Durch den Einsatz dieser API soll ein späterer Austausch der XML-Datenbank erleichtert werden. Im Rahmen der Entwicklung des DigiPub-DMS stellte sich jedoch heraus, dass es Unterschiede bei den Implementationen dieser API gibt. So liefert zum Beispiel die Methode `getContentAsDOM` der Klasse `XMLResource` aus der XML:DB-API ein `org.w3c.dom.Node`-Objekt zurück, wobei es sich im Fall von Tamino um eine `org.w3c.dom.Document`-Instanz handelt, im Fall von eXist (Version 0.81) um eine `org.w3c.dom.Element`-Instanz.

Die XML:DB-API ist allgemein für XML-Datenbanken ausgelegt, Tamino hingegen enthält ein eigenes Konzept zur Organisation der Datenbanken. Die XML:DB-API spezifiziert, dass eine Datenbank Collections enthält, welche Dokumente enthalten, eine Schema- oder Dokumententypangabe ist hier nicht erforderlich. Des Weiteren muss die ID eines Dokumentes in jeder Collection eindeutig sein. In Tamino hingegen beinhaltet eine Datenbank Collections, die Schemata mit Dokumententypangabe enthalten, welche wiederum Dokumente enthalten. Die ID eines Dokumentes muss innerhalb eines Schemas eindeutig sein, nicht jedoch innerhalb einer Collection (als ID des Dokumentes wird das Attribut ino:docname verwendet).

Weiterhin wird bei der Verwendung der XML:DB-API mit Tamino eine Namensgleichheit sowohl von Collection und Schemanamen als auch dem Namen des Dokumententyps empfohlen. Außerdem sollte jedes Dokument eine ID enthalten, dessen erster Teil (abgetrennt durch ein „ / “) aus dem Namen des Dokumententyps besteht[45]. Dies ist notwendig, da bei der Speicherung eines Dokumentes ein Vergleich des ersten Teils der Dokumenten-ID mit den Namen der Schemata in der Collection, in welcher das Dokument gespeichert wird, erfolgt. Anhand dieses Vergleichs wird das entsprechende Schema zur Speicherung gewählt. Bei XML-Dokumenten könnte für diesen Vorgang auch ein Vergleich des Root-Elementes aus dem Dokument mit dem Schemanamen erfolgen, bei Binärdateien hingegen nicht.

Dieses Konzept hat jedoch für die Verwendung einer anderen Datenbank einen Nebeneffekt: eXist beispielsweise interpretiert bei einer Dokumenten-ID, dessen erster Teil durch ein / abgetrennt ist, diesen als Collectionnamen. Es speichert das Dokument in dieser Collection und unter einer ID, die dem Teil nach dem / entspricht. Existiert die entsprechende Collection nicht, wird sie angelegt. Das Dokument ist anschließend nicht durch die vorab vergebene ID erreichbar.

Das DigiPub-DMS ist daher auf die Verwendung von Tamino als Datenbank ausgelegt, es wird aber dennoch die XML:DB-API genutzt. Während der Entwicklung des DigiPub-DMS wurden bereits einige Fehler in dieser noch relativ neuen Schnittstelle für den Tamino XML Server entdeckt und mit der Software AG besprochen.

Als Abfragesprache für den Zugriff auf die XML-Datenbank nutzt das DigiPub-DMS XPath (siehe „XPath“), da für diese eine entsprechende Implementation in der XML:DB-API enthalten ist. Das *World Wide Web Consortium* entwickelt neben XPath eine weitere Anfragesprache: XML Query. Diese hat zum jetzigen Zeitpunkt (08.01.2002) den Status eines Working Drafts, und eine entsprechende Implementation für die XML:DB-API steht noch nicht zur

[45] Information per Email von Trevor Ford (Software AG USA) am 24.10.2002

Verfügung. Die Software AG bietet jedoch eine prototypische Implementation dieser Abfragesprache an, die den Namen *Quip* trägt und unter http://developer.softwareag.com/tamino/quip/ (Zugriff: 08.01.2003) erhältlich ist. XML Query basiert auf XPath und ist durch mehrere andere Sprachen wie SQL oder XML-QL[46] beeinflusst worden. Für Interessierte wird auf die XML Query Webseite des W3C [47] und [Klettke2002] (S. 266 ff.) verwiesen.

Auch bei der Implementation von XPath bieten verschiedene Datenbanken eigene Erweiterungen an. Zur Realisierung einer Volltextsuche existiert beispielsweise in eXist der Operator &=, durch dessen Nutzung innerhalb eines XML-Baumes nach dem Vorkommen einzelner Worte gesucht werden kann. Tamino bietet einen ähnlichen Mechanismus an, jedoch dient dort der Operator ~= diesem Zweck. Zur Realisierung der Volltextsuche im DigiPub-DMS wurde dieser Operator genutzt. Dies ist ein weiterer Grund, weshalb das System explizit auf die Nutzung des Tamino XML Servers abgestimmt ist.

4.3.2. Relationale Datenspeicherung

Neben XML-Daten fallen im DigiPub-DMS weitere Daten an, die eine Speicherung in einem Datenbanksystem erfordern wie beispielsweise Benutzerdaten. Obwohl es ohne weiteres möglich ist, diese Daten ebenfalls in einer XML-Datenbank zu speichern, wurde im Rahmen des DPS-Projektes entschieden, diese in einer relationalen Datenbank zu speichern. Relationale Datenbanken wurden genau für diese Art Daten konzipiert.

> In der Regel werden Daten in traditionellen Datenbanken, wie relationalen, objektorientierten oder hierarchischen Datenbanken gespeichert. [...] Dokumente werden in einer nativen XML Datenbank (einer Datenbank, die speziell für die Speicherung von XML entwickelt wurde) oder einem Content-Management-System (einer Anwendung, welche zum Handhaben von Dokumenten entwickelt wurde und auf einer nativen XML Datenbank aufsetzt) gespeichert.
>
> — [Bourret Online], Kapitel 4.3, eigene Übersetzung

Auch Entwickler von XML Datenbanken unterscheiden zwischen der Speicherung von einfachen Daten und XML-Dokumenten und raten zur Nutzung einer Datenbank, die für die entsprechenden Daten konzipiert ist. Hier kann zum Beispiel die Software AG genannt werden[48], welche neben dem Tamino XML Server auch die relationale Datenbank *Adabas*[49] anbietet.

Für das DigiPub-System wurde die Nutzung einer *MySQL* Datenbank gewählt, da diese frei verfügbar, sehr einfach zu bedienen und sehr performant ist.

[46] XML-QL ist eine SQL ähnliche Anfragesprache für XML, vgl. [Klettke2002], S 295 ff.

[47] Erreichbar unter http://www.w3.org/XML/Query (Zugriff: 08.01.2003)

[48] Vgl. [Steffen2001]

[49] Die Adabas Produktseite ist erreichbar unter http://www.softwareag.com/adabas/ (Zugriff: 13.01.2003).

> Wir haben IBM's DB2 7.2 mit FixPack 5, Microsoft Corp.'s SQL Server 2000 Enterprise Edition mit Service Pack 2, MySQL AB's MySQL 4.0.1 Max, Oracle Corp.'s Oracle9i Enterprise Edition 9.0.1.1.1 und Sybase Inc.'s ASE (Adaptive Server Enterprise) 12.5.0.1. Overall getestet, Oracle9i und MySQL hatten die beste Performance und Skalierbarkeit. [..] Die Oracle und MySQL Treiber hatten die beste Kombination von kompletter JDBC Unterstützung und Stabilität. [...] SQL Server und MySQL waren am leichtesten einzustellen, und Oracle9i am schwersten [...]
>
> — [eWeeks2002], eigene Übersetzung

Als Schnittstelle für den Zugriff auf die relationale Datenbank wurde JDBC verwendet, eine von *Sun Microsystems* entwickelte Datenbankschnittstelle. Diese Schnittstelle abstrahiert (genau wie die XML:DB-API, siehe „Datenbankzugriff“) den Zugriff auf die Datenbank, wodurch ein Austausch der Datenbank gegen eine andere, welche ebenfalls einen JDBC Treiber anbietet, leicht und ohne Eingreifen in den Quelltext der Applikation möglich ist. Für eine große Anzahl relationaler Datenbanksysteme existieren Implementationen eines JDBC Treibers[50].

