik (siehe in Listing 1 die Datenbankabfrage in den Zeilen #5 und #9) übergeben.
3. Die Antwort: Model und View Name In der Antwort des PageController's ist das Model, das üblicherweise in Form einer java.util.Map realisiert wird. Das Model-Objekt umfasst die Daten, welche in einer entsprechenden View gerendert werden sollen. Damit der FrontController die korrekte View Klasse auswählen kann, muss der PageController zusätzlich zum Model auch einen logischen View Name zurückschicken.

Listing 2.1: Controller Example

```
1  @Controller // Controller Tag
   @RequestMapping("/questionnaires") //Mapped URL innerhalb Servlet von FrontController.
   public class QuestionnaireController {
   @RequestMapping(method = RequestMethod.GET) //Welche HTTP Methode es drauf Antwortet.
   public String list(Model uiModel) {  // Automatische Object vom Typ Model, kann Key Value
       Paare beinhalten.
6  uiModel.addAttribute(("questionnaires",
   Questionnaire.findAllQuestionnaires()); //Key Value Setzung
   return "questionnaires/list"; //View Name
   }
   @RequestMapping(value = "/{id}", method = RequestMethod.GET)  // questionnaires/id
       behandelung Id ist ein Parameter aus GET Request.
11 public String show(@PathVariable("id") Long id, Model uiModel) {  // Konvertierung aus GET
       Request in gewunschten Typ mittels @PathVariable
   uiModel.addAttribute("questionnaire",
   Questionnaire.findQuestionnaire(id)); //setzen von Key Value Paaren.
   uiModel.addAttribute("itemId", id);
   return "questionnaires/show";  //View Name
16 }
   }
```

2.2 Statische Inhalte

Statische Ressourcen wie Bilder oder CSS-Stylesheets sollten nicht als dynamische Inhalte behandelt werden, also nicht über einer Servlet geladen werden, sondern direkt vom Webserver geliefert werden. Tomcat kann statische Inhalte direkt liefern. Dies erfordert aber eine Konfiguration im File "web.xml". Spring stellt eine andere, flexiblere Variante zur Verfügung. In Spring gibt es zwei Konfigurationseinstellungen um die Requests nach statische Inhalte einfach verarbeiten zu können.

Listing 2.2: Statische Inhalt Konfiguration

```
    <!-- Handles HTTP GET requests for /resources/** by efficiently
   serving up static resources -->
3  <mvc:resources location="/, classpath:/META-INF/web-resources/"
   mapping="/resources/**"/>
   #1
   <!-- Allows for mapping the DispatcherServlet to "/" by forwarding
   static resource requests to the container's default Servlet -->
```

```
8 <mvc:default-servlet-handler/>
```

Sie finden diese Einstellungen im File "webmvc-config.xml". Im File "header.jspxßoll z.B. das Banner-Image "banner-graphic.pngängezeigt werden. Das Image selber ist im Ordner "webapp/imagesäbgelegt. Über die Mapping-Definition in #1 wird die entsprechende URL zum Banner-Image zu /resources/images/banner-graphic.png

2.3 webmvc-config.xml

Damit ein PageController als Controller-Komponente von Spring erkannt werden kann, d.h. dass die Annotation @Controller korrekt interpretiert wird, muss Spring beim Bootstrap der Webapplikation ein Scanning durchführen und nach diesen Annotationen suchen. Dies erreicht man mit der Konfiguration

```
<context:component-scan>
```

Damit der FrontController den Request an den PageController übergeben kann, muss der FrontController die Strategie kennen wie das HandlerMapping ausgeführt werden soll. Spring kennt mehrere solche Strategien und jede ist in einer konkreten Implementation wie z.B. BeanNameUrlHandlerMapping umgesetzt. Arbeitet man nun mit der @RequestMapping Annotationen, muss das Spring Bean DefaultAnnotationHandlerMapping aktiviert werden. Dies erreicht man mit der Zeile

```
<mvc:annotation-driven/>
```

im Spring Konfigurationsfile "webmvc-config.xml".

3 Composite Views (Apache Tiles)

3.1 Motivation (bla)

Webseiten bestehen üblicherweise aus verschiedenen Content-Bereichen wie einem Header, einer Navigationsleiste oder einem Footer. Die Webseite selber setzt sich dann aus diesen verschiedenen Bereichen zusammen, wobei die Struktur mehrheitlich über alle Seiten gleich bleibt. Header, Footer oder auch Navigation bleiben unverändert oder ändern sich kaum - nur der zentrale Content wechselt je nach Kontext. Damit die Webapplikation gut erweiterbar bleibt und der Wartungsaufwand tief gehalten werden kann, ist auch hier ein komponentenorientierter Ansatz sinnvoll. Jeder Content-Bereich sollte als eine unabhängige Einheit vorliegen. Diese Einheiten können dann beliebig zu einer Komposition zusammengesetzt werden. Änderungen in der Einheit selber werden dann automatisch in allen Kompositionen sichtbar, welche die entsprechende Einheit nutzen.

3.2 Composite View Pattern

- bindet Unteransichten dynamisch zu einer zusammengesetzten Ansicht ein.
- Code für jede Unteransicht nur einmal vorhanden sein und jede Anpassung schlägt sich automatisch in allen zusammengesetzten Ansichten nieder.
- Die Klasse BasicView stellt die allgemeine Form einer Ansicht dar und kann entweder eine View (einzelne Ansicht) oder eine CompositeView (zusammengesetzte Ansicht) sein. Die CompositeView ist aus BasicView's zusammengesetzt; das bedeutet, dass sie sowohl einzelne Views als auch andere CompositeView's beinhalten kann

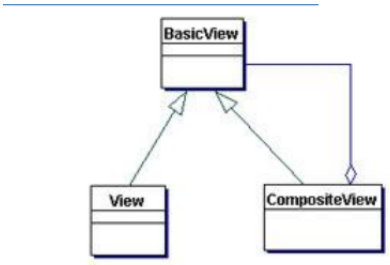

Abbildung 3.1: Composite View Pattern

3.3 Tiles Ansatz

Mit Hilfe von Tiles werden Design Vorlagen erstellt, in denen Referenzen zu Subviews überall dort stehen wo der eigentliche Inhalt in andere Views ausgelagert werden soll. Das hat den enormen Vorteil, dass man gleiche Vorlagen mehrfach verwenden kann. Dadurch wird es einfacher sein das komplette Design der Webapplikation bei Bedarf an neue Anforderungen anzupassen.

SpringMVC unterstützt die Tiles Integration effizient. Ein entsprechender View Resolver kann aus dem logischen View-Namen - er wird von der Controller-Methode geliefert (wie in Arbeitsblatt 4 erfahren) - die entsprechende Tiles Definitionen finden und das Tiles Framework generiert anschliessend die entsprechende Web-View mit allen notwendigen Subview's, die selber wiederum Tiles sind.

