Entwicklung einer zensursicheren Medienplattform in Java unter Verwendung der Open Source Frameworks Spring und Tapestry

Masterarbeit

Hochschule Merseburg
Fachbereich Ingenieur- und Naturwissenschaften
Studiengang Informatik und Kommunikationssysteme

Autor Rudolf Niehoff

Erstprüfer
Prof. Dr. rer. pol. Uwe Schröter
Zweitprüfer
Prof. Dr. rer. nat. habil. Dr. phil. Michael Schenke

Merseburg den 06.11.2019

Inhaltsverzeichnis

Danksagung

Auch in dieser Arbeit möchte ich mich zu aller erst wieder bei allen bedanken, die mich bei der Umsetzung meiner Masterarbeit unterstützt haben.

Aus diesem Grund bedanke ich mich vor allem bei Herrn Prof. Dr. rer. pol. Uwe Schröter für die ausgezeichnete Betreuung während meiner Ausarbeitungen. Auch bei Herrn Prof. Dr. rer. nat. habil. Dr. phil. Michael Schenke möchte ich mich für die sofortige und freundliche Zusage als Zweitprüfer bedanken.

Beiden Professoren danke ich außerdem für die Unterstützung bei den verschiedenen Herausforderungen während meiner Studienzeit.

Auch meinen Eltern danke ich wieder für die Korrektur der Masterarbeit und der moralischen Unterstützung während des gesamten Studiums .

Und zu guter Letzt bedanke ich mich bei meiner ganzen Familie für die großartige Unterstützung zu jeder Zeit.

1. Einführung in das Projekt

1.1 Ausgangssituation und Zielsetzung

Ziel dieser Arbeit ist es, eine zensursichere Medienplattform zu implementieren, in welcher Beiträge redundant eingestellt werden können. Redundanz bedeutet hierbei, dass es mehrere Links zu anderen Quellen mit dem selben Inhalt geben kann. Der Inhalt entspricht einem Videobeitrag, welcher somit auf mehreren unterschiedlichen Videoportalen wie z.B. YouTube, Vimeo oder Dailymotion gespeichert sein kann. Sollte ein Video auf einem der Videoportale aus irgendeinem Grund gelöscht werden oder aus technischen Gründen nicht mehr erreichbar sein, dann greift die Redundanz, sodass einfach das gleiche Video von einem anderen Videoportal geladen und angezeigt wird.

Dem Nutzer der Plattform fällt dieser Vorgang nicht auf, da das Video ja somit verfügbar ist. Der Ersteller des Beitrags ist hingegen informiert und kann bei Bedarf weitere Links hinzufügen, um die Redundanz auch in Zukunft zu gewährleisten. Die Nutzung vorhandener externer Videoportale ermöglicht einen relativ kostengünstigen Betrieb der Plattform, da die Videos nicht selber gehostet werden müssen. Dennoch ist auch die Möglichkeit gegeben die Videoquellen selber zu hosten, da ein Link zu einer Videoquelle natürlich auch auf einen eigenen Server verweisen kann. Für diesen Fall sollte das Video allerdings für adaptives Streaming angepasst werden, worauf in dieser Arbeit jedoch nicht eingegangen wird.

1.2 Notwendigkeit von Zensursicherheit

Gründe für die Notwendigkeit einer Zensursicherheit sind unter anderem neuere Gesetze wie die DSGVO und damit geplante Upload-Filter. Zensur muss nicht zwangsläufig böswilliger Natur sein, sondern kann auch aus technisch unausgereifter Software heraus resultieren. Videoportale wie Youtube haben einfach nicht die Möglichkeit jedes einzelne Video von einem Mitarbeiter kontrollieren zu lassen und sperren Videoinhalte mittlerweile schon automatisch, wenn diese z.B. von Zuschauern gemeldet werden. Durch den Einsatz von Upload-Filtern wird es zudem noch häufiger zu unrechtmäßigen Löschungen kommen, da die Technik hinter solchen Upload-Filtern natürlich auch Fehler machen kann.

Das zusätzliche Verwenden von alternativen Videoportalen wird somit für den Videoersteller sozusagen unausweichlich. Allerdings weiß davon der potenzielle Zuschauer oftmals nichts, weshalb die Werke des Medienmachers aus diesem Grund dennoch nicht gesehen werden können. Durch die Verwaltung von Beiträgen auf einer zensursicheren Plattform, die lediglich Verweise zu den Videos enthält, kann dem Problem aus dem Weg gegangen werden. Sollte beispielsweise ein Satire-Video von YouTube fälschlicherweise als terroristischer Inhalt eingestuft und daraufhin gesperrt werden, kann der Ersteller des Beitrags das Video vorerst auf ein anderes Videoportal hochladen.

Durch die Verlinkungen auf der hier zu implementierenden Medienplattform, hat der Videoersteller somit die Möglichkeit alle seine Beiträge auf der selben Seite zu verwalten und zu präsentieren. Der Zuschauer findet durch dieses Prinzip zu jeder Zeit alle Beiträge des Medienmachers auf einer Plattform.

1.3 Begriffe: Medienplattform und Videoportal

Ein Videoportal oder auch Videoplattform ist eine Webseite, über welche die Benutzer Videos bereitstellen und wieder abrufen können. Meist werden die Videos, oder auch Webvideos genannt, vom Ersteller hochgeladen und können dann durch Streaming von einem Zuschauer direkt angesehen werden. Auch das gleichzeitige Hochladen vom Ersteller und Streamen beim Zuschauer, das so genannte Live-Streaming ist auf einigen Videoportalen möglich. Videos als Datei herunterladen ist in den meisten Fällen nur über technische Umwege, wie mit speziellen Browser-Plugins, zu realisieren. Bekannte Videoportale sind z.B. YouTube, Vimeo oder Dailymotion.