4.4. Jakarta Struts

Das DigiPub-DMS ist unter Zuhilfenahme des *Jakarta Struts Framework* (siehe Struts 1.1) der Apache Software Foundation realisiert. Dieses Framework bietet u.a. folgende Features (Auszug, vgl. [Brown2001] S. 259):

- Erzwingt die Trennung von Anwendungs- und Präsentationslogik
- Bietet eine XML basierte Konfiguration zur Abbildung von logischen Aktionen auf konkrete Anwendungskomponenten und dementsprechende Präsentationskomponenten
- Unterstützung zur Validierung von Formulardaten und entsprechender Fehlerbehandlung (`struts-config.xml`)
- Unterstützung zum Einsatz von Internationalisierung

4.4.1. JSP Model 2 Architektur

Die strikte Trennung von Anwendungs- und Präsentationslogik wird durch die Implementation einer JSP Model 2 Architektur erreicht, wobei es sich um eine Variation des

[50] Die JDBC Treiber-Datenbank von Sun Microsystems enthält zum jetzigen Zeitpunkt (09.01.2003) 175 Treiber, siehe http://industry.java.sun.com/products/jdbc/drivers

„Model-View-Controller“ (MVC) Designmusters handelt. Sun Microsystems empfiehlt den Einsatz des MVC-Designmusters für interaktive Anwendungen[51]. Abbildung 4.4. *JSP Model 2 Architektur* veranschaulicht die Funktionsweise dieser Architektur.

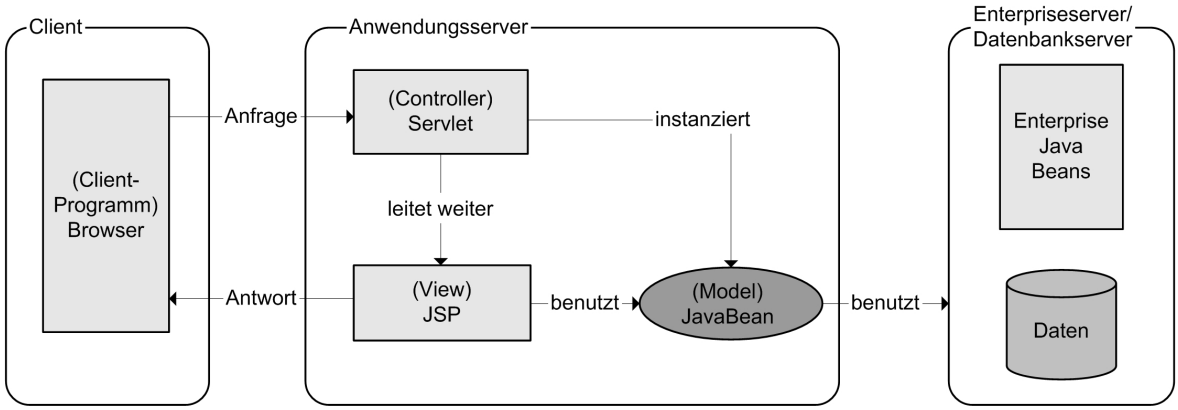

Abbildung 4.4. JSP Model 2 Architektur

Quelle für Abbildung 4.4. *JSP Model 2 Architektur*: Eigener Entwurf in Anlehnung an [Govind1999 Online].

Anfragen an die Anwendung werden von einem zentralen Servlet in der Rolle des *Controllers* angenommen. Dieser Controller entscheidet entsprechend der Anfrage über die auszuführende Aktion und leitet die zur Anfrage gehörigen Daten an ein Java Bean weiter. Das Java Bean sammelt die für die Antwort benötigten Daten und liefert diese zurück an den Controller. Dieser sendet die Daten an die der Aktion zugeordnete Java Server Page (JSP) weiter, welche mithilfe dieser Daten eine HTML-Ergebnisseite erstellt und diese an den Client zurückliefert.

Das *Jakarta Struts Framework* bietet verschiedene Komponenten zur Umsetzung dieses Konzeptes an[52]:

- Die Klasse `ActionServlet`: diese dient als zentraler Controller gemäß dem MVC-Entwurfsmuster. In der Anwendung gibt es nur eine Instanz dieser Klasse.

- Die Klasse `ActionForm`: diese dient als Oberklasse für anwendungsspezifische Java Bean Komponenten, welche zur Datenhaltung von Formulardaten dienen und daher Attribute gemäß eines entsprechenden HTML-Formulars enthalten. Das `ActionServlet` kann empfangene Daten eines Formulars an ein entsprechendes Bean weiterleiten, die Konfiguration hierfür erfolgt in einer zentralen Konfigurationsdatei (`struts-config.xml`). Zur Überprüfung von Formulardaten dienen `validate`-Methoden, welche von der anwendungsspezifischen Klasse überschrieben werden können. Diese werden vom `ActionServlet` nach dem Senden der Daten aufgerufen und liefern ein Objekt der Klasse `ActionErrors` zurück. Anhand dieses Objektes wird entschieden, ob die gesen-

[51] Vgl. [J2EE], Abschnitt 4.4

[52] Vgl. [Brown2001] S. 780/781

deten Daten valide sind und die angefragte Aktion ausgeführt wird oder nicht. Sollten die Daten nicht valide sein, wird die anfragende JSP-Seite wieder an den Client zurückgeliefert.

- Die Klasse `Action`: diese dient als Oberklasse für anwendungsspezifische Klassen, in denen die Anwendungslogik implementiert ist. Wird eine Aktion vom Client angefragt, ruft das `ActionServlet` die `execute`-Methode der entsprechenden Klasse auf. Die Abbildung von logischen Aktionen auf die entsprechende Klasse wird mithilfe der Klasse `ActionMapping` durchgeführt. Auch diese Abbildung wird in der zentralen Konfigurationsdatei `struts-config.xml` definiert.

- Tag-Bibliotheken: diese enthalten verschiedene vorgefertigte Komponenten, um in JSP-Dateien einen einfachen Zugriff auf Java Bean Komponenten zu erhalten. Hierdurch wird der in den JSP-Dateien nötige Java Quellcode verringert. Das Jakarta Struts Framework bietet hierfür u.a. folgende Tag-Bibliotheken:

  - Bean Tags: Komponenten für den Zugriff und die Erstellung von Java Beans.
  - HTML Tags: Komponenten zur Erstellung von HTML-Elementen.
  - Logic Tags: Komponenten zur Abbildung von logischen Elementen (wie beispielsweise eine Prüfung auf Gleichheit zweier Bean-Attribute).
  - Template Tags: Komponenten zur Erstellung von Schablonen für dynamische JSP-Seiten.

Unter Berücksichtigung der von Struts zur Verfügung gestellten Komponenten lässt sich der Ablauf auf dem Anwendungsserver (siehe Abbildung 4.4. *JSP Model 2 Architektur*) in erweiterter Form darstellen. Diese Darstellung ist zu sehen in Abbildung 4.5. *Struts Architektur*.