Der View-Resolver wird im File "webmvc-config.xml"konfiguriert. Zusätzlich zur dieser Konfiguration muss Tiles selber auch konfiguriert werden. In erster Linie muss Tiles wissen, wo die Tiles-Definitionen liegen. Hier der Eintrag aus dem File "WEB-INF/spring/webmvc-config.xml":

Listing 3.1: View Resolver Config Tiles

```
<bean class="org.springframework.web.servlet.view.tiles2.TilesConfigurer" id="
    tilesConfigurer">
```

```
<property name="definitions">
<list>
<value>/WEB-INF/layouts/layouts.xml</value>
#1
<!-- Scan views directory for Tiles configurations -->
<value>/WEB-INF/views/**/views.xml</value>
#2
</list>
</property>
</bean>
```

Listing 3.2: CompositeView mit Tiles(default.jspx)

```
//DEFINIERT SUBVIEW PLAZHALTER
 <body class="tundra spring">
<div id="wrapper">
<tiles:insertAttribute name="header" ignore="true" />
<tiles:insertAttribute name="menu" ignore="true" />
<div id="main">
<tiles:insertAttribute name="body"/>
<tiles:insertAttribute name="footer" ignore="true"/>
</div>
</div>
/body>
```

Zu diesen Referenzen müssen konkrete View's vorhanden sein, damit die gesamte Seite aufgebaut werden kann. Das Default-Layout der flashcard-Applikation findet man im File "WEB- INF/layouts/layouts.xml", mit folgenden Tiles Definitionen:

Listing 3.3: Layout Beschreibung

```
<!--
FIXIERUNG VON GEWISSE ELEMENTE IN DEM DER ATTRIBUT EIN "DEFAULTWERT" BEKOMMT
--!>
 <definition name="default" template="/WEB-INF/layouts/default.jspx">
<put-attribute name="header" value="/WEB-INF/views/header.jspx" />
<put-attribute name="menu" value="/WEB-INF/views/menu.jspx" />
<put-attribute name="footer" value="/WEB-INF/views/footer.jspx" />
</definition>
```

Listing 3.4: Auszug aus views.xml

```
 <definition extends="default" name="index">
<put-attribute name="body" value="/WEB-INF/views/index.jspx"/>
</definition>
```

Die Wahl der Tiles-View durch den FrontController sollte nun klar sein. Gibt ein ApplicationController einen logischen View-Name zurück, wird dieser Name verwendet, um innerhalb der Tiles Konfiguration nach einer gleichnamigen Definition zu suchen. Wird diese Definition gefunden, kann die komplette Webseite aus den einzelnen Tiles (Kacheln) zusammengesetzt werden.

4 JSTL

Template View" Design Pattern Renders information into HTML by embedding markers in an HTML page. Mit dem Template will man in einer Webapplikation eine Trennung zwischen HTML und Programmcode erwirken. Es gibt verschiedene Gründe, um eine solche Trennung anzustreben:

- HTML Design und Applikationsentwicklung verlangen unterschiedliches Know-how HTML Design und Applikationsentwicklung sind unterschiedliche Rollen bei der Entwicklung einer Webapplikation. Diese Rollen werden oft durch verschiedene Fachleute zu verschiedenen Zeiten wahrgenommen. Durch die Separierung von HTML vom Programmcode kann jede Person an seinen eigenen Files arbeiten.
- Development Tool Support Durch die Separierung von HTML vom Programmcode können HTML WYSIWYG Editoren eingesetzt werden, welche die Gestaltung des User Interfaces stark unterstützen können. Designer sind keine Programmierer!

Mit der Expression Language erhält man einen einfachen Zugriff auf dynamische Werte wie die Anfrageparameter oder die Ergebnisse der Geschäftslogik, d.h. das Model. Expression Language-Ausdrücke werden durch spezielle Delimiter gekennzeichnet. Sie beginnen entweder mit "$öder mit "#{ünd enden mit "}":

Mit der Expression Language kann man den Zugriff auf das Modell vereinfachen. Jedoch fehlt bei der EL die Möglichkeiten einfache Kontrollstrukturen einzubinden, die es in der Präsentationslogik zum Teil braucht

4.1 JSTL Java Tag Library

Name	Prefix	URI	Aufgabe
core	c	http://java.sun.com/jsp/jstl/core	Basistags, die vor allem Programmablauf-Tags wie Schleifenkonstrukte oder Verzweigungen
fmt	fmt	http://java.sun.com/jsp/jstl/fmt	Tags zur Formatierung von Datums-, Währungs- und Zahlwerten und zur Internationalisierung
xml	x	http://java.sun.com/jsp/jstl/xml	Geeignet zur Bearbeitung von XML- Dokumenten. Enthält u.a. wie core Schleifen- und Verzweigungstags. Der Unterschied zu den core-Tags liegt darin, dass XPath-Ausdrücke zur Bedingungsentscheidung bzw. zur Feststellung der Menge dienen, über die iteriert werden soll
sql	sql	http://java.sun.com/jsp/jstl/sql	Tag-Bibliothek zum Ansprechen einer Datenbank direkt aus der JSP heraus
functions	fn	http://java.sun.com/jsp/jstl/functions	Die Tags dieser Bibliothek wenden Funktionen auf Expression Language- Ausdrücke an. Es handelt sich mit einer Ausnahme durchweg um String- Manipulationen

Mit Einführung der Servlet-API 2.4 und der Verabschiedung von JSP 2.0 ist es möglich, komplett auf JSP-Skriptelemente zu verzichten. JSPX-Dokumente entsprechen wohlgeformten XML- Dokumenten, deren Tag-Elemente durch eindeutige Namensräume definiert sind. So ist es beispielsweise möglich, XHTML-konforme Ausgaben zu erzeugen, ohne dass Tag-Bezeichner kollidieren.

4.2 JSP als Servlet

Für die Entwicklung mit JavaServer Pages ist es wichtig zu verstehen, welche Phasen eine JSP- Datei bei ihrem Weg durch den JSP-Container durchläuft. Es ist wichtig festzuhalten, dass jeder JSP-Container stets auch ein Servlet-Container sein muss. Jede JSP wird im Verlaufe ihres Lebenszyklus vom Container zunächst in ein Servlet transformiert. Zur Ausführung kommen anschliessend Servlets

- Servlet Quellcode erstellt

  Übernommen von Container am ersten Aufruf.

- Änderung der JSP heisst neue Übersetzung wie oben.

  Aber hier ist auch mitunter Vorsicht angebracht. Eine Datei erst umzubenennen, dann vorübergehend eine andere Datei gleichen Namens zu verwenden (bspw. eine spezielle Debug- Version oder eine andere Index-Seite zur Bekanntgabe von Wartungsarbeiten, etc.) und anschliessend die alte Datei zurück zu kopieren, funktioniert nicht.

- n dieser Phase erkennt der Übersetzer nur Fehler, die direkt mit der JSP-Syntax zusammenhängen. Bspw. wenn ein öffnender Skriptlet-Tag nicht geschlossen wird, falsche Direktiven verwendet werden, Tag-Libraries nicht gefunden werden etc. Fehler, die im Java-Code selber enthalten sind, werden in dieser Phase noch nicht gefunden. Daher ist die Anzeige der Fehler in dieser Phase und der Fehler der folgenden Phase auch häufig unterschiedlich detailliert.

- n der Compile-Phase wird vom Container der gewöhnliche Java-Compiler angeworfen, um aus dem generierten Servlet-Quellcode eine Java-Class Datei zu erzeugen. In dieser Phase werden nun die Fehler im ggf. vorhandenen Java-Code der JSP entdeckt, wie Syntax-Fehler, nicht vorhandene Klassen, die per import-Statements referenziert werden, usw.

5 Formulare

5.1 Motivation

Zu jedem Formular gehört ein Form Bean (oder Form Object oder Backing Bean). Hier ist es die Domain Entität Questionnaire:

```
 public class Questionnaire {
private String subject;
private String description;
...
```

Entität als Form Bean? Es ist problemlos möglich das selbe POJO als Form Bean und als Domain Entität einzusetzen. Jedoch, in bezug auf die Applikationsarchitektur und dem Paradima SSeparation of conecrnïst es eine bessere Praxis zwei unterschiedliche Klassen zu verwenden, vor allem dann wenn es keine eindeutige Beziehung zwischen Entität und Form Bean gibt. Ein Form Bean kann ja problemlos ein Aggregat verschiedener Entitäten sein!