Eine Medienplattform ist im Grunde das selbe wie ein Videoportal, außer dass es sich bei Medien nicht nur um Videos handeln muss sondern auch Musik oder Bilder vorhanden sein können. Somit sind alle Videoportale gewissermaßen auch Medienplattformen. Da sich diese Abhandlung nur mit der Einbindung von Videoquellen beschäftigt, würde somit der Begriff Videoportal auch ausreichen. Da das grundlegende Konzept allerdings auf sämtliche Medien angewendet werden kann und um eine begriffliche Abgrenzung zu gewährleisten, wird die zu implementierende Plattform künftig als Medienplattform und andere Plattformen wie YouTube als Videoportal bezeichnet.

2. Beschreibung der verwendeten Frameworks

2.1 Spring Framework

Das Spring Framework ist ein meist für die Webentwicklung eingesetztes Java Framework, das mittels Dependency Injection (Einbringen von Abhängigkeiten) und aspektorientierter Programmierung einen leichteren und besser wartbaren Programmcode ermöglichen soll. Es ist modular aufgebaut, gilt als sehr leichtgewichtig und ist ein OpenSource-Projekt, welches die Komplexität von Java EE (Enterprise Edition) deutlich reduzieren soll. Dadurch, dass der Quellcode in Spring-Anwendungen kürzer wird, wird auch der Aufwand für spätere Wartungen und Änderungen einfacher. Das Programmiermodell basiert auf so genannten POJOs (Plain Old Java Objects), also leichtgewichtige Java-Klassen, welche nur die Java-Spezifikation als Grundlage umsetzen. Das heißt, dass solche Klassen keine vorspezifizierten Klassen erweitern oder vorspezifizierte Interfaces implementieren müssen. Interfaces dienen in Spring dazu, mittels Dependency Injection die Abhängigkeiten von dem eigentlichen Quellcode fernzuhalten, der dadurch stark reduziert wird und lediglich lose gekoppelte Systemkomponenten aufweist. Folgendes Beispiel soll das aufzeigen.

```java
// Interface für die Spring-Komponente
public interface ICalculator {

	double add(final double... summands);
}

// Spring-Komponente
@Component
public class Calculator implements ICalculator {

	@Override
	public double add(final double... summands) {

		double result = 0.0;

		for (final double sum : summands) {
			result += sum;
		}

		return result;
	}
}

// POJO: Plain Old Java Object
// Einbindung der Komponente über @Autowired
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class CalculatorTest {

	@Autowired
	private ICalculator calculator;

	@Test
	public void testAddDoubleArray() {

		final double result = calculator.add(1.0, 2.0);
		assertEquals(result, 3.0);
	}
}
```

Quelltext 1: Dependency Injection einer Spring-Komponente als Beispiel

Durch die komponentenübergreifende Strukturierung von Zusammenhängen wird auch die aspektorientierte Programmierung realisiert, was die Trennung des eigentlichen Programmablaufs von anderen Aspekten wie Validierung und Fehlerbehandlung erlaubt.

2.2 Apache Tapestry

Mit dem Framework Apache Tapestry können Webanwendungen in der Programmiersprache Java erstellt werden. Es ist ein OpenSource-Projekt, dass keinen eigenen Server bereitstellt sondern im Kontext eines bereits vorhandenen Servlet-Containers, wie dem Webserver Tomcat, läuft. Eine Tapestry-Anwendung besteht aus verschiedenen Seiten, welche wiederum aus mehrfach verwendbaren Komponenten bestehen kann. Sowohl Seiten als auch Komponenten sind aus einer XML-Vorlage (.tml → Tapestry Markup Language) und einer Javaklasse (.java) aufgebaut. Die XML-Vorlage, welche im Ganzen oder in Teilen den eigentlichen Seitenaufbau beschreibt, kann somit auch HTML-Tags enthalten und dadurch wie folgt aussehen.

```
1  <t:layout title="${myTitle}"
2          xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd"
3          xmlns:p="tapestry:parameter">
4
5          <p>${myText}</p>
6
7          <t:pagelink page="myPage">${myLink1}</t:pagelink>
8          <t:eventlink event="myEvent">${myLink2}</t:eventlink>
9  </t:layout>
```

Quelltext 2: Tapestry XML-Vorlage (.tml) als Beispiel

Die zugehörige Javaklasse dient zur Verarbeitung von Ereignissen und Formularen sowie für zusätzliche Aufgaben vor, während und nach dem Rendern der Seite. Die Java-Klasse kann wie folgt aufgebaut sein.

```
1  public class Index {
2
3          @Inject
4          private PageRenderLinkSource prls;
5
6          @Property
7          private String myTitle;
8
9          @Property
10         private String myLink1;
11
12         @Property
13         private String myLink2;
14
15         @SetupRender
16         void setup() {
17
18                 myTitle = "Titel der Seite";
19                 myLink1 = "Erster Link";
20                 myLink2 = "Zweiter Link";
21         }
22
23         public String getMyText() {
24
25                 return "Der Text, der auf der Seite angezeigt wird";
26         }
27
28         Object onMyEvent() {
29
30                 return prls.createPageRenderLink(MyPage.class);
31         }
32 }
```

Quelltext 3: Tapestry-Page Java-Klasse als Beispiel

Die aufzurufenden Methoden können entweder über spezielle Annotationen oder wie bei getMyText() über den Methodennamen bestimmt werden. Da das Programmiermodell der Java-Klassen wie im Spring Framework ebenfalls auf POJOs basiert, existieren hier auch Annotationen um Abhängigkeiten einzubinden. Das Äquivalent zum @Autowired in Spring nennt sich in Tapestry @Inject.