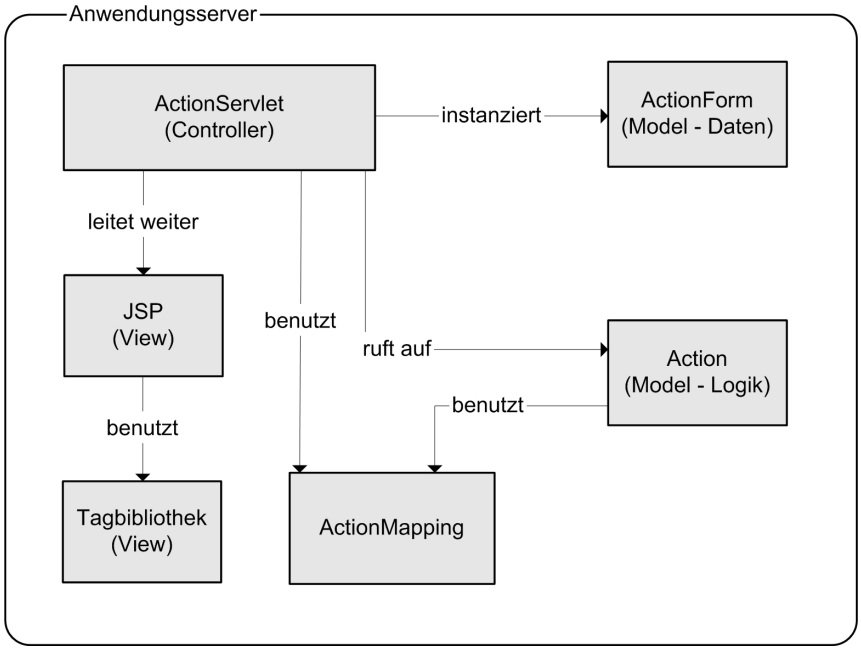

Abbildung 4.5. Struts Architektur

Quelle für Abbildung 4.5. *Struts Architektur*: Eigener Entwurf in Anlehnung an [Brown2001], S. 781.

4.4.2. Internationalisierung

Neben den in Abbildung 4.5. *Struts Architektur* abgebildeten und bereits erklärten Komponenten nutzt das DigiPub-DMS den von Java und Jakarta Struts Framework zur Verfügung gestellten Mechanismus zur Internationalisierung[53]. Bei der Technik der Internationalisierung existiert nur eine Implementation des Programms, welche für alle Sprachen genutzt wird. Textinhalte werden mit einem Schlüssel versehen und in einer für die entsprechende Sprache vorgesehenen Datei abgelegt. Eingefügt werden diese Textinhalte dynamisch anhand des Schlüssels und abhängig von der jeweils gewählten Sprache.

Java bietet eine gute Unterstützung für den Einsatz von Internationalisierung, u.a. durch die Verwendung des Unicode-Zeichensatzes, durch den alle existierenden geschriebenen Sprachen abgebildet werden können. Verschiedenen Regionen (geographisch, politisch und kulturell) werden durch Instanzen der Klasse `Locale` dargestellt. Anwendungsspezifische Sprachbausteine können in `Properties`-Dateien gespeichert werden, welche einen einheitlichen Dateinamen erhalten, an den ein Kürzel für die jeweilige Sprache (ISO-639 Format) und das jeweilige Land (ISO-3166 Format) angehängt wird. `Properties`-Dateien enthalten Einträge in der Form *schluessel=wert*. Ein Beispiel für eine solche Properties-Datei ist in Beispiel 4.4. *Textbausteine_en_US.properties* dargestellt, welche englischen Text („en“ im Dateinamen) für die USA („US“ im Dateinamen) enthält. In einem Java-Programm wird dann anhand eines definierten `Locale`-Objektes auf den entsprechenden Text zugegriffen.

[53] Siehe http://jakarta.apache.org/struts/userGuide/building_view.html#i18n (Zugriff 17.12.2002)

Beispiel 4.4. Textbausteine_en_US.properties

```
beispiel.text=This is an example
```

Für den Einsatz von Internationalisierung in einer Webanwendung muss ein `Locale`-Objekt für die Sprache des Benutzers referenziert werden. Diese wird in einer HTTP-Anfrage durch einen Browser vom Benutzer mitgesendet und mit Hilfe des Struts Framework unter dem Attribut `LOCALE_KEY` der Klasse `Action` in der Session des Nutzers gespeichert. Das Struts Framework ermöglicht auch einen einfachen Zugriff auf die entsprechenden Sprachbausteine durch Bean Tags (siehe Struts Tagbibliotheken). Hierzu muss eine entsprechende `Properties`-Datei der Applikation bekannt gemacht werden (Konfiguration in `struts-config.xml`). Ein Beispiel für ein entsprechendes Bean Tag zeigt Beispiel 4.5. *Internationalisierung mit Struts*. Hier wird der dem Schlüssel „beispiel.text“ zugeordnete Text in der Sprache ausgegeben, die dem Benutzer zugewiesen wurde.

Beispiel 4.5. Internationalisierung mit Struts

```
<%@page contentType="text/html"%>
<%@taglib uri="/struts-bean" prefix="bean"%>      ❶
<html>
  <body>
    <p>Text:
      <bean:message key="beispiel.text"/>         ❷
    </p>
  </body>
</html>
```

❶ JSP-Direktive zum Einbinden der *struts-bean* Tagbibliothek.

❷ Referenz auf den einzufügenden Textbaustein.

Die Nutzung des Jakarta Struts Framework bringt deutliche Vorteile mit sich, besonders im Hinblick auf eine Erweiterung und Wiederverwendung: Durch die Trennung von Anwendungslogik und Präsentationsschicht herrscht eine klare Aufgabenteilung für Programmierer und Webdesigner. Auch die Modularität der einzelnen Komponenten trägt zur Vereinfachung einer späteren Erweiterung bei[54].

[54] Vgl. [iX 11/02]

Kapitel 5. Grundlegende Entscheidungen

In diesem Kapitel werden die grundlegenden Entscheidungen zum Design und der Implementation des DigiPub-DMS erläutert. Hierbei wird bereits auf die Realisierung einzelner Komponenten eingegangen.

5.1. MVC-Design

Grundlegend für das Design des DigiPub-DMS ist der Einsatz des Jakarta Struts Frameworks und die Verwendung der von diesem Framework angebotenen MVC Implementation (vgl. Abschnitt 4.4). Die Präsentationsschicht (view) besteht aus Java Server Pages (JSP), und einzelne auszuführende Aktionen sind als `Action`-Klassen implementiert. Für Formulardaten, die eine Überprüfung auf Korrektheit verlangen, existieren entsprechende `ActionForm` Implementationen. Zur einfachen Untergliederung ist die Anwendung in Pakete aufgeteilt (packages, siehe Abschnitt A.2). Für den Zugriff auf Modell-Klassen existieren eigene Klassen, um auch hier die Anwendungslogik strikt von den Daten zu trennen (z.B. dient für die Klasse `User` die Klasse `UserAdministration` als Zugriffsschnittstelle).

5.2. Präsentationsschicht

Um eine möglichst flexible Präsentationsschicht zu erhalten, wird die Tagbibliothek `struts-template` eingesetzt. Durch eine Schablone erlaubt diese die Unterteilung der Präsentationsschicht in einzelne Bereiche, denen verschiedene Inhalte (JSP-Dateien) zugewiesen werden. Für das DigiPub-DMS wurde eine Unterteilung in fünf Bereiche vorgenommen (vgl. Abbildung 5.1. *DigiPub Präsentationsschablone*).