Web Formular verarbeiten

Listing 5.1: Web Formular verarbeiten Spring

```
@Controller
@RequestMapping("/questionnaires")
public class QuestionnaireController {
@RequestMapping(params = "form", method = RequestMethod.GET)#1
public String createForm(Model uiModel) {#2
uiModel.addAttribute("questionnaire", new Questionnaire());#3
return "questionnaires/create";#4
}
...
@RequestMapping(method = RequestMethod.POST) #5
public String create(@Valid Questionnaire questionnaire, ...)#6 {
...
return "redirect:/";#7
}
}
```

1. Mapping für Handler-Methode, die auf einen GET-Request mit dem Parameter form gemappt wird. Zum Beispiel localhost:8080/flashcard/questionnaires?form
2. Handler-Methode mit einem Model Objekt als Parameter, das an die View weitergegeben werden kann.
3. Hier wird ein neues Form Bean erzeugt und unter dem Attribute questionnaire im Model abgelegt.
4. Logischer View Name wird retourniert.
5. Mapping für Handler-Methode, für den POST-Request. Das Mapping entspricht /questionnaires. Handler-Methode, um den Questionnaire zu verarbeiten. Der Questionnaire enthält die im Formular abgefüllten Daten.
6. Redirect zu einer Resultat-Seite. (siehe Aufgabe 5 "Double Submit Problem")

View für Controller Methode

Listing 5.2: View mit Form Spring

```
 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div ...
xmlns:field="urn:jsptagdir:/WEB-INF/tags/form/fields"
xmlns:form="urn:jsptagdir:/WEB-INF/tags/form"
... version="2.0">
<form:create id="fc_ch_fhnw_edu_flashcard_domain_Questionnaire"#1
```

```
  modelAttribute="questionnaire" path="/questionnaires">#2
  <field:input field="subject"#3
  id="c_ch_fhnw_edu_flashcard_domain_Questionnaire_subject"
10 required="true" />#4
  <field:input field="description"#5
  id="c_ch_fhnw_edu_flashcard_domain_Questionnaire_description"
  required="true" />
  </form:create>
15 </div>
```

- form Tag (aus der Spring Roo Tag Library) erzeugt ein HTML Formular. Das Attribute id gibt dem Formular-Element eine Identifikation. Spring Roo nutzt diese Identifikation auch, um aus dem Wert das internationalisierte Label abzuleiten.
- Referenz auf das Model-Attribut questionnaire, das im Model entsprechend gesetzt wurde. URL-Path auf /questionnaires gesetzt, so dass der POST-Request auf die Methode create() des QuestionnaireController mappt. Das Attribut modelAttribute in Zeile #2 referenziert das Form-Bean, welches im Model abgelegt ist. Das Model-Objekt wird beim Umgang mit dem Formular in beide Richtungen (Outbound - Inbound) eingesetzt.
- Hier wird ein HTML-Textfeld erzeugt, um das Thema subject eingeben zu können. Das Attribut field referenziert das Property subject des Form-Bean Questionnaire. Mit dem Attribut required kann eine Validierung mittels JavaScript bereits im Browser aktiviert werden.
- Hier wird ein HTML-Textfeld erzeugt, um die Beschreibung description zum Questionnaire eingeben zu können. Das Attribut field referenziert das Property description des Form- Bean Questionnaire.

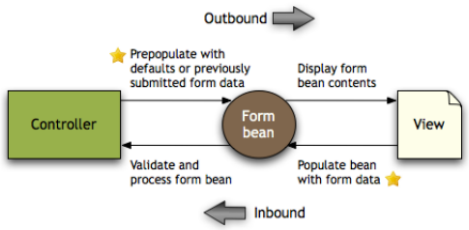

Abbildung 5.1: Form Bean

5.2 Validierung von Formular Daten

Formular Daten können vom Benutzer falsch eingegeben werden. Um einen konsistenten Datenbestand halten zu können, ist es deshalb sehr wichtig, dass diese Daten validiert werden, bevor sie an die Geschäftslogik oder sogar in die Datenbank gehen. Und falls Fehler bei der Validierung auftreten, muss der Benutzer entsprechend orientiert werden. Mit dem JSR-303 “Bean Validation” hat Java nun seit Java EE Version 6 ein mächtiges und Schichten übergreifendes Mittel an die Hand bekommen, um komfortabel schon zur Entwurfszeit bekannte Constraints per Annotation auf Java-Beans zu legen. Die Java-Bean “weiss” damit also selbst, welche Daten sie annehmen darf oder nicht. Die Referenzimplementierung zum JSR-303 kommt von Hibernate. Da der JSR-303 mit Java EE 6 eingeführt wurde, ist er natürlich Bestandteil aller zertifizierten Java EE 6 Application Server wie JBoss, Geronimo und weitere. Erfreulicherweise lässt sich der JSR-303 aber auch im Java SE Umfeld mit wenig Aufwand einsetzen und bietet auch hier allen Komfort und Flexibilität. Auch Spring 3 bietet eine vollständige Unterstützung von JSR 303.

Schicht	Vorteil	Nachteil
Datenbank	sehr zentrale Validierung.eine Umgehung der Validierung ist fast nicht möglich	Validierung sehr weit unten im Call-Stack.Netzwerk wird belastet, da eine Client-Server Kommunikation aufgebaut wird, die Antwort an den Benutzer kann dauern.
Controller	keine Zugriff auf das Backend-System notwendig,unabhängig vom Client.	Validierung auf Web-Clients beschränkt Netzwerk wird belastet, da eine Client-Server Kommunikation aufgebaut wird
View	sehr schnelles Feedback ist möglich, da keine Netzwerkkommunikation notwendig. Entlastung des Servers keine Netzwerkkommunikation	hier JavaScript als weitere Programmiersprache Client muss JavaScript unterstützen aus Sicherheitsgründen ist eine Validierung auf Serverseite immer noch notwendig - Aufwand.

5.3 View Validierung

Bei der flashcard-Applikation kann man ein erstes Validierungselement erkennen (siehe Abbildung 2). Hier wird die Validierung bereits in der View vorgenommen. Der Benutzer erhält augenblicklich ein Feedback : DOJO Bibliothek : custom jstl tags eingesetzt die javascript code im View generieren.

Controller Validierung:

Listing 5.3: Controller Validierung Spring

```
 public String create(@Valid Questionnaire questionnaire, #1
BindingResult bindingResult, Model uiModel) {#2
if (bindingResult.hasErrors()) {
populateEditForm(uiModel, questionnaire);#3
return "questionnaires/create";#4
}
}
```

1. @Valid ist eine JSR-303 Annotation. Aufgrund der Annotationen im Form Bean (hier auch Domain Entität) wird der Parameter validiert.
2. Das Validierungsresultat wird in bindingResult abgelegt
3. Tritt bei der Validierung ein Fehler auf muss das uiModel neu gefüllt werden. Hier werden die Eingabewerte der Formulardaten wieder in das Form Bean zurückgeschrieben.
4. Wiederum wird die View zum Formular selber referenziert. Das Formular enthält dank dem Modell die eingegeben aber falschen Werte, die nun korrigiert werden können.

5.4 Validierung auf Datenschicht

Forcieren sie eine Validation Exception auf der Datenbankschicht. Ergänzen sie dazu z.B. die Property ßubject"der Entität Questionnaire mit der JSR-303 Annotation

```
@Size(mix = 1, max = 50)
```

Entfernen sie in der Methode create() des QuestionnaireController die @Valid Annotation und setzen sie im File "WEB-INF/views/questionnaires/create.jspx"die required-Attribute auf false". Die Applikation wird mit einer Exception unter dem logischen View Name üncaughtExceptionäntworten. Sie müssen in "views.xmläuch kontrollieren, ob unter diesem View-Name die Composite-View korrekt aufgebaut werden kann! Starten sie die flashcard-Applikation. Was können sie erkennen? Aus dem obigen Beispiel wird offensichtlich, dass die @Valid Annotation in einer Controller-Klasse (@Controller) eine zentrale Rolle bei der Validierung der Formulardaten einnimmt. Die mit @Valid versehenen Methodenparameter werden über das JSR-303 Framework validiert und das Spring Framework wird das Validierungsresultat in eine Instanz BindingResult schreiben. So kann bereits auf Schicht Controller eine Prüfung stattfinden, bevor ein Datenbankzugriff ausgelöst werden muss.