3. Aufbau der Datenbankstruktur

3.1 Allgemeiner Aufbau

Die Daten der Medienplattform untergliedern sich in zwei Hauptbestandteile. Das sind einerseits die Mediendaten, bestehend aus Studios, Serien, Beiträgen und Videoquellen sowie andererseits die Nutzerverwaltung bestehend aus Nutzerkontos, Nutzerdaten und Abonnements von Beiträgen eines Nutzers (Nutzerabonnements). Für alle Bestandteile wurden Datenmodelle mittels Java Persistence API (JPA) implementiert, die automatisch die zugehörigen Tabellen in der Datenbanksprache SQL erzeugt, um die Datenhaltung der Medienplattform zu gewährleisten. Im folgenden sind die Tabellen den zwei Hauptbestandteilen zugeordnet.

Mediendaten[1]	Feldbenennung	**Nutzerverwaltung**	Feldbenennung
Studio	STUDIO	Nutzerkonto	USER_ACCOUNT
Serie	ARTICLE_GROUP	Nutzerdaten	USER_DATA
Beitrag	ARTICLE	Nutzerabonnement	USER_SUBSCRIPTION
Videoquelle	ARTICLE_VIDEO		

Tabelle 1: Bezeichnungen der Datenbanktabellen der Medienplattform

Ein Studio besteht aus Serien und die Serien enthalten die Beiträge. Ein Beitrag kann nun mehrere Videoquellen enthalten, um die bereits erwähnte Redundanz zu gewährleisten. Dies entspricht in allen Fällen einer 1:n-Relation. Die Nutzerkonten sind mit den Nutzerdaten über eine 1:1-Relation verknüpft. Das Nutzerkonto enthält zudem noch eine 1:1-Relation zu einer Serie, um diese verwalten zu

1 Im weiteren Verlauf der Arbeit können weiterhin die deutschen Bezeichnungen auftreten. In Quelltexten und Diagrammen werden allerdings die Feldbenennungen auftauchen.

können. Zuletzt gibt es noch das Nutzerabonnement, wobei ein Eintrag in diese Tabelle ein Nutzerkonto zu einer Serie zuordnet und somit ein Abo darstellt. Das entspricht wieder zwei 1:n-Relationen. Nachfolgend sind die Tabellen mit den jeweiligen Relationen grafisch dargestellt.