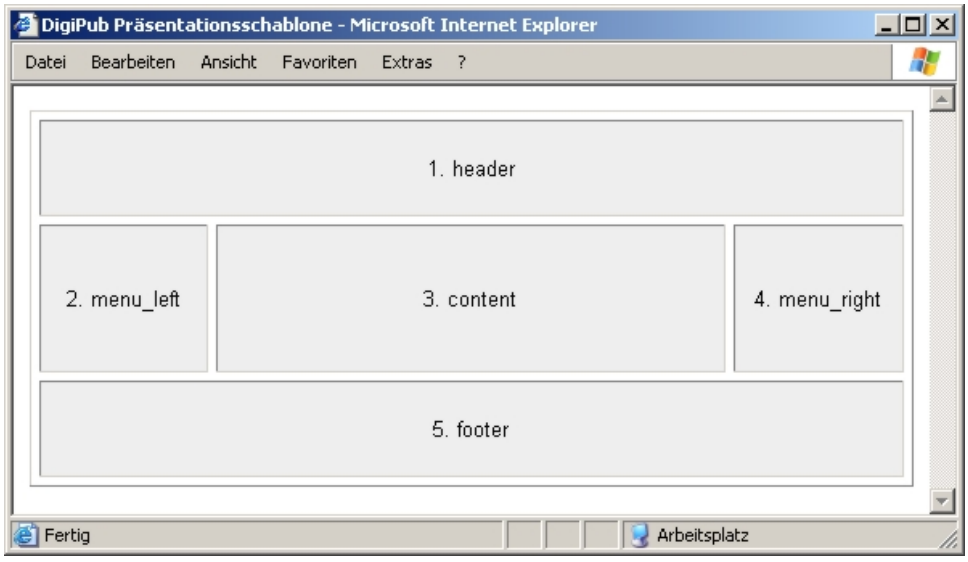

Abbildung 5.1. DigiPub Präsentationsschablone

Die in Abbildung 5.1. *DigiPub Präsentationsschablone* abgebildeten Bereiche dienen im DigiPub-System folgendem Zweck:

1. header: Der obere Fensterbereich. Dieser Teil ist gedacht für statische Inhalte oberhalb der eigentlichen Inhaltsausgabe, welche auf allen auszuliefernden Seiten enthalten sind.

2. menu_left: Der linke Teil des mittleren Fensterbereiches. In diesem sind das Administrationsmenu und das Menu für den Zugriff auf Dokumente untergebracht.

3. menu_right: Der rechte Teil des mittleren Fensterbereiches. In diesem ist das Benutzermenu untergebracht.

4. content: Der mittlere Teil des mittleren Fensterbereiches. Dieser Teil enthält den angeforderten Inhalt oder die Ausgabe einer Aktion.

5. footer: Der untere Fensterbereich. Dieser Teil ist gedacht für statische Inhalte unterhalb der eigentlichen Inhaltsausgabe, welche auf allen auszuliefernden Seiten enthalten sind.

Zur Steuerung der Zuweisung von JSP-Seiten zu den einzelnen Bereichen der Schablone dient ein eigener Controller (`PageController`), als Schablone dient die JSP-Datei `digipubTemplate.jsp`.

5.3. Internationalisierung

Um einen Einsatz im internationalen Bereich zu ermöglichen (vgl. L07), muss es einfach sein, die Inhalte in verschiedenen Sprachen zu repräsentieren. Hierzu wurden verschiedene Ansätze für dynamische und statische Präsentationsseiten gewählt:

1. Internationalisierung (vgl. Abschnitt 4.4): Diese Technik wurde für Seiten mit dynamischen Inhalten gewählt, um eine redundante Programmierung der Anwendungslogik zu vermeiden.

2. Lokalisierung: Bei dieser Technik wird für jede unterstützte Sprache eine eigene Version der sprachabhängigen Programmteile erstellt. Dies hat in der Regel den Nachteil, dass bei einer möglichen Änderung mehrere Dateien gepflegt werden müssen (vgl. [Deitsch2001], S. 6). Gewählt wurde diese Technik bei Seiten mit statischen Inhalten. Neben jeweils einer globalen JSP-Seite, in der sprachunabhängige Ausgaben getätigt werden können, existieren für diese Seiten jeweils eine HTML-Datei mit lokalisierten Inhalten, welche dynamisch eingebunden werden. Diese Inhalte sind als HTML-Seiten leichter pflegbar, da sie unter anderem durch einen entsprechenden HTML-Editor bearbeitet werden können.

Im Rahmen des DPS-Projektes wurden die Sprachen Deutsch und Englisch implementiert, eine Erweiterung für andere Sprachen ist durch das Erstellen dementsprechender HTML- bzw. `Properties`-Dateien möglich. Das DigiPub-DMS enthält vier verschiedene Gruppen von Sprachdateien, welche in Tabelle 5.1. *Properties-Dateien zur Internationalisierung* erläutert sind. Ein Beispiel für die Einbindung lokalisierter HTML-Dateien ist in Beispiel 5.1. *Einbindung lokalisierter HTML-Dateien* zu finden. Für die Sprache Deutsch wird hier zum Beispiel die HTML-Datei `about.de.html` eingebunden.

Tabelle 5.1. Properties-Dateien zur Internationalisierung

Properties-Datei	**Nutzungszweck**
ApplicationResources	Sprachbausteine in JSP-Dateien.
ExceptionResources	Sprachbausteine für Fehlermeldungen.
MailResources	Sprachbausteine für Text zum Versenden einer Email.
XSLResources	Sprachbausteine für XSL-Transformationen.

Beispiel 5.1. Einbindung lokalisierter HTML-Dateien

```
<%
   Locale currentLocale = (Locale)
                          session.getAttribute(Action.LOCALE_KEY);
   String includePage = "about." +
                        currentLocale.getLanguage() + ".html";
%>
<jsp:include page="<%= includePage %>"/>
```

5.4. Zugriff auf XML-Daten

Zum Zugriff auf XML-Daten verwendet das DigiPub-DMS die *Java API for XML Processing* (JAXP). Diese Schnittstelle ermöglicht es, auf XML-Daten zuzugreifen und diese zu transformieren, ohne einen speziellen XML-Parser oder XSLT-Prozessor vorauszusetzen. Es existieren zwei standardisierte Schnittstellen für diesen Zugriff, die von JAXP unterstützt werden: Die Simple API for XML (SAX) und das Document Object Model (DOM). Der im DigiPub-DMS eingesetzte XML-Parser (siehe Xerces-J 2.2.1) sowie der XSLT-Prozessor (siehe Xalan-Java 2.4.1) implementieren beide Standards (jeweils SAX 2.0 und DOM Level 2).

Die Verarbeitung von XML auf Basis von SAX erfolgt ereignisgesteuert. Das XML-Dokument wird durch einen SAX-Parser sequentiell abgearbeitet, wobei die Anwendung direkt beim Auftreten eines Elementes hierüber informiert wird. Dazu muss die Anwendung die Schnittstelle `DocumentHandler` implementieren und für die zu verarbeitenden Komponenten

entsprechende Methoden zur Verfügung stellen. Diese werden aufgerufen, sobald der Parser eine entsprechende Komponente im XML-Dokument erreicht. Beispielsweise wird die Methode `startElement` aufgerufen, sobald ein Element erreicht wird. Dieser Methode werden dann der Elementname und dessen Attribute als Parameter übergeben, sodass eine Verarbeitung durch die Anwendung erfolgen kann. SAX bietet eine schnelle und einfache Möglichkeit für den Zugriff auf XML, hat allerdings auch den Nachteil, dass kein kontextabhängiger Zugriff auf einzelne Elemente ermöglicht wird, da die aufgerufenen Methoden hierzu keine Informationen erhalten und SAX zustandslos ist[55]. Auch eine Manipulation der XML-Daten ist nicht möglich.