5.4.1 Validierung Config

Für die Validierung braucht es einen entsprechenden Validator. Dieser kann manuell instanziiert werden oder es kann ein JSR-303 Validator genutzt werden. Spring stellt einen Validator zur Verfügung, der in den meisten Fällen genügt. Konfiguriert wird dieser Validator mittels

```
<mvc:annotation-driven/>
```

Es kann jedoch die Situation auftreten, wo ein eigener Validator nützlich ist.

5.5 Formular Daten Speichern

Listing 5.4: Speichern mit POST Method Controller

```
 @RequestMapping(method = RequestMethod.POST)
public String create(@Valid Questionnaire questionnaire,
BindingResult bindingResult, Model uiModel,
HttpSession session, HttpServletRequest httpServletRequest) {
if (bindingResult.hasErrors()) {#1
uiModel.addAttribute("questionnaire", questionnaire);
return "questionnaires/create";
}
uiModel.asMap().clear();#2
questionnaire.persist();#3
return "redirect:/";
}
```

1. Zuerst muss geprüft werden, ob das Binding und die Validierung fehlerlos verlief.
2. Das Model wird geleert.
3. Die Entität wird gespeichert. Es ist in diesem Falle auch das Form Bean.

5.6 Double Submit Problem

Refresh auf View mit Formular über POST macht dann nochmals POST. Gelöst mit ein redirect nach post auf eine View die über GET im Controller funktioniert.

6 Sichere Webapps

6.1 Spring Security

Eine deklarative Beschreibung der relevanten Sicherheitsaspekte wie Authentifizierung und Autorisierung bei einer SpringMVC-Applikation erlaubt. nhand der flashcard-Applikation wird besprochen, wie eine Webapplikation gesichert werden kann. Dabei ist zu beachten, dass in der Applikation alle sicherheits-relevanten Artefakten vorhanden sind. Diese sind mit dem ROO-Befehl security setup geladen worden und umfassen:

- spring-security-*.jar Bibliotheken. Sie werden mittels Maven geladen
- Konfigurationsdatei applicationContext-security.xml (in Directory META-INF/spring)
- Webpage login.jspx, mit den entsprechenden Updates für Tiles.

6.1.1 Authenticationsschritte

Webapplikation empfängt einen nicht authentifizierten HTTP-Request Aus sicherheitstechnischen Überlegungen ist es deshalb sinnvoll, den Sicherheitsmechanismus um diese HTTP-Requests aufzubauen. Mit Spring Security ist dies durch eine einfache deklarative Beschreibung möglich. Mit Hilfe des Files applicationContext-security.xml erkannt man, wie eine URL geschützt wird. Der Eintrag

```
<intercept-url pattern="/member/**" access="isAuthenticated()" />
```

legt z.B. fest, dass jede URL, die mit /member startet, nur für authentifizierte Benutzer sichtbar sein soll.

```
<intercept-url pattern="/**" access="isAuthenticated()" method="POST" />
```

Mit diesem Pattern wird ausgedrückt, dass alle POST-Requests authentifiziert sein müssen. Falls nun verschiedene Patterns auf einen HTTP-Request zutreffen, wird Spring Security dasjenige auswählen, das die strikteste Regel definiert.

Listing 6.1: spring security xml

```
    <intercept-url pattern="/questionnaires/**"
access="isAuthenticated()" method="GET" />
<intercept-url pattern="/resources/**"
access="permitAll" />
```

Authorizierung Bei alle URls wird ein springSecurityFilterChain benutzt mit mehreren Filter. Diese Filter sind z.B. LogoutFilter, AnonymousAuthenticationFilter, UsernamePasswordAuthenticationFilter, SessionManagementFilter. Innerhalb dieser Filterkette wird zuerst überprüft, ob ein SecurityContext in der Session abgelegt ist. Der SecurityContext enthält u.a. die Credentials wie Benutzername, Passwort oder Rolle. Ist kein SecurityContext vorhanden, wird der AnonymousAuthenticationFilter einen entsprechenden Context für den Anonymous-Benutzer anlegen. Mit diesem Context kann der Filter FilterSecurityInterceptor überprüfen, ob der entsprechende Benutzer die angeforderte Ressource nutzen darf. Ist dies nicht der Fall, wird eine AccessDeniedException geworfen. Diese Exception löst in der flashcard-Applikation ein Redirect auf die Login-Page aus.

Leitung der Request auf Login Page

Listing 6.2: security xml login weiterleitung

```
<form-login
login-processing-url="/resources/j_spring_security_check"
login-page="/login"
authentication-failure-url="/login?login_error=t"/>
```

Listing 6.3: login jspx spring security

```
<spring:url value="/resources/j_spring_security_check" var="form_url" />
<form name="f" action="${fn:escapeXml(form_url)}" method="POST">
<div>
...
<input id="j_username" type='text' name='j_username' ... />
...
<input id="j_password" type='password' name='j_password' ... />
...
</form>
```

j_ variablen sind gemäss Spezifikation

Listing 6.4: spring security tiles Änderungen

```
<definition extends="public" name="login">
<put-attribute name="body" value="/WEB-INF/views/login.jspx"/>
<put-attribute name="header-center" value="/WEB-INF/views/welcome.jspx"/>
<put-attribute name="header-right" value="/WEB-INF/views/empty.jspx"/>
</definition>
```

Verarbeitung Login Request Der Login-Request wird über die Logik hinter / resources / j_spring_security_check verarbeitet. Dabei wird die Filterkette hinter springSecurityFilterChain abgearbeitet. Der UsernamePasswordAuthenticationFilter überprüft, ob die eingegebenen Werte einem vorhandenen Account entsprechen. Trifft dies zu, wird der entsprechende SecurityContext erzeugt und mit dem Filter HttpSessionSecurityContextRepository in die Session abgespeichert.

Es wird innerhalb der login.jspx-Seite einen Test auf den Request-Parameter login_error"vorgenommen. Falls ein Wert gesetzt ist - hier z.B. t"(er kann beliebig sein!) - wird eine Fehlermeldung am Anfang der Login-Seite mit dem Fehlergrund angezeigt.

Listing 6.5: error Behandelung spring security error output

```
  <c:if test="${not empty param.login_error}">
<div class="errors">
<p>
<spring:message code="security_login_unsuccessful" />
<c:out value="${SPRING_SECURITY_LAST_EXCEPTION.message}" />
.
</p>
</div>
</c:if>
```

Authorizierung HTTP Request Nach einer erfolgreichen Authentifizierung ist der notwendige SecurityContext in der Session gespeichert und kann bei allen weiteren Requests für die Autorisierung beigezogen werden. Es ist auch in diesem Falle der Filter FilterSecurityInterceptor, der die Autorisierung mit Hilfe der Regeln aus dem File applicationContext-security.xml überprüft.