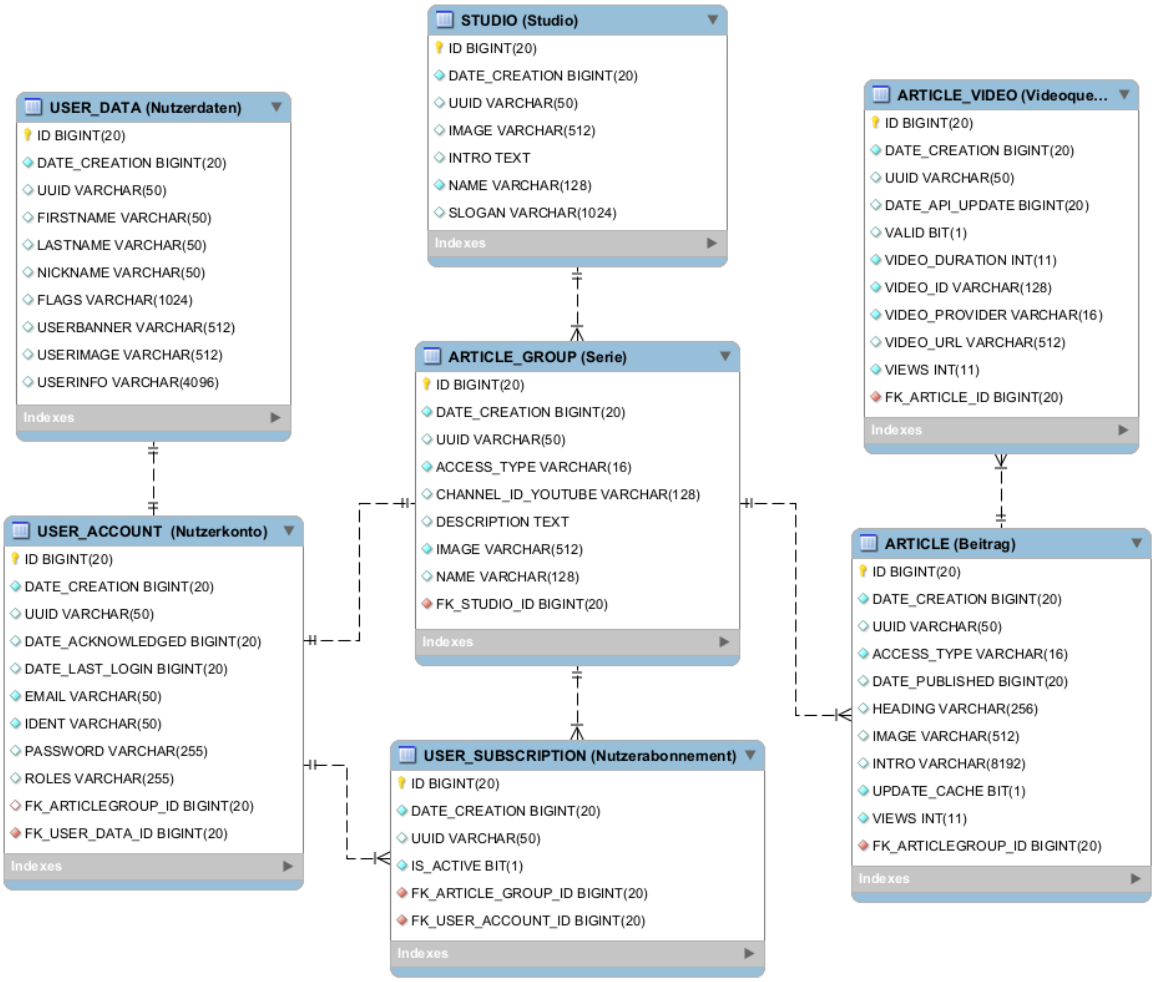

Abbildung 1: Datenbanktabellen der Medienplattform und ihre Relationen

In den folgenden Kapiteln werden die einzelnen Felder in den Tabellen erläutert. Ausgenommen davon sind folgende Felder, da diese in jeder Tabelle vorkommen und somit hier kurz beschrieben werden können:

- ID (Primärschlüssel)
- UUID (Zeichenkette mit einem „Universally Unique Identifier“)
- DATE_CREATION (Erstellungsdatum in Unixzeit)

Da die Medienplattform in Java programmiert wird, werden auch die Tabellen mittels Java Persistence API (JPA) erzeugt. Da sich eben genannte Felder in allen Tabellen gleichen, ist es hierfür nur folgerichtig, eine abstrakte Klasse zu implementieren, welche die Felder beinhaltet.

```
1  @Getter
2  @Setter
3  @MappedSuperclass
4  public abstract class PAbstractEntity {
5
6          @Id
7          @Column(name = "ID")
8          @GeneratedValue(strategy = GenerationType.IDENTITY)
9          private Long id;
10
11         @Column(name = "UUID", unique = true, nullable = true, length = 50)
12         protected String uuid;
13
14         @Column(name = "DATE_CREATION", nullable = false)
15         @NotNull
16         private Long dateCreation = System.currentTimeMillis();
17
18         public PAbstractEntity() {
19                 uuid = UUID.randomUUID().toString();
20         }
21 }
22
```

Quelltext 4: Abstrakte Klassen für generelle vorkommende Datenbankfelder

Sichtbar wird hierbei auch, dass sämtliche Felder bei der Erzeugung des Datenmodels auch gleich mit einem Wert belegt sind. Dadurch müssen diese später nicht mehr explizit gesetzt werden, bevor der entsprechende Datensatz in der Datenbank angelegt wird. Es handelt sich um rein technisch relevante Felder, die für den späteren Nutzer der Medienplattform keine Rolle spielen. Die ID ist der Primärschlüssel, welcher in jeder SQL-Datenbank vorhanden sein sollte und für die Verküpfung der Tabellen miteinander über den Fremdschlüssel anderer Tabellen entscheidend ist. Das UUID Feld beinhaltet einen „Universally Unique Identifier“, also eine Zeichenkette, die als zusätzliche Identifikation des Datenbankeintrags neben dem Primärschlüssel dient. DATE_CREATION ist das Erstelldatum des Datenbankeintrags und wird über die Systemzeit automatisch erzeugt.

3.2 Datenmodelle für die Mediendaten

Die Mediendaten werden, wie bereits erwähnt, von den vier Datenmodellen für Studios, Serien, Beiträge und Videoquellen gehalten. Durch die 1:n-Relationen haben die einzelnen Datenmodelle ein hierarchisches Verhältnis zueinander, sodass vorhandene Daten leicht durch eine Baumstruktur darstellbar sind, wie in folgendem Beispiel zu sehen ist.

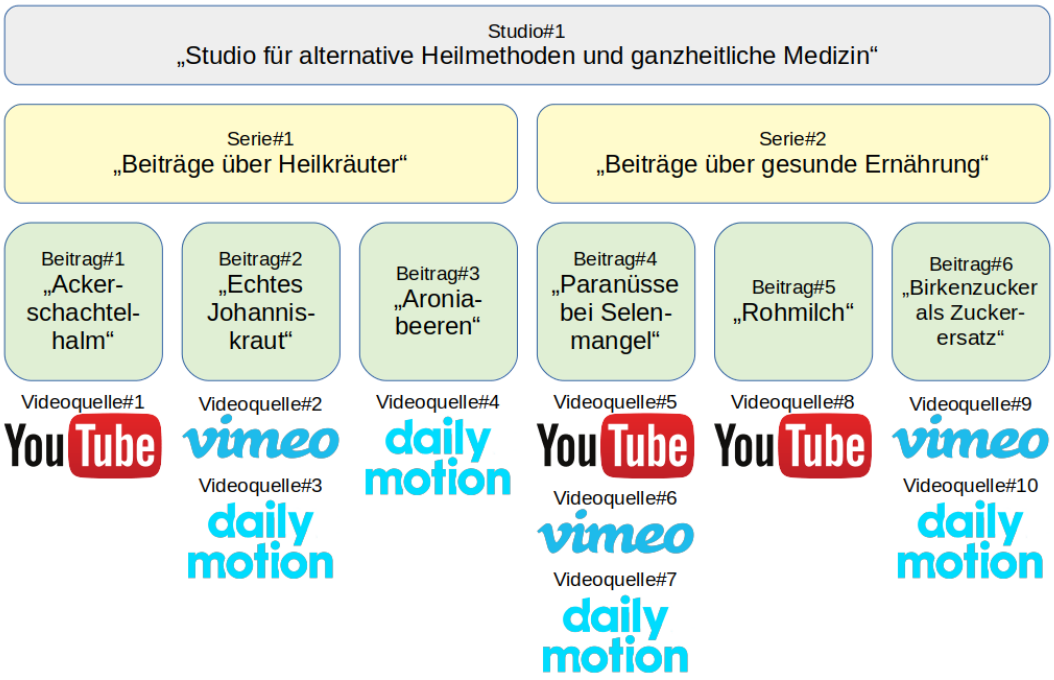

Abbildung 2: Visualisierung der Mediendaten als Baumstruktur

Nachfolgend soll nun der Aufbau aller Datenmodelle, beginnend mit dem Studio, beschrieben werden, um die Rangfolge in der Baumstruktur einzuhalten.