Wie bereits in „Datenbankauswahl“ erwähnt, ermöglicht das DOM einen kontextabhängigen Zugriff auf einzelne Elemente eines XML-Dokumentes. Mit Hilfe dieser API können komfortable Zugriffe auf einzelne Komponenten sowie deren Änderung erfolgen, da auf die gesamte Struktur des XML-Dokumentes zugegriffen werden kann. Dies bietet einen Vorteil gegenüber SAX. Nachteilig hingegen ist der erheblich höhere Speicheraufwand von DOM, der für diese Strukturinformationen nötig ist (vgl. „Datenbankauswahl“, dort ist auch eine mögliche Alternative, das PDOM, beschrieben). Somit eignet sich SAX besonders für sehr große Dokumente, die keine Manipulation erfordern und eine einfache Struktur besitzen. Das DOM hingegen bietet erhebliche Vorteile bei aufwendigen Umformoperationen, insbesondere dann, wenn diese kontextabhängig durchgeführt werden sollen[55]. Die verschiedenen Arbeitsweisen von SAX und DOM sind zur Veranschaulichung in Abbildung 5.2. *Vergleich von DOM und SAX* illustriert.

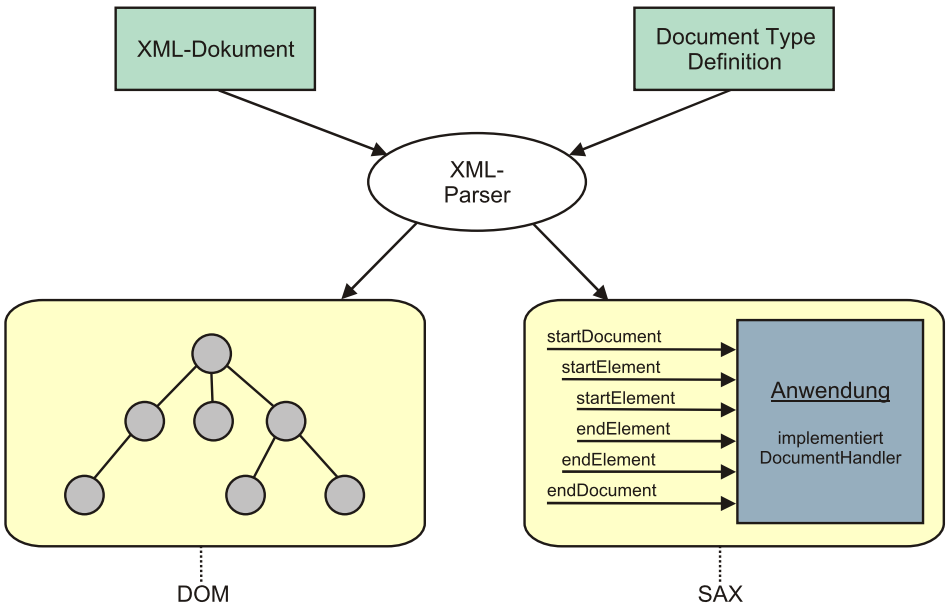

Abbildung 5.2. Vergleich von DOM und SAX

Quelle für Abbildung 5.2. *Vergleich von DOM und SAX*: Eigener Entwurf in Anlehnung an [Böndgen2000].

[55] Vgl. [Klettke2002], S. 52 ff.

Das DigiPub-DMS benutzt für die Verarbeitung von XML-Daten das *Document Object Model*, da die Hauptaufgabe hierbei in Umformoperationen liegt. Einfache Transformationen, bei denen die Elemente, auf welche zugegriffen wird, bekannt sind, werden durch die Nutzung der Java DOM-API durchgeführt. Ein Beispiel hierfür ist das Einfügen einer Annotation in ein DocBook-Dokument (siehe Abschnitt 6.4). Umfangreiche Transformationen hingegen, wie beispielsweise die Anzeige eines Suchergebnisses aus dem Inhalt einer XML-Datenbankabfrage (siehe Abschnitt 6.3.4), werden durch die Nutzung eines XSLT-Stylesheets von einem XSLT-Prozessor durchgeführt. Diese Stylesheets können bei einer Erweiterung der DocBook-DTD ebenfalls leicht erweitert werden, ohne dass die Anwendung neu erstellt werden muss. Das Stylesheet, welches für die Darstellung des Suchergebnisses verwendet wird, kann somit auch leicht vom Layout her angepasst werden, ohne in den Java Quelltext eingreifen zu müssen.

Neben den beiden standardisierten Schnittstellen SAX und DOM existiert eine weitere oft genutzte API für den Zugriff auf XML: *JDOM* [56].

> JDOM ist sowohl zentriert auf Java als auch für Java optimiert. Es verhält sich wie Java, benutzt Java Collections, ist eine vollkommen naturgemäße API für gegenwärtige Java-Entwickler und bietet einen einfachen Einstieg zur Nutzung von XML.
>
> — http://www.jdom.org/mission/index.html, Zugriff: 10.01.2003, eigene Übersetzung

JDOM wird ebenfalls von dem im DigiPub-System eingesetzten Tamino XML Server unterstützt. Dies gilt jedoch nicht für die XML:DB-API, weshalb eine Benutzung dieser Schnittstelle nicht in Betracht gezogen wurde.

5.5. Sicherheitskonzept

Das Sicherheitskonzept für den Zugriff auf das DigiPub-System basiert auf verschiedenen Benutzerrollen mit unterschiedlichen Rechten. Eine Auflistung der Rollen ist in Tabelle 5.2. *Nutzerrollen* dargestellt. Ein Nutzer, der sich nicht am System angemeldet hat (vgl. Abschnitt 6.2), erhält die Rolle Gast. Jeder sich in dem System registrierende Nutzer erhält standardmäßig die Rolle Benutzer, welche ihm bei jeder erfolgreicher Anmeldung am System wieder zugewiesen wird. Die Rollen System-Administrator und Inhalts-Administrator werden nicht von dem System vergeben; die entsprechenden Einträge für die Zuweisung dieser Rollen sind separat in der Tabelle *user* der Datenbank *digipub* zu ändern.

[56] Informationen zu JDOM sind erhältlich auf der offiziellen Website des JDOM-Projects (siehe http://www.jdom.org, Zugriff: 10.01.2003).

Tabelle 5.2. Nutzerrollen

Rolle	Systemeinstellungen vornehmen	Anlegen neuer Mandanten	Veröffentlichen von ISBN-Dokumenten	Einstellen von Dokumenten	Abrufen von Dokumenten
System-Administrator	Ja	Ja	Ja	Ja	Ja
Inhalts-Administrator	Nein	Nein	Ja	Ja	Ja
Benutzer	Nein	Nein	Nein	Abhängig von Mandanteneinstellungen (vgl. Abschnitt 6.1)	Ja
Gast	Nein	Nein	Nein	Nein	Ja

Der System-Administrator erhält den vollen Zugriff auf das System, welcher sowohl für Systemeinstellungen und Wartungsarbeiten als auch für Testläufe im System nötig ist. Die Rolle des System-Administrators beinhaltet die Verantwortung für das komplette System. Der Inhalts-Administrator erhält den vollen Zugriff auf alle Dokumente und ist verantwortlich für die Veröffentlichung von ISBN-Dokumenten. Die Veröffentlichung von Dokumenten mit ISBN muss von einem Verantwortlichen ausgeführt werden, der sich mit den Grundsätzen des Verlagsrechtes auskennt, da nicht jedes Dokument eine ISBN erhalten kann[57].

Das DigiPub-DMS startet bei einem Zugriff auf das System eine Sitzung, in der unter anderem festgehalten wird, ob der entsprechende Nutzer am System angemeldet ist. Ist dies der Fall, so wird weiterhin festgehalten, um welchen Nutzer es sich handelt. Hierzu wird ein `HttpSession`-Objekt genutzt, welches auf dem Server verwaltet wird. Da die Kommunikation mit dem DigiPub-DMS über das zustandslose HTTP-Protokoll stattfindet, muss beim Nutzer vermerkt werden, welcher Sitzung er zugeordnet wird. Diesbezüglich bietet das Session-Konzept von Java neben dem Einsatz von *Cookies*[58] auch *URL-Rewriting*. Durch die Verwendung von URL-Rewriting wird eine eindeutige ID in der URI übergeben, wodurch eine clientseitige Aktivierung von Cookies nicht zwingend erforderlich ist. Die Sitzungssteuerung vom DigiPub-DMS funktioniert auch ohne diese Aktivierung.