Logout Security xml :

```
<logout logout-url="/resources/j_spring_security_logout" />
```

Listing 6.6: spring security tag lib

```
<security:authorize access="! isAuthenticated()">
<c:out value=" | "/>
<a href="login">Login</a>
</security:authorize>
<security:authorize access="isAuthenticated()">
<c:out value=" | "/>
<a href="resources/j_spring_security_logout">Logout</a>
</security:authorize>
```

6.2 Sicherheit mit Servlet

Programmtechnisch definiert das Interface javax.servlet.http.HttpServletRequest die Methoden getRemoteUser() isUserInRole() getUserPrincipal()

Tomcat verwendet das Konzept eines Realms (Gebiet, Reich). Ein Realm stellt einen Bestand von Benutzern, Passwörter und Rollen dar. Fünf Arten von Realms sind standardmässig vorhanden: JDBCRealm, DataSourceRealm, JNDIRealm, MemoryRealm und JAASRealm. Wir werden den MemoryRealm einsetzen. Hier sind die Authentifizierungsinformationen in einer XML-Datei abgelegt (

`$TOMCAT_HOME/conf/tomcat-users.xml`

). Hier ein Beispiel:

Listing 6.7: users.xml servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<tomcat-users>
<role rolename="admin"/>
<role rolename="user"/>
<user username="admin" password="admin" roles="admin,user"/>
<user username="user" password="user" roles="user"/>
</tomcat-users>

<security-constraint>
<web-resource-collection>
<web-resource-name>Questionnaires Resource</web-resource-name>
<url-pattern>/questionnaires/*</url-pattern>
<http-method>GET</http-method>
<http-method>POST</http-method>
</web-resource-collection>
<auth-constraint>
<role-name>admin</role-name>
<role-name>user</role-name>
</auth-constraint>
</security-constraint>

<security-role>
<role-name>admin</role-name>
</security-role>
<security-role>
<role-name>user</role-name>
</security-role>
<login-config>
<auth-method>FORM</auth-method>
<form-login-config>
<form-login-page>/login</form-login-page>
<form-error-page>/login?login_error=t</form-error-page>
</form-login-config>
</login-config>
```

7 Sessions

For example, when clients at an online store add an item to their shopping carts, how does the server know what's already in the carts? Similarly, when clients decide to proceed to checkout, how can the server determine which previously created shop- ping carts are theirs? These questions seem very simple, yet, because of the inadequacies of HTTP, answering them is surprisingly complicated. There are three typical solutions to this problem: cookies, URL rewriting, and hidden form fields. The following subsections quickly summarize what would be required if you had to implement session tracking yourself (without using the built-in session-tracking API) for each of the three ways.

7.1 Cookies

Listing 7.1: Java Using Session

```
String sessionID = makeUniqueString(); HashMap sessionInfo = new
HashMap();
HashMap globalTable = findTableStoringSessions();
globalTable.put(sessionID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessionID);
sessionCookie.setPath("/"); response.addCookie(sessionCookie);
```

Then, in later requests the server could use the globalTable hash table to asso- ciate a session ID from the JSESSIONID cookie with the sessionInfo hash table of user-specific data. Using cookies in this manner is an excellent solution and is the most widely used approach for session handling. Still, it is nice that servlets have a higher-level API that handles all this plus the following tedious tasks:

- Extracting the cookie that stores the session identifier from the other cookies (there may be many cookies, after all).
- Determining when idle sessions have expired, and reclaiming them.
- Associating the hash tables with each request.Generating the unique session identifiers.

7.2 URL Rewriting

For example, with http://host/path/file.html;jsessionid=a1234, the session identifier is attached as jsessionid=a1234, so a1234 is the ID that uniquely identifies the table of data associated with that user. URL rewriting is a moderately good solution for session tracking and even has the advantage that it works when browsers don't support cookies or when the user has disabled them. However, if you implement session tracking yourself, URL rewriting has the same drawback as do cookies, namely, that the server-side program has a lot of straightforward but tedious processing to do. Even with a high-level API that handles most of the details for you, you have to be very careful that every URL that references your site and is returned to the user (even by indirect means like Location fields in server redirects) has the extra information appended. This restriction means that you cannot have any static HTML pages on your site (at least not any that have links back to dynamic pages at the site). So, every page has to be dynamically generated with servlets or JSP. Even when all the pages are dynamically generated, if the user leaves the session and comes back via a bookmark or link, the session information can be lost because the stored link contains the wrong identifying information.

7.3 Hidden Form Fields

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="a1234">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. This hidden field can be used to store information about the session but has the major disadvantage that it only works if every page is dynamically generated by a form submission. Clicking on a regular

```
(<A HREF...>)
```

hypertext link does not result in a form submission, so hidden form fields cannot support general session tracking, only tracking within a specific series of operations such as checking out at a store.

7.3.1 Servlets and Session Tracking

Servlets provide an outstanding session-tracking solution: the HttpSession API. This high-level interface is built on top of cookies or URL rewriting. All servers are required to support session tracking with cookies, and most have a setting by which you can globally switch to URL rewriting. Either way, the servlet author doesn't need to bother with many of the implementation details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store arbitrary objects that are associated with each session.

7.4 Session Tracking basics

1. Accessing the session object associated with the current request. Call request.getSession to get an HttpSession object, which is a simple hash table for storing user-specific data.

```
 HttpSession session = request.getSession(false);
if (session == null) { printMessageSayingCartIsEmpty();
} else { extractCartAndPrintContents(session);
}
```

---

2. Looking up information associated with a session. Call getAttribute on the HttpSession object, cast the return value to the appropriate type, and check whether the result is null.

```
 HttpSession session = request.getSession();
SomeClass value = (SomeClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session
value = new SomeClass(...);
session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

In most cases, you have a specific attribute name in mind and want to find the value (if any) already associated with that name. However, you can also discover all the attribute names in a given session by calling getAttributeNames, which returns an Enumeration.

3. Storing information in a session. Use setAttribute with a key and a value. Be aware that setAttribute replaces any previous values; to remove a value without supplying a replacement, use removeAttribute. This method triggers the valueUnbound method of any values that implement HttpSessionBinding- Listener. Following is an example of adding information to a session. You can add information in two ways: by adding a new session attribute (as with the bold line in the example) or by augmenting an object that is already in the session (as in the last line of the example). This distinction is fleshed out in the examples of Sections 9.7 and 9.8, which contrast the use of immutable and mutable objects as session attributes

```
 HttpSession session = request.getSession(); SomeClass value =
(SomeClass)session.getAttribute("someIdentifier");
if (value == null) { // No such object already in session value = new
SomeClass(...); session.setAttribute("someIdentifier", value);
}
doSomethingWith(value);
```

---

4. Discarding session data. Call removeAttribute to discard a spe- cific value. Call invalidate to discard an entire session. Call logout to log the client out of the Web server and invalidate all ses- sions associated with that user.
   - Remove only the data your servlet created. You can call removeAttribute("key") to discard the value associated with the specified key. This is the most common approach.
   - Delete the whole session (in the current Web application). You can call invalidate to discard an entire session. Just remember that doing so causes all of that user's session data to be lost, not just the session data that your servlet or JSP page created. So, all the servlets and JSP pages in a Web application have to agree on the cases for which invalidate may be called.
   - Log the user out and delete all sessions belonging to him or her. Finally, in servers that support servlets 2.4 and JSP 2.0, you can call logout to log the client out of the Web server and invalidate all sessions (at most one per Web application) associated with that user. Again, since this action affects servlets other than your own, be sure to coordinate use of the logout command with the other developers at your site.