3.2.1 Studio

Das Studio stellt das Wurzelelement dar, über welches letztendlich alle anderen zugehörigen Knoten wie Serien, Beiträge und Videoquellen erreichbar werden. Jedes weitere Studio erzeugt somit einen eigenen Baum. Jede Serie muss einem Studio zugeordnet sein und kann nicht ohne Studio oder gleich mit zwei Studios verwaltet werden. Gleiches gilt für Beiträge und Videoquellen. Jeder Beitrag muss einer Serie zugeordnet werden und jede Videoquelle muss einem Beitrag zugeordnet werden. Studios können beliebig viele Serien enthalten, wie Serien beliebig viele Beiträge und Beiträge beliebig viele Videoquellen enthalten können.
Ein Studio enthält neben dem Namen (NAME) noch Felder für eine Einleitung des Studios (INTRO), ein Motto (SLOGAN), sowie die URL zu einem Bild (IMAGE), was vorher hochgeladen wurde. Damit sieht das Datenmodel für das Studio wie folgt aus.

```
1  @Getter
2  @Setter
3  @EqualsAndHashCode(callSuper = false)
4  @Entity
5  @Table(name = "STUDIO")
6  public class PStudio extends PAbstractEntity {
7
8          @Column(name = "NAME", unique = true, nullable = false, length = 128)
9          @NotNull
10         private String name;
11
12         @Lob
13         @Column(name = "INTRO", columnDefinition = "TEXT")
14         private String intro;
15
16         @Column(name = "SLOGAN", unique = false, nullable = true, length = 1024)
17         private String slogan;
18
19         @Column(name = "IMAGE", unique = false, nullable = true, length = 512)
20         private String image;
21
22         // ---> relations
23
24         @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL,
25                         mappedBy = "studio", orphanRemoval = true)
26         @Cascade(value = { org.hibernate.annotations.CascadeType.ALL })
27         private List<PArticleGroup> articleGroups = new ArrayList<PArticleGroup>();
28 }
```

Quelltext 5: Datenmodel der Studios in Java

Als nächster Bestandteil folgt die Serie. Es gilt jedoch vorher noch zu beachten, dass es sich bei dem Aufbau der Datenstruktur natürlich nicht um einen echten Baum, wie in der Auszeichnungssprache XML, handelt sondern dieser nur zum besseren Verständniss der Datenstruktur dienen sollte. Daher wird im folgenden der Begriff „Baumstruktur“ nicht mehr verwendet.

3.2.2 Serie

Eine Serie enthält genau wie das Studio einen Namen (NAME) sowie eine Beschreibung (DESCRIPTION) und ein Bild (IMAGE). Bei Serien kann der Zugriff eingeschränkt sein und wird bestimmt durch die Zugriffsart (ACCESS_TYPE). Eine Serie ist außerdem vergleichbar mit einem YouTube-Kanal wo Videos hochgeladen werden können. Sie kann somit bei Bedarf einen YouTube-Kanal spiegeln, wozu die YT-Kanal-ID (CHANNEL_ID_YOUTUBE) notwendig ist.

```java
@Getter
@Setter
@EqualsAndHashCode(callSuper = false)
@Entity
@Table(name = "ARTICLE_GROUP")
public class PArticleGroup extends PAbstractEntity {

        @Column(name = "ACCESS_TYPE", nullable = false, length = 16)
        @Enumerated(EnumType.STRING)
        @NotNull
        private EArticleAccessType articleAccessType;

        @Column(name = "NAME", unique = true, nullable = true, length = 128)
        private String name;

        @Lob
        @Column(name = "DESCRIPTION", columnDefinition = "TEXT")
        private String description;

        @Column(name = "CHANNEL_ID_YOUTUBE", unique = true, nullable = true)
        private String channelIdYoutube = null;

        @Column(name = "IMAGE", unique = false, nullable = false, length = 512)
        @NotNull
        private String image;

        // ---> relations

        @ManyToOne
        @JoinColumn(name = "FK_STUDIO_ID", nullable = false)
        private PStudio studio;

        @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL,
                        mappedBy = "articleGroup", orphanRemoval = true)
        @Cascade(value = { org.hibernate.annotations.CascadeType.ALL })
        private List<PArticle> articles = new ArrayList<PArticle>();
}
```

Quelltext 6: Datenmodel der Serien in Java

Durch die zwei Annotationen @ManyToOne in der Klasse PArticleGroup und @OneToMany in der Klasse PStudio ist sehr gut zu erkennen, wie das Erzeugen von Relationen mit der Java Persistence API (JPA) funktioniert. Hierdurch wird die 1:n-Relation gebildet, wodurch allein in der Tabelle für die Serie der entsprechende Fremdschlüssel für das Studio angelegt wird.