[57] Vgl. „Merkblatt zur Handhabung der Internationalen Standard-Buchnummer (ISBN) in der Bundesrepublik Deutschland“, erreichbar unter http://www.german-isbn.de/isbn_merkblatt.html, Zugriff: 12.01.2003.

[58] Cookies sind Textdateien, die vom Browser direkt beim Nutzer gespeichert werden und anwendungsspezifische Informationen enthalten können.

Der Zugriff auf einzelne Aktionen im System hängt neben den globalen Nutzerrollen von mandantenabhängigen Einstellungen ab (vgl. Abschnitt 6.1) und wird daher in den einzelnen Abschnitten dargestellt, in denen diese Aktionen behandelt werden.

Der Zugriff auf Daten in der MySQL-Datenbank und dem Tamino XML Server ist durch deren Authentifizierungsmechanismen geschützt und kann nur über entsprechende Anmeldungsdaten erfolgen. Dieser Zugriff ist lediglich dem DigiPub-DMS und dem System-Administrator vorbehalten.

5.6. Fehlerbehandlung

Mögliche auftretende Fehler wie zum Beispiel Fehler bei Datenbankverbindungen werden über das Java *ExceptionHandling* verarbeitet. Innerhalb der Anwendung werden Fehlermeldungen so lange weitergereicht, bis eine Entscheidung zur Ausgabe einer Meldung für den Benutzer getroffen werden kann. Dieser enthält gemäß dem Internationalisierungsmechanismus (vgl. Abschnitt 5.3) eine Fehlermeldung in der von ihm gewählten Sprache, welche abhängig von seiner Aktion ist. Um Fehler zu analysieren, kann das DigiPub-System in einem entsprechenden Modus gestartet werden. Hierzu muss in der zentralen Konfigurationsdatei der Wert für *debugMode* auf *debug* gesetzt werden. In diesem Fall wird für den Nutzer neben der Fehlermeldung der gesamte Verlauf des Fehlers ausgegeben.

Um die Fehlermeldung direkt an den Nutzer in seiner gewählten Sprache weiterleiten und Fehlermeldungen des DigiPub-DMS eindeutig identifizieren zu können, wurde die Klasse `DigiPubException` erstellt, von der alle weiteren Exceptions des DigiPub-DMS erben. Sie enthält einen Konstruktor, in dem ein `Locale`-Objekt zu übergeben ist, um die Sprache der Fehlermeldung festzulegen. Ein Auszug aus dieser Klasse ist in Beispiel 5.2. *DigiPubException* dargestellt. Aufgrund der Übersichtlichkeit sind nur ein Konstruktor und eine Methode dargestellt, zudem wurde ein Großteil der JavaDoc-Kommentare entfernt. Für eine genaue Beschreibung wird auf die JavaDoc-Dokumentation und den Quelltext verwiesen. Zur Formatierung von Fehlermeldungen im debug-Modus dient die Klasse `ExceptionFormat`.

Beispiel 5.2. DigiPubException

```
public class DigiPubException extends java.lang.Exception
{
    /** Path to ResourceBundle for DigiPub ExceptionMessages */
    public static final String BUNDLEPATH =
                        "de.fhnon.digipub.resources.ExceptionResources";

    /** Array for message parameters */
    private String[] params;

    public DigiPubException(String messageKey, String[] params,
                            Throwable throwable)
    {
        super(messageKey);
        this.params = params;
    }
```

```
    public String getLocalizedMessage(Locale locale)
    {
       ResourceBundle bundle =
              ResourceBundle.getBundle(BUNDLEPATH, locale);

       String localizedMessage = super.getMessage();

       try
       {
           localizedMessage = bundle.getString(super.getMessage());
       }
       /**
        * If no message was found, an exception is thrown. This will be
        * ignored, in this case the message key is used.
        */
       catch (Exception e) { }

       return MessageFormat.format(localizedMessage, this.params);
   }
}
```

Zur Protokollierung von Systeminformationen wird der Standard Logging-Mechanismus von Java verwendet, zur zentralen Verwaltung im DigiPub-System dient die Klasse `LoggerInitiator`[59]. Im DPS-Projekt wurde entschieden, alle Fehlermeldungen als schwerwiegend (Java Logging-Level *severe*) einzustufen. Neben Fehlermeldungen werden auch wichtige Meldungen des Systems protokolliert (Java Logging-Level *info*). Zur Protokollierung wird standardmäßig Englisch verwendet.

5.7. Datenbankdesign

In diesem Abschnitt werden die im DigiPub-DMS verwendeten Entitäten in der relationalen Datenbank sowie die einzelnen Collections in der XML-Datenbank dargestellt.

5.7.1. Relationale Datenbank

Dem Entity-Relationship-Diagramm in Abbildung 5.3. *Relationale Datenbank "digipub"* sind die Beziehungen einzelner Entitäten in der Datenbank „digipub“ zu entnehmen. Für diese Entitäten sind entsprechende Tabellen in der MySQL Datenbank des Systems angelegt. Die Tabelle *client* enthält Mandanten und deren nötigen Daten (vgl. Abschnitt 6.1). In der Tabelle *level* sind die einzelnen Nutzerrollen (vgl. Tabelle 5.2. *Nutzerrollen*) definiert und die Tabelle *user* dient zur Speicherung von Benutzerdaten für registrierte Nutzer (vgl. Abschnitt 6.2). In der Tabelle *document* werden Dokumentendaten abgelegt. Diese Daten beinhalten die nötigen Informationen zur preisabhängigen Auslieferung von Dokumenten und zur Generierung von Sicherheitsaspekten in den auszuliefernden PDF-Dokumenten, welche für die Erweiterungen von Stephan Wiesner erforderlich sind. Das Design dieser Entität wurde gemeinsam im DPS-Projekt besprochen.

[59] Die Klasse `LoggerInitiator` wurde im Rahmen des DPS-Projektes von Stephan Wiesner entwickelt.

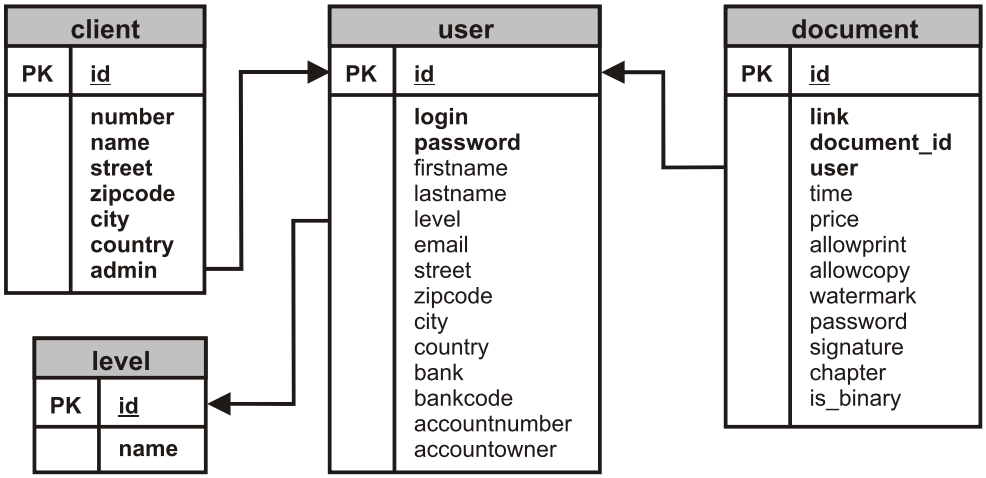

Abbildung 5.3. Relationale Datenbank "digipub"

Für den Zugriff über JDBC dient die Klasse `JDBCConnectionPool`, welche Verbindungen zur Datenbank verwaltet.