7.5 Session Tracking API

Although the session attributes (i.e., the user data) are the pieces of session infortion you care most about, other information is sometimes useful as well. Here is a summary of the methods available in the HttpSession class.

public Object getAttribute(String name) This method extracts a previously stored value from a session object. It returns null if no value is associated with the given name.

public Enumeration getAttributeNames() This method returns the names of all attributes in the session. public void setAttribute(String name, Object value) This method associates a value with a name. If the object supplied to setAttribute implements the HttpSessionBindingListener interface, the object's valueBound method is called after it is stored in the session. Simi- larly, if the previous value implements HttpSessionBindingListener, its valueUnbound method is called.

public void removeAttribute(String name) This method removes any values associated with the designated name. If the value being removed implements HttpSessionBindingListener, its valueUnbound method is called.

public void invalidate() This method invalidates the session and unbinds all objects associated with it. Use this method with caution; remember that sessions are associated with users (i.e., clients), not with individual servlets or JSP pages. So, if you invalidate a session, you might be destroying data that another servlet or JSP page is using.

public void logout() This method logs the client out of the Web server and invalidates all sessions associated with that client. The scope of the logout is the same as the scope of the authentication. For example, if the server implements single sign-on,calling logout logs the client out of all Web applications on the server and invalidates all sessions (at most one per Web application) associated with the client. For details, see the chapters on Web application security in Volume 2 of this book.

public String getId() This method returns the unique identifier generated for each session. It is useful for debugging or logging or, in rare cases, for programmatically moving values out of memory and into a database (however, some J2EE servers can do this automatically).

public boolean isNew() This method returns true if the client (browser) has never seen the session, usually because the session was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions. The main reason for mentioning this method is to steer you away from it: isNew is much less useful than it appears at first glance. Many beginning developers try to use isNew to determine whether users have been to their servlet before (within the session timeout period), writing code like the following:

Listing 7.2: Wring isNew for session

```
 HttpSession session = request.getSession(); if (session.isNew()) {
doStuffForNewbies();
} else {
doStuffForReturnVisitors();
// Wrong!
}
```

Wrong! Yes, if isNew returns true, then as far as you can tell this is the user's first visit (at least within the session timeout). But if isNew returns false, it merely shows that they have visited the Web application before, not that they have visited your servlet or JSP page before.

public long getCreationTime() This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was first built. To get a value useful for printing, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.

public long getLastAccessedTime() This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was last accessed by the client.

public int getMaxInactiveInterval()

public void setMaxInactiveInterval(int seconds) These methods get or set the length of time, in seconds, that a session should go without access before being automatically invalidated. A negative value specifies that the session should never time out. Note that the timeout is maintained on the server and is not the same as the cookie expiration date. For one thing, sessions are normally based on in-memory cookies, not persistent cookies, so there is no expiration date. Even if you intercepted the JSESSIONID cookie and sent it out with an expiration date, browser sessions and server sessions are two distinct things. For details on the distinction, see the next section.

7.6 Browser vs Server Sessions

By default, session-tracking is based on cookies that are stored in the browser's memory, not written to disk. Thus, unless the servlet explicitly reads the incoming JSESSIONID cookie, sets the maximum age and path, and sends it back out, quitting the browser results in the session being broken: the client will not be able to access the session again. The problem, however, is that the server does not know that the browser was closed and thus the server has to maintain the session in memory until the inactive interval has been exceeded.

The analogous situation in the servlet world is one in which the server is trying to decide if it can throw away your HttpSession object. Just because you are not currently using the session does not mean the server can throw it away. Maybe you will be back (submit a new request) soon? If you quit your browser, thus causing the browser-session-level cookies to be lost, the session is effectively broken. But, as with the case of getting in your car and leaving Wal-Mart, the server does not know that you quit your browser. So, the server still has to wait for a period of time to see if the session has been abandoned. Sessions automatically become inactive when the amount of time between client accesses exceeds the interval specified by getMaxInactiveInterval. When this happens, objects stored in the HttpSession object are removed (unbound). Then, if those objects implement the HttpSessionBindingListener interface, they are automatically notified. The one exception to the "the server waits until sessions time out" rule is if invalidate or logout is called. This is akin to your explicitly telling the Wal-Mart clerk that you are leaving, so the server can immediately remove all the items from the session and destroy the session object.

7.7 URL Encoding

URL rewriting instead? How will your code have to change?

The goods news: your core session-tracking code does not need to change at all. The bad news: lots of other code has to change. In particular, if any of your pages contain links back to your own site, you have to explicitly add the session data to the URL. Now, the servlet API provides methods to add this information to whatever URL you specify. The problem is that you have to call these methods; it is not technically feasible for the system to examine the output of all of your servlets and JSP pages, figure out which parts contain hyperlinks back to your site, and modify those URLs. You have to tell it which URLs to modify. This requirement means that you cannot have static HTML pages if you use URL rewriting for session tracking, or at least you cannot have static HTML pages that refer to your own site. This is a significant burden in many applications, but worth the price in a few.

There are two possible situations in which you might use URLs that refer to your own site.

The first one is where the URLs are embedded in the Web page that the servlet generates. These URLs should be passed through the encodeURL method of HttpServletResponse. The method determines if URL rewriting is currently in use and appends the session information only if necessary. The URL is returned unchanged otherwise.

The second situation in which you might use a URL that refers to your own site is in a sendRedirect call (i.e., placed into the Location response header). In this second situation, different rules determine whether session information needs to be attached, so you cannot use encodeURL. Fortunately, HttpServletResponse supplies an encodeRedirectURL method to handle that case. Here's an example:

Listing 7.3: Java Examples for Encoded URLS

```
#1 String originalURL = someRelativeOrAbsoluteURL; String encodedURL
= response.encodeURL(originalURL);
out.println("<A HREF=\"" + encodedURL + "\">...</A>");

#2
String originalURL = someURL;
String encodedURL = response.encodeRedirectURL(originalURL);
response.sendRedirect(encodedURL);
```

8 JSF

8.1 Motivation

Komponenten JSF erlaubt es, vollständige Webanwendungen in einfacher Form aus Komponenten aufzubauen. Darüber hinaus kann man Komponenten selbst erstellen und beliebig wiederverwenden. Die Komponenten einer Webpage bilden eine hierarchische Struktur, einen Komponentenbaum.

DatenTransfer JSF macht es sehr einfach möglich, Daten von der Applikation in die Benutzerschnittstelle (und wieder zurück) zu transferieren. Eine automatische (oder vom Entwickler gesteuerte) Konvertierung und Validierung ist dabei vorgesehen.

Zustandsspeicherung JSF ermöglicht die automatische Speicherung des Zustands der Applikation - sowohl des Komponentenbaums, also der einzelnen Komponenten als auch der Applikationsdaten. Diese Speicherung kann in einer Benutzersitzung am Server oder im HTML-Quelltext am Client erfolge

Event Handling Vom Benutzer am Client generierte Ereignisse (auch Event genannt) können am Server behandelt werden. Dazu werden Ereignisbehandlungsmethoden mit den einzelnen Komponenten verknüpft. Durch die strikte Trennung der Schichten der Applikation im Sinne der MVC-Architektur können die einzelnen an der Applikation beteiligten Personen (z.B. Webdesigner, Komponentenentwickler und Applikationsentwickler) unabhängig voneinander arbeiten.

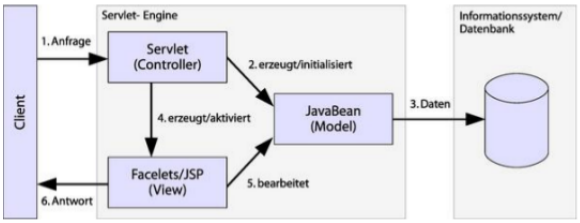

Abbildung 8.1: JSF Arch

Architektur

8.2 JSF Lebenszyklus

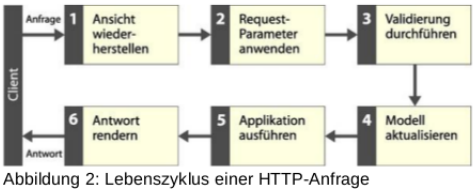

Abbildung 2: Lebenszyklus einer HTTP-Anfrage

Abbildung 8.2: JSF Lebenszyklus

Phase 1: Ansicht wiederherstellen (Restore View)

- Bei der allerersten Anfrage existiert dieser Komponentenbaum noch nicht und JSF baut ihn aus der Seitendeklaration neu auf
- JSP als View-Technologie zum Einsatz, leitet JSF die Anfrage an die hinter der Ansicht liegende JSP-Seite weiter. Diese wird abgearbeitet und bei jedem Antreffen eines neuen, noch nicht zu einer initialisierten Komponente gehörenden Tags wird eine neue Komponente erzeugt und mit den Attributwerten aus der JSP-Seite initialisiert.
- **Facelets** ist eine moderne View-Technologie, die mit JSF 2.0 eingeführt wurde. Facelets hat JSP abgelöst, da die älte" JSP-Technologie in Kombination mit JSF wegen der unterschiedlichen Konzepte zur Schwierigkeiten führte. Facelets verfolgt eine ganz ähnliche Strategie, baut den Baum beim Parsen aber auf der Basis eines XHTML-Dokuments auf.