3.2.3 Beispiel in SQL

Zusammen mit der zuvor beschriebenen abstrakten Klasse, welche die Felder für alle Tabellen enthält, wird jedes Datenmodel ausschließlich in Java mittels Java Persistence API (JPA) definiert. Die JPA erzeugt daraus automatisch SQL-Code z.B. für das Studio und die Serie. Dieser soll als einmaliger Vergleich dienen, um

den Aufbau einer Datenbanktabelle mithilfe der JPA zu verstehen und wird bei den anderen Tabellen nicht wiederholt.

```
 1  CREATE TABLE IF NOT EXISTS STUDIO (
 2    ID BIGINT(20) NOT NULL AUTO_INCREMENT,
 3    DATE_CREATION BIGINT(20) NOT NULL,
 4    UUID VARCHAR(50) NULL DEFAULT NULL,
 5    IMAGE VARCHAR(512) NULL DEFAULT NULL,
 6    INTRO TEXT NULL DEFAULT NULL,
 7    NAME VARCHAR(128) NOT NULL,
 8    SLOGAN VARCHAR(1024) NULL DEFAULT NULL,
 9    PRIMARY KEY (ID),
10    UNIQUE INDEX UK_r4v390i8xj6d4gftu9u2jvw4n (UUID ASC),
11    UNIQUE INDEX UK_riol9i0suf9taneehgvd5jq6o (NAME ASC)
12  )
13
14  CREATE TABLE IF NOT EXISTS ARTICLE_GROUP (
15    ID BIGINT(20) NOT NULL AUTO_INCREMENT,
16    DATE_CREATION BIGINT(20) NOT NULL,
17    UUID VARCHAR(50) NULL DEFAULT NULL,
18    ACCESS_TYPE VARCHAR(16) NOT NULL,
19    CHANNEL_ID_YOUTUBE VARCHAR(128) NULL DEFAULT NULL,
20    DESCRIPTION TEXT NULL DEFAULT NULL,
21    IMAGE VARCHAR(512) NOT NULL,
22    NAME VARCHAR(128) NULL DEFAULT NULL,
23    FK_STUDIO_ID BIGINT(20) NOT NULL,
24    PRIMARY KEY (ID),
25    UNIQUE INDEX UK_mdcjmvs7br7gm10ou7s5194lr (UUID ASC),
26    UNIQUE INDEX UK_o5yhynwlsglr7nl14r1d9fhrn (CHANNEL_ID_YOUTUBE ASC),
27    UNIQUE INDEX UK_67nlbwj8m4qltumrr3clu2ml7 (NAME ASC),
28    INDEX FKfemyv3ncvu9v44ykeocjcbh2g (FK_STUDIO_ID ASC),
29    CONSTRAINT FKfemyv3ncvu9v44ykeocjcbh2g
30      FOREIGN KEY (FK_STUDIO_ID)
31      REFERENCES STUDIO (ID)
32      ON DELETE RESTRICT
33      ON UPDATE RESTRICT
34  )
```

Quelltext 7: Automatisch erzeugter SQL-Code für die Studio- und Serientabelle

Da sich das Schema für den Aufbau der Datenmodelle ab jetzt sehr ähnelt, werden für die weiteren Tabellen nur die Felder mit einer kurzen Beschreibung genannt, ohne den Quelltext hierfür abdrucken zu müssen.

3.2.4 Beitrag

Der letzte Bestandteil für die Mediendaten ist der Beitrag. Die im Anschluss beschriebenen Videoquellen gehören eigentlich mit zum Beitrag, haben jedoch aufgrund der entstandenen Komplexität ein eigenes Datenmodel. Wie bereits erwähnt soll jede Serie beliebig viele Beiträge enthalten können. Für die

Zuordnung zu der entsprechenden Serie dient der Fremdschlüssel FK_ARTICLEGROUP_ID. Folgende Liste zeigt alle Felder, die für einen Beitrag notwendig sind:

- ACCESS_TYPE (Zugriffstyp)
- DATE_PUBLISHED (Veröffentlichungsdatum)
- HEADING (Überschrift)
- IMAGE (Titelbild)
- INTRO (Einleitung/Beschreibung)
- UPDATE_CACHE (Indikator ob Aktualisierung notwendig ist)
- VIEWS (Anzahl der Aufrufe)
- FK_ARTIKLEGROUP_ID (Fremdschlüssel auf Serie)

Der Zugriffstyp (ACCESS_TYPE) bestimmt, wie bei einer Serie, ob der Beitrag einem Nutzer angezeigt wird oder nicht. Zudem enthält jeder Beitrag ein Veröffentlichungsdatum (PUBLISHED), eine Überschrift (HEADING), eine Beschreibung des Beitrags bzw. eine Einleitung (INTRO), sowie ein Titelbild (IMAGE). Um größerem Nutzeraufkommen auf der Plattform gerecht zu werden, sind häufig angefragte Daten zwischengespeichert. Wenn sich ab und zu Daten ändern sollten, muss dieser Zwischenspeicher dann aktualisiert werden. Das Feld UPDATE_CACHE dient als Indikator für solche Veränderungen eines Beitrags. Weiterhin exisitiert noch ein Feld für die Anzahl der Aufrufe (VIEWS) eines Beitrags. Dieses wird momentan aus den Aufrufen der einzelnen Videoquellen zusammen addiert. Zukünftig könnte hier noch ein Zähler für den direkten Aufruf eines Beitrages implementiert werden.