5.7.2. XML-Datenbank

In XML-Datenbanken wird die Speicherung unterschiedlicher Daten nicht durch Tabellen, sondern durch *Collections* organisiert. Der Tamino XML Server erlaubt die Definition von XML Schemata innerhalb von Collections, um die Struktur der zu speichernden XML-Dokumente vorzugeben. Da die DocBook DTD über 300 Elemente enthält[60], ist der Umfang für ein entsprechendes XML Schema zu groß, um an dieser Stelle übersichtlich innerhalb einer Grafik dargestellt werden zu können. Aus diesem Grund sind in Abbildung 5.4. *XML-Datenbank "digipub"* lediglich die verschiedenen Collections mit zugehörigem Dokumententyp und dem Namen des Tamino-Schemas abgebildet. Die Grafik ist angelehnt an das Entity-Relationship-Diagramm in Abbildung 5.3. *Relationale Datenbank "digipub"*. Allerdings werden anstatt von Entitätsnamen Collectionnamen verwendet und anstatt des Primärschlüssels der Dokumententyp gesondert abgetrennt.

[60] Vgl. [Walsh1999], S. ix

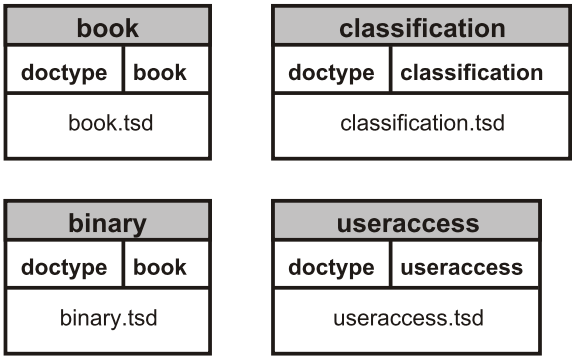

Abbildung 5.4. XML-Datenbank "digipub"

In der Collection *book* werden die XML-Daten der DocBook-Dokumente gespeichert, mögliche enthaltene Binärdaten wie beispielsweise Bilder werden für diese Dokumente in der Collection *binary* abgelegt (vgl. Abschnitt 6.3.1.1). Die Collection *classification* dient zur Speicherung von Kategorielisten für einzelne Mandanten, *useraccess* zur Speicherung von deren Zugriffslisten (vgl. Abschnitt 6.1). Zur Erzeugung der Datenstrukturen wurde neben der DocBook-DTD für jede Collection eine entsprechende DTD erstellt und anschließend mit Hilfe des Tamino Schema Editors zu einem XML-Schema konvertiert. Für die Collection *book* wurde die DocBook-DTD um einige systemspezifische Komponenten erweitert. Abbildung 5.5. *Tamino Schema Editor* zeigt exemplarisch die Erweiterung für Metainformationen, das Element digipubinfo, in dem Tamino Schema Editor. Zusätzlich wurden dem Element title die Attribute chapter_id, section_id und parent_id hinzugefügt (vgl. „Dokumentenupdate“).

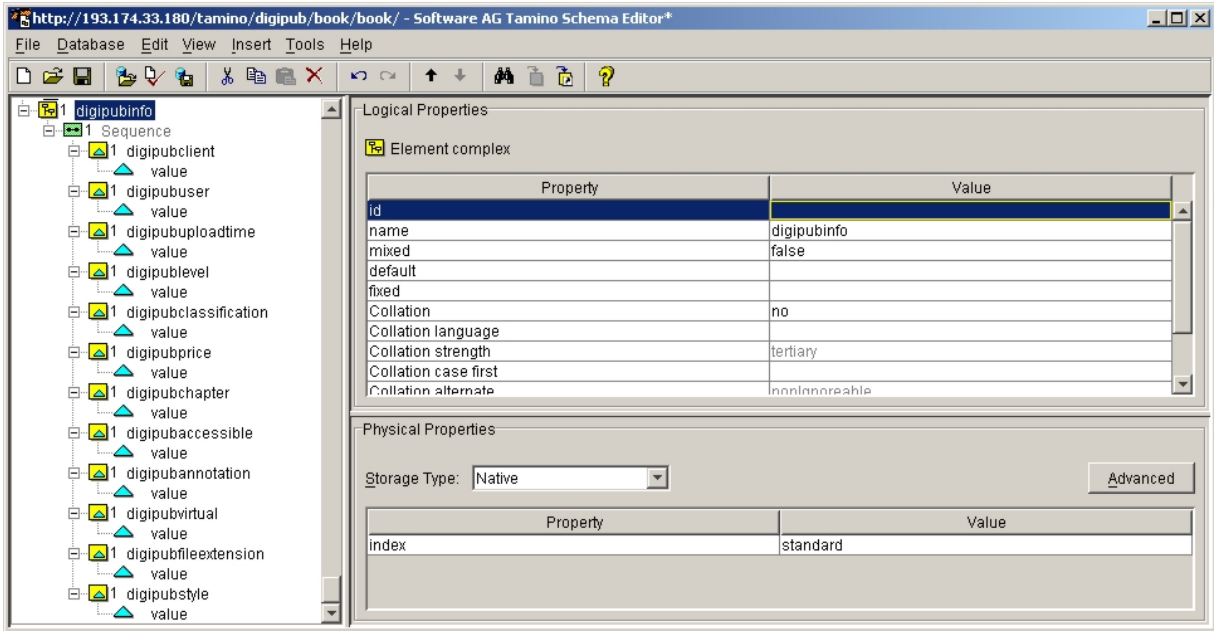

Abbildung 5.5. Tamino Schema Editor

Um eine organisierte Trennung der einzelnen Dokumentenarten zu erhalten, wurde für jeden Dokumententyp eine eigene Collection angelegt. Hierdurch kann eine Verletzung der Eindeutigkeit von Dokumenten-ID's ausgeschlossen werden und eine eindeutige Zuweisung von Dokumenten zu ihrem Dokumententyp erfolgen, da die Angabe hierzu in der XML:DB-API nicht vorgesehen ist (vgl. Abschnitt 4.3.1).

Für den zentralen Zugriff auf die Tamino-Datenbank dient die Klasse `XMLDBHandler`. Diese bietet verschiedene Methoden für die nötigen Datenbankanweisungen, welche bereits speziell auf XML- bzw. Binärdaten ausgelegt sind.

5.8. Konfiguration

Zur Konfiguration der Anwendung dient primär die zentrale Properties-Datei `digipub.conf`. In ihr sind alle relevanten Daten gespeichert, die systemabhängig sind oder eine mögliche Änderung erfahren sollen (wie zum Beispiel Passwörter). Der Zugriff auf diese Konfigurationsdatei erfolgt über eine zentrale Controller-Klasse, `DigiPubConfig`, welche als *Singleton* implementiert und somit nur eine Instanz in der Anwendung vorhanden ist.