- **Unterschied Facelets JSP** Caching von alten Werte. Komponentenbaum hat Cache sowie auch Validatoren/Konverter.

Phase 2: Request-Parameter anwenden (Apply Request Values) • Komponentenbaum gearbeitet.

- Benutzer Werte von Formular werden an einzelnen Komponenten zugewiesen.

  geschieht, indem am Wurzelknoten die Methode processDecodes() aufgerufen wird - der Wurzelknoten ruft dann die gleiche Methode auf seinen Kindknoten und diese wiederum auf ihren Kindknoten rekursiv auf.

  ucht sich beim Abarbeiten der Methode jede Komponente (oder genauer gesagt der der Komponente zugeordnete Renderer) aus der HTTP-Anfrage, und zwar aus den Parametern, HTTP-Kopfzeilen (Header) und Cookies , die Werte heraus, die diese Komponente betreffen, und speichert sie als "übermittelter"Wert (Submitted-Value).

  Decoding Process genannt.

  Ist allerdings noch nicht der Wert, der dann später tatsächlich ins Modell geschrieben wird - Validierung kommt noch.

Phase 3: Konvertierung und Validierung durchführen (Process Validations) • konvertiert und validiert.

- Die Konvertierung erfolgt vom zeichenkettenbasierten Submitted-Value auf die für das dahinterliegende Datenmodell notwendige Darstellung. Aus der Zeichenkette "01.01.2012"wird dann zum Beispiel eine Instanz der Klasse "java.util.Date".
- Abschluss des Konvertierungsvorgangs wird der Wert der Komponente validiert; das erledigen sogenannte Validatoren

  Es gibt im JSF-Standard bereits einige vorgefertigte Validatoren (z.B. ein LengthValidator oder ein DoubleRangeValidator

Zwischenbemerkung Setzen des konvertierten und validierten Wertes: Allerdings noch nicht in die Managed-Beans, sondern vorerst nur in die Eigenschaft value innerhalb der Komponente. Fehlschlagen? → Fehlermeldungen generiert und die aktuelle Seite wird inklusive Fehlermeldungen als Antwort gerendert. Das bedeutet, dass alle folgenden Phasen ausser der Äntwort rendernPhase übersprungen werden. **Von Schritt 3 direkt zu Schritt 6 Springen**

Phase 4 : Model Aktualisieren Unter der Voraussetzung, dass die abgesendeten Werte richtig konvertiert, validiert und lokal gespeichert werden konnten, werden diese Werte jetzt auf die von den einzelnen Komponenten referenzierten Eigenschaften der Geschäftsdaten übertragen.

Üblicherweise wird dafür die value -Eigenschaft der Komponente mit einer Value-Expression an eine Eigenschaft der Geschäftslogik gebunden. Ein Beispiel hierfür ist der bereits bekannte Ausdruck "#customer.firstName", mit dem die Eigenschaft firstName"der Managed-Bean customer referenziert wird.

5)Applikation Ausführen usführen der Businesslogik. Diese speichert beispielsweise geänderte Geschäftsdaten, liest Geschäftsdaten auf der Basis geänderter Filterkriterien neu aus oder kommuniziert mit anderen Systemen. edenfalls bestimmen die Methoden der Businesslogik durch ihren Rückgabewert, wohin die Reise in der Anwendung gehen wird: **View Navigation auf basis von Ergebnissen aus Business Logik.**

6) Render Response - Antwort Rendern Komponentenbaum gerendert und die Ausgabe wird als Antwort der JSF-Anfrage zum Client geschickt.

- Der Komponentenbaum wird aus der Seitendeklaration aufgebaut. Mit JSP passiert das durch einen Forward auf die JSP-Datei, in Facelets beim Parsen der XHTML-Datei.
- Der in Schritt 1 erstellte Komponentenbaum wird durch einen Aufruf der Methode ëncodeAlläuf dem Wurzelknoten gerendert. *Rendern der Werte der einzelnen Komponenten wieder die bereits erwähnten Konverter ins Spiel: Der Renderer holt den Wert der Komponente, ruft die Methode "getAsString()äuf dem Konverter auf und rendert das in eine Zeichenkette verwandelte Objekt zurück zum Client.*

8.3 JSF im Überblick

- Facelets wurde in JSF 2.0 in den Standard aufgenommen und ist die Sprache der Wahl für die Deklaration von Seiten.
- Kompositkomponenten ermöglichen ab JSF 2.0 das Erstellen von eigenen Komponenten - ohne eine Zeile Java-Code zu schreiben.

- Die Integration von Bean-Validation erlaubt eine metadatenbasierte Validierung.
- Ajax wurde in den Standard integriert. Eine Reihe neuer Annotationen macht die Konfiguration von JSF-Anwendungen so einfach wie nie zuvor.
- JSF 2.0 standardisiert die Verwaltung von Ressourcen wie Skripte oder Stylesheets System-Events bieten die Möglichkeit, auf spezielle Ereignisse im Lebenszylus zu reagieren.
- JSF 2.0 vereinfacht das Navigieren mit impliziter Navigation.
- Mit dem View-Scope gibt es einen neuen Gültigkeitsbereich für Managed-Beans.

8.4 Beispiel Applikation

8.4.1 Servlet Config

Listing 8.1: JSF Web Config

```
  <servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class> #1
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.jsf</url-pattern>  #2
</servlet-mapping>
```

- Implementation von Java Servlet Spezifikation/Interface
- alle .jsf URLs gehen über diese Servlet.

8.4.2 Was sind ManagedBeans

Das Model ist ein normales JavaBean oder POJO. Es wird als Managed Bean eingeführt, das durch den JSF Container verwaltet wird. Das SSimpleBeanïst ein solches Model:

Listing 8.2: Managed Bean Beispiel

```
 @ManagedBean(name = "mySimpleBean") #1
@RequestScoped #2
public class SimpleBean {
public String getMessage() {#3
return "Hello from JSF";
}
}
```

1. Wie in Spring - name des Beans, sonnst wird Klassenname verwendet.
2. Klasse wird nur für 1 Request am leben gehalten, neu Instanzierung pro Request.
3. Get methode, kann in View mit jstl gehohlt werden.

View

Die View ist eine XHTML-Page, da Facelets als View-Technologie eingesetzt werden soll. Das folgemde File ßimple.xhtml"wird im Ordner "webapp/pagesërstellt.

Listing 8.3: Grundlegende View Beispiel

```
  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"> #1
<h:outputText value="#{mySimpleBean.message}" />#2
</html>
```

1. Registrierung von komponenten unter h (html) von jsf - jetzt kann ich h komponenten wie output in mein View nutzen ohne Exceptions :)
2. Ausgeben von message.

8.5 Design Patterns in JSF

FrontController Pattern: Wie bei SpringMVC stellt auch JSF den FrontController. Hier ist es das Servlet FFacesServlet". Einen PageController kennt JSF aber nicht.

Composite View Pattern: Dieses Pattern wird mit Facelets umgesetzt. Facelets stellt ein XML- Schema mit verschiedenen Elementen zur Verfügung, um ein Composite aufzubauen.

Template View Pattern: Wie bei SpringMVC wird in JSF mit der View-Technologie Facelets ebenfalls auf XHTML gesetzt. Der statische Inhalt wird demnach mit einem wohlgeformten HTML realisiert, während die dynamischen Werte über die JSF-Expression-Language in das Template einfliessen.