3.2.5 Videoquellen

Wie bereits erwähnt zählen die Videoquellen mit als Bestandteil eines Beitrags. Um jedoch eine Einschränkung der zu verwendenden Videoportale zu besitzen und dadurch beispielsweise die einzelnen Videos besser einbetten und Videodaten wie Dauer und Aufrufe aktualisieren zu können, existiert ein weiteres Datenmodell für die Videoquellen, welches folgende Felder enthält:

- DATE_API_UPDATE (letzter Aktualisierungszeitpunkt)
- VALID (Videoquelle noch gültig, ja/nein)
- VIDEO_ID (Video-ID auf dem Videoportal)
- VIDEO_PROVIDER (Videoanbieter/Videoportal)
- VIDEO_URL (Link zu dem Video auf dem Videoportal)
- VIDEO_DURATION (Videodauer)
- VIEWS (Anzahl der Aufrufe)

Eine Videoquelle besteht in erster Linie aus dem Videolink (VIDEO_URL), welcher zu dem Video verweisen und von dem Ersteller angegeben werden kann. Der Videoanbieter (VIDEO_PROVIDER) und die Video-ID (VIDEO_ID) werden dann aus dem Link extrahiert um die Daten hinter dem Link aktualisieren zu können. Außerdem kann der Link hiermit auch neu zusammengesetzt werden. Zu den aktualisierbaren Daten gehören einerseits die Videodauer (VIDEO_DURATION), welche konstant bleiben sollte sowie die Anzahl der Aufrufe (VIEWS). Wenn diese Daten nicht mehr von dem Videoportal abgerufen werden können muss davon ausgegangen werden, dass die Videoquelle nicht mehr gültig ist. Ein entsprechendes Flag für die Gültigkeit (VALID) kann dann dementsprechend gesetzt werden. Bei jeder Aktualisierung der Videoquelle wird der letzte Aktualisierungszeitpunkt (DATE_API_UPDATE) neu gesetzt. Das geschieht über sogenannte Cronjobs, welche im späteren Verlauf noch genauer beschrieben werden.

3.3 Datenmodelle für die Nutzerverwaltung

3.3.1 Nutzerkonto und Nutzerdaten

Nachfolgend sind sämtliche Felder für die beiden Datenmodelle Nutzerkonto und Nutzerdaten zu sehen.

Nutzerkonto	**Nutzerdaten**
DATE_ACKNOWLEDGED (Bestätigungsdatum)	FIRSTNAME (Vorname)
DATE_LAST_LOGIN (letztes Login-Datum)	LASTNAME (Nachname)
EMAIL (Email-Adresse)	NICKNAME (Spitzname)
IDENT (Nutzerkürzel)	FLAGS (Einstellungen)
PASSWORD (Passwort)	USERBANNER (Bannerbild)
ROLES (Berechtigungen des Nutzers)	USERIMAGE (Profilbild)
FK_ARTICLEGROUP_ID (FK für Serie)	USERINFO (Nutzer über sich)
FK_USER_DATA_ID (FK für Nutzerdaten)	

Tabelle 2: Datenmodelle mit ihren Feldern für die Nutzerverwaltung

Für die Anmeldung auf der Medienplattform wird das Nutzerkonto verwendet, welches die Email (EMAIL) des Nutzers und ein Passwort (PASSWORD) enthält. Für die leichtere Identifikation gibt es außerdem noch ein Nutzerkürzel (IDENT). Weiterhin wird gespeichert, welche Rolle (ROLES) der Nutzer auf der Plattform einnimmt, also ob er ein normaler Anwender, ein Moderator oder ein Administrator ist. Nach einer Registrierung auf der Medienplattform muss die Email-Adresse des Nutzerkontos über einen an diese Adresse versandten Link bestätigt werden. Ein entsprechendes Datum (DATE_ACKNOWLEDGED) wird dann gesetzt, bevor man sich auf der Plattform anmelden kann. Nach jeder Anmeldung wird immer das aktuelle Datum in ein eigenes Feld (DATE_LAST_LOGIN) gespeichert. Außerdem gibt es den Fremdschlüssel für eine Serie (FK_ARTICLEGROUP_ID), welche der

Nutzer verwalten kann, und den Fremdschlüssel für die Nutzerdaten (FK_USER_DATA_ID).

In den Nutzerdaten des Kontos kann nun zusätzliche Information wie Vorname (FIRSTNAME), Nachname (LASTNAME), Spitzname (NICKNAME), ein Profilbild (USERIMAGE) sowie ein Bannerbild (USERBANNER) eingestellt werden. Jeder Nutzer kann außerdem noch Information (USERINFO) über sich angeben, welche den anderen Nutzern angezeigt werden. Zuletzt gibt es noch viele Einstellmöglichkeiten über Kontrollkästchen, um beispielsweise Benachrichtigungen zu aktivieren oder zu deaktivieren. Diese werden als Zeichenkette über das Feld FLAGS in der Datenbank gespeichert.