Zur Verwaltung verschiedener XSL-Stylesheets, MIME-Typen und Standardkategorien für einzelne Mandanten existiert jeweils eine XML-Datei, wobei die Pfade zu diesen Dateien ebenfalls in der zentralen Konfigurationsdatei enthalten sind. Der Zugriff auf die Konfigurationsdatei für MIME-Typen erfolgt über die Klasse `MimeTypeAdministration`, welche für jeden enthaltenen Eintrag eine `MimeMapping`-Instanz erzeugt. Für den Zugriff auf die verschiedenen möglichen XSL-Stylesheets dient die Klasse `StyleAdministration`. Auch diese zwei Zugriffsklassen sind nach dem *Singleton*-Entwurfsmuster implementiert. Die erforderliche Struktur der XML-Dateien kann Abbildung 5.6. *FoStyle.dtd* und Abbildung 5.7. *MimeType.dtd* entnommen werden.

```
<!ELEMENT fo-style-types (style-type)+>
   <!ELEMENT style-type (path, description)>
      <!ELEMENT path (#PCDATA)>
      <!ELEMENT description (#PCDATA)>
```

Abbildung 5.6. FoStyle.dtd

```
<!ELEMENT mime-types (mime-mapping)+>
   <!ELEMENT mime-mapping (extension, mime-type)>
      <!ELEMENT extension (#PCDATA)>
      <!ELEMENT mime-type (#PCDATA)>
```

Abbildung 5.7. MimeType.dtd

Kapitel 6. Konzeption und Implementation

Nachdem im vorangegangenen Kapitel allgemeine Komponenten und Entscheidungen dargestellt wurden, beschreibt dieses Kapitel sowohl die Konzeption als auch die Implementation der Hauptfunktionalitäten des DigiPub-DMS. Viele Konzepte sind hier jedoch übergreifend, da beispielsweise bereits beim Einstellen von Dokumenten Aktionen vorgenommen werden, die für einen späteren Abruf nötig sind. Diese übergreifenden Entscheidungen sind entsprechend gekennzeichnet.

6.1. Mandantenmanagement

Um eine Möglichkeit zu schaffen, Dokumente unter verschiedenen Verantwortungsbereichen veröffentlichen zu können, enthält das DigiPub-DMS die Funktionalität zum Anlegen verschiedener Mandanten (vgl. L01). Ein Mandant bezeichnet in diesem Kontext einen Verantwortungsbereich mit zugehörigem Verantwortlichen (Mandant [lat.]: Der Auftraggeber). Ein Mandant kann verwendet werden, um einen Verlag in dem DigiPub-DMS einzurichten oder auch dafür, einfache Bereiche zu unterteilen wie beispielsweise Fachbereiche an einer Hochschule oder Abteilungen in einem Unternehmen. Zum Erhalt einer möglichst flexiblen Nutzung erfolgte diesbezüglich keine genaue Festlegung. Zur Erstellung eines Mandanten sind neben einem Namen und einer Nummer (für einen Verlag dienen diese Angaben als Verlagsname und -nummer) auf Grund der Impressumpflicht[61] ebenfalls die Angaben zu Adresse und einem Verantwortlichen erforderlich. Als Verantwortlicher dient ein Benutzer des Systems, dem die Rolle des Administrators für diesen Mandanten zugewiesen wird. Weitere Abhängigkeiten zwischen Mandant und Benutzer sind nicht im Datenbankmodell abgebildet (vgl. Abbildung 5.3. *Relationale Datenbank "digipub"*, Entität „client“).

Eine Zuweisung von Rechten für einzelne Benutzer zu einem Mandanten wird individuell unter Prüfung der Emailadresse vorgenommen (vgl. L01.1). Für diesen Zweck wird für jeden Mandanten eine Liste erstellt, in der anhand einzelner Emailadressen oder auch Email-Bereichen (wie beispielsweise „alle Emailadressen, die mit @fhnon.de enden“) die Zuteilung von Nutzern zu einer Benutzergruppe innerhalb des Mandanten erfolgt. Die verschiedenen Nutzergruppen sind in Tabelle 6.1. *Nutzergruppen eines Mandanten* dargestellt. Bis auf die Gruppe admin besteht in den Rechten der Gruppen kein Unterschied, beim Einstellen von Dokumenten in das System (vgl. Abschnitt 6.3.1) wird jedoch die Gruppe des einstellenden Nutzers mit festgehalten, um dem Dokument eine hierarchische Stufe zuordnen zu können. So kann beispielsweise im Bereich der FH NON ein Dokument eines Professors (Zuordnung zu *employee*) von einem Dokument eines Studenten (Zuordnung zu *dependant*) leicht unterschieden werden (Näheres siehe Abschnitt 6.3.4).

[61] Vgl. $6 Abschnitt 2 in „Gesetz über die Nutzung von Telediensten“, http://www.netlaw.de/gesetze/tdg.htm, Zugriff: 11.01.2003

Tabelle 6.1. Nutzergruppen eines Mandanten

Gruppe	Administrationsrechte	Rechte zum Einstellen von Dokumenten
admin	Ja	Ja
management	Nein	Ja
employee	Nein	Ja
dependant	Nein	Ja

Durch die mögliche Zuweisung von Email-Bereichen zu einzelnen Nutzergruppen muss nicht für jeden Nutzer eine neue Zuweisung erfolgen. Ein Nutzer, der sich neu registriert, kann bereits in einer Gruppe implizit erfasst sein. Zum einfachen Verständnis zeigt Abbildung 6.1. *Konfiguration der Nutzergruppen* ein Beispiel für eine Konfiguration. Durch die Verwendung des Zeichens * wird der entsprechende Teil der Emailadresse als irrelevant gekennzeichnet. In dieser beispielhaften Konfiguration werden alle Nutzer, deren Emailadressen mit *@fhnon.de* enden, dem Bereich employee zugewiesen (durch den Einsatz von Internationalisierung, vgl. Abschnitt 5.3, hier als „Angestellter“ bezeichnet). Dem Bereich management wurden in diesem Beispiel zwei eindeutige Emailadressen zugewiesen, wobei die Adresse *bonin@fhnon.de* ebenfalls in dem Bereich employee gültig ist. Eine Überprüfung der Zuweisung von Nutzern zu Bereichen erfolgt hierarchisch absteigend, weil auf diese Weise einem Nutzer die höchste für ihn zutreffende Nutzergruppe zugeordnet werden kann. Ein neu registrierter Benutzer, dessen Emailadresse auf *@fhnon.de* endet, wird nach dieser Konfiguration automatisch der Gruppe employee zugewiesen. Eine zusätzliche Anmeldung bei diesem Mandanten ist somit nicht nötig.

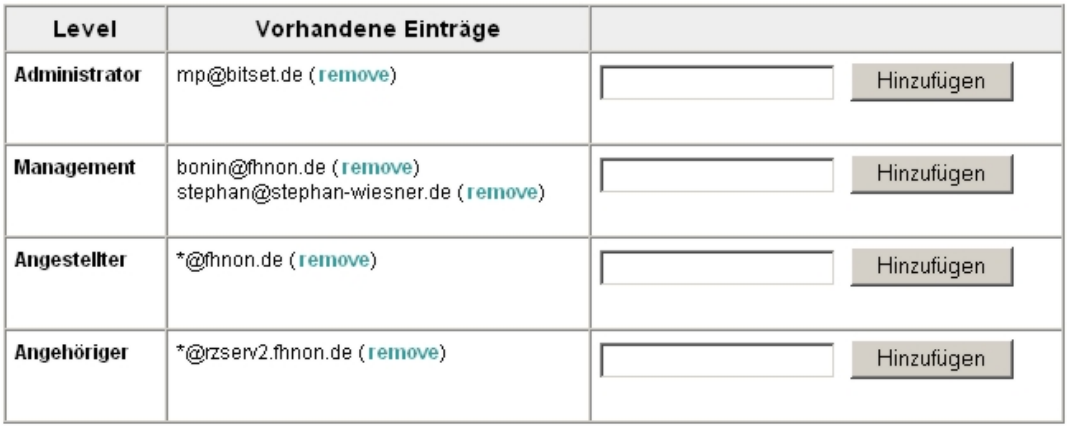

Abbildung 6.1. Konfiguration der Nutzergruppen

Aufgrund der Listenstruktur dieser Zuweisungen und um eine möglichst einfache Erweiterung zu unterstützen, werden diese Zugriffslisten im XML-Format erstellt und in dem Tamino XML Server gespeichert (Collection *useraccess*, vgl. Abbildung 5.4. *XML-Datenbank "digi-*