8.6 Composite View in JSF

Listing 8.4: CompositeView Beispiel JSF

```
...
<h:body>
<div id="wrapper">
<ui:insert name="header-left">
<ui:include src="logo.xhtml"/>
</ui:insert>
<ui:insert name="header-center" />
<ui:insert name="header-right" />
<div id="main">
<ui:insert name="content" />
<ui:insert name="footer">
<ui:include src="footer.xhtml" />
</ui:insert>
</div>
</div>
</h:body>
...
```

Erklärung: ui:insert - Dynamischer Bereich mit Namen, ui:include (dynamisches Subview) : Statischer Bereich mit Inhalt in Datei (subview).

Im Gegensatz zu Tiles, wo die Komposition in einem externen XML-Konfigurationsfile aufgebaut wird, definiert man mit Facelets dies direkt in einer XHTML-Page.

Listing 8.5: JSF Dynamisches Subview

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="../templates/layout.xhtml">#1
<ui:define name="header-center">#2
<ui:include src="header.xhtml" />#3
</ui:define>
```

1. Welche Template soll benutzt werden?
2. Definition von dynamischem Bereich.
3. Inhalt von externem Datei wird hier angeziegt.

8.7 Vertiefung Template View in JSF

Die Unified Expression Language wurde entwickelt, weil sowohl JSP als auch JSF ihre eigene Expression Language (EL) definiert hatten. Für JSF war deswegen eine eigene Expression Language zusätzlich zur JSP-EL notwendig, weil

die Ausdrücke der JSP-EL sofort evaluiert werden, die Ausdrücke der JSF-EL aber später - zum Beispiel in einer späteren Phase des Lebenslaufs einer HTTP-Anfrage - evaluiert werden können. Ein Beispiel: Würde der Entwickler in einer JSP/JSF-Seite statt

```
 value="#{myBean.myValue}"
den Ausdruck
value="${myBean.myValue}"
```

verwenden, so würde der zweite Ausdruck bereits beim Erstellen der Komponente aufgelöst werden, also beim Parsen der Seite und JSF hätte keine Möglichkeit mehr, diesen Ausdruck in der Modell- Aktualisierungsphase auszuwerten, um einen Setter für den Wert zu erhalten. In der Unified-EL wurde auf dieses Bedürfnis Rücksicht genommen, und tatsächlich kann sowohl mit #ausdruck als auch mit $ausdruck ein Ausdruck definiert werden, der von der Anwendung (oder vom JSP-Tag) zum richtigen Zeitpunkt evaluiert wird. Die beiden Notationen sind also in der neuen Unified-EL äquivalent.

JSF stellt dem Programmierer eine umfangreiche Auswahl an vordefinierten Komponenten zur Verfügung, die viel Funktionalität zum Erstellen von Benutzeroberflächen mitbringen. Für die meisten Anwendungsfälle beherrschen diese Standardkomponenten das gewünschte Verhalten, 29.10.2013 2/6angefangen von einfachen Eingabeschaltflächen über Textfelder bis hin zur Darstellung von Daten in Tabellenform:

8.8 JSF View JSTL Komponenten

html	h	http://java.sun.com/jsf/html	HTML Custom-Tag-Library	Die Tags für die Standard-JSF-Komponenten und ihre Darstellung als HTML-Ausgabe befinden sich in dieser Library
core	f	http://java.sun.com/jsf/core	Core Tag-Library Diese Tags sind für die Basis-funktionalität unabhängig von speziellen Renderern zuständig	
facelets	ui	http://java.sun.com/jsf/facelets	Facelets Tag-Library Die Tags für die neue View-Handler-Technologie von JSF	

Variante 1: Facelets mit component-aliasing" Ein wichtiges Merkmal von Facelets ist das sogenannte component-aliasing". Damit ist es möglich, statt der Tags für die UI-Komponenten normale HTML-Tags, wie zum Beispiel ¡input¿ zu nutzen. Die Verbindung zu der UI-Komponente wird über das alias-Attribut "jsfcïm Tag hergestellt. Die entsprechende Komponente wird beim Kompilieren der Seite durch Facelets eingefügt. Der Vorteil des component-aliasingïst, dass Webdesigner die Seite mit herkömmlichen HTML-Editoren bearbeiten können, da die normalen HTML-Tags benutzt werden. Die zusätzlichen Attribute für JSF stören dabei nicht. Hier ein Beispiel:

Listing 8.6: Component Aliasing

```
   <table>
2  <thead>
   <th> Order No</th>
   <th> Product Name</th>
   </thead>
   <tbody>
7  <tr jsfc="ui:repeat"
   value="#{order.orderList}"
   var="o">
   <td>#{o.orderNo}</td>
   <td>#{o.productName}</td>
12 </tr>
   </tbody>
   </table>
```

1. Table tag
2. Table row mit jsfc repeat tag damit man es für alle in Orderlist wird wiederhohlt.
3. Enumrable Objekt.

4. Variable für Objekt im Enumrable Objekt.
5. Hohlen von Variablen.

Listing 8.7: Klassische JSF

Variante 2: Klassische JSF Komponenten

```
<h:dataTable value="#{order.orderList}"
var="o"
>
<h:column>
<!-- column header -->
<f:facet name="header">Order No</f:facet>
<!-- row record -->
#{o.orderNo}
</h:column>
<h:column>
<f:facet name="header">Product Name</f:facet>
#{o.productName}
</h:column>
</h:dataTable>
```

1. Datatable - Tabellentag wird generiert mit referenz auf Enumerable.
2. Variable für Items in Enumerable.
3. Generator für td tags.
4. 1 tr generiert als Kopfzeile.
5. Tr's wierderhohlt für jede Item.

tr = Table row.

8.9 Navigation links mittels JSF

Ein wichtiger Teil jeder JSF-Applikation ist die Definition der Navigation zwischen den einzelnen Ansichten. Damit der Benutzer im Browser überhaupt von einer Ansicht der Anwendung zu einer anderen wechseln kann, muss die Seite eine Steuerkomponente enthalten. Darunter versteht man eine Komponente, die das Absenden der aktuellen Seite an den Server startet und somit die Abarbeitung des Lebenszyklus am Server anstösst. Von diesen Steuerkomponenten gibt es in JSF zwei: "h:commandButtonünd "h:commandLink". Sie werden als Schaltfläche beziehungsweise Link in HTML ausgegeben:

```
 <h:commandButton action="#{questionnaireBean.persist}" value="Save"/>
<h:commandLink action="#{questionnaireBean.create}" value="Create">
```

Der ausschlaggebende Faktor für den Einsatz der Navigation ist das Attribut äction"der Steuerkomponenten. Darüber wird am Ende der Invoke-Application-Phase entschieden, welche Ansicht von JSF gerendert und zum Benutzer zurückgesendet wird. Ab JSF 2.0 kann direkt die View-ID einer Ansicht im action -Attribut angegeben oder von der Action- Methode zurückgegeben werden - wir bezeichnen das als implizite Navigation. Will man z.B. nach dem persist()ßur Hauptseite wechseln, wird dies in der Action-Methode des QuestionnaireBean zu:

```
 public String persist() {#1
...
return "/pages/main?faces-redirect=true";#2
}
```

1. Der Action.Methode, die vom "h:commandButtonïm obigen Beispiel aufgerufen wird. Die Action-Methode muss einen String Return-Werte zurückgeben. Hier wird die neue View festgelegt.
2. Hier führt die implizite Navigation zur Page "/pages/main.xhtml". Durch die Einführung der impliziten Navigation muss auch das Auslösen eines Redirect möglich sein, um das Double-Submit- Problem verhindern zu können. Mit dem Parameter faces-redirect=true"wird dieses Redirect festgelegt.