3.3.2 Eine Serie abonnieren

Das Datenmodell für die Nutzerabonnements ist relativ klein gehalten. Es besteht im wesentlichen nur aus zwei Relationen was im folgenden nochmal mithilfe der Java Persistence API (JPA) aufgezeigt werden soll.

```
1  @Getter
2  @Setter
3  @EqualsAndHashCode(callSuper = false)
4  @Entity
5  @Table(name = "USER_SUBSCRIPTION")
6  public class PUserSubscription extends PAbstractEntity {
7
8          @Column(name = "IS_ACTIVE", nullable = false)
9          @NotNull
10         private boolean active = false;
11
12         //
13         // ---> relations
14         //
15
16         @ManyToOne
17         @JoinColumn(name = "FK_USER_ACCOUNT_ID", nullable = false)
18         private PUserAccount userAccount;
19
20         @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
21         @JoinColumn(name = "ARTICLE_GROUP_ID")
22         private PArticleGroup articleGroup = new PArticleGroup();
23 }
```

Quelltext 8: Datenmodel des Nutzerabonnements in Java

Die Nutzer können mit diesem Datenmodel jede verfügbaren Serien abonnieren. Es wird lediglich in der Tabelle für Abonnements ein neuer Eintrag angelegt, welcher mit dem entsprechenden Nutzer (FK_USER_ACCOUNT_ID) und der entsprechenden Serie (FK_ARTICLE_GROUP_ID) verknüpft ist. Sollte die Serie von dem Nutzer wieder deabonniert werden, wird ein zusätzliches Feld von aktiv (IS_ACTIVE) auf nicht aktiv gesetzt.

3.4 Zugriff auf die Daten mittels JpaRepository

Sind die Datenmodelle für die Medienplattform angelegt, muss auch der Zugriff auf die Daten gewährleistet werden. Das geschieht mittels JpaRepository, welches von Haus aus schon jede Menge Methoden zum Abrufen, Speichern und Löschen von Datenbankeinträgen mit sich bringt. Zusätzlich können eigene Datenbankabfragen entweder über die Annotation @Query oder über den Methodennamen implementiert werden. Es ist hierbei nur das Interface notwendig, da die Implementation der Methoden automatisch von Hibernate übernommen wird. Dies soll Anhand des JpaRepositories für die Beiträge kurz erläutert werden.

```
1  public interface PArticleRepository extends JpaRepository<PArticle, Long> {
2
3          List<PArticle> findByArticleGroup(final PArticleGroup articleGroup);
4
5          PArticle findByHeading(final String heading);
6
7          List<PArticle> findByHeadingContainingIgnoreCase(final String heading);
8
9          Long countByArticleGroup(final PArticleGroup articleGroup);
10
11         @Query("select count(particle) from PArticle particle inner join
12                 particle.articleGroup.studio studio where studio.id = :studioid")
13         Long countByStudioId(@Param(value = "studioid") final Long studioId);
14
15         Long countByDateCreationGreaterThan(final Long date);
16
17         Long countByArticleAccessTypeIn(
18                 final List<EArticleAccessType> articleAccessTypes);
19
20         PArticle findByUuid(final String uuid);
21
22         Page<PArticle> findByDateCreationLessThanAndDateCreationGreaterThan(
23                 final Long date1, final Long date2, final Pageable pageable);
24 }
```

Quelltext 9: Erweiterung eines JpaRepository für den Zugriff auf die Beiträge

Wie zu sehen ist, können hier sehr viele verschiedene Arten von Datenbankanfragen ganz leicht über den Methodennamen sowie über den Rückgabewert erzeugt werden. Die Namen der Parameter sind hierbei irrelevant, lediglich der übergebene Typ muss mit dem im Datenmodel übereinstimmen. Eine Übersicht über alle Möglichkeiten sind auf der Webseite des Spring-Frameworks (spring.io) zu finden. Das so erzeugte PArticleRepository kann nun über die Spring-Annotation @Autowired z.B. in eine Service-Klasse oder gleich über die Tapestry-Annotation @Inject direkt in eine Tapestry-Seite eingebunden werden.

```
1  // Service-Klasse in Spring
2  public class ArticleAccessService implements IArticleAccessService {
3
4          @Autowired
5          private PArticleRepository pArticleRepository;
6
7          @Override
8          public PArticle getArticleByHeading(final String heading) {
9
10                 return pArticleRepository.findByHeading(heading);
11         }
12 }
13
14 // Tapestry-Seite
15 public class InvalidArticles {
16
17         @Inject
18         private PArticleRepository pArticleRepository;
19
20         @Property
21         private List<PArticle> articles;
22
23         @SetupRender
24         void setup() {
25
26                 articles = pArticleRepository
27                         .findAllByValidIsFalseAndArticleAccessTypeIn(
28                                 ArticleRepository.VISIBLE_ARTICLES);
29         }
30 }
```

Quelltext 10: Dependency Injection vom Beitrags-Repository

Das gleiche Prinzip gilt dann auch für die anderen Datenmodelle. Für die Studios heißt das JpaRepository dann dementsprechend PStudioRepository und für die Serien ParticleGroupRepository. Das Einbinden über @Autowired bzw. @Inject wird daher in den weiteren Quelltexten nicht weiter dargestellt.

4. Verwaltung der Mediendaten

4.1 Anlegen eines Studios mit einer Serie

Wie bereits beim Aufbau der Datenbankstruktur erwähnt, gibt es Studios mit Serien, welche wiederum die Beiträge enthalten. Ein Studio kann über eine Tapestry-Seite angelegt werden, die verkürzt folgendermaßen aussieht.

```
<div xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd"
        xmlns:p="tapestry:parameter" t:id="template">
...
<t:form t:id="studioform" class="" t:autofocus="false">
        <t:beaneditor t:id="studio" object="pStudio" include="name, intro, slogan">
                <p:intro>
                        <label for="intro">Beschreibung</label>
                        <t:textarea t:id="intro" value="pStudio.intro" />
                </p:intro>
        </t:beaneditor>
        <hr />
        <div class="t-beaneditor-row">
                <t:submit value="Speichern" />
        </div>
</t:form>
...
</div>
```

Quelltext 11: Tapestry XML-Vorlage mit Beaneditor für Studiobearbeitung

Zusammen mit der Java-Klasse, welche weiter unten erläutert wird, resultiert das in folgendem Formular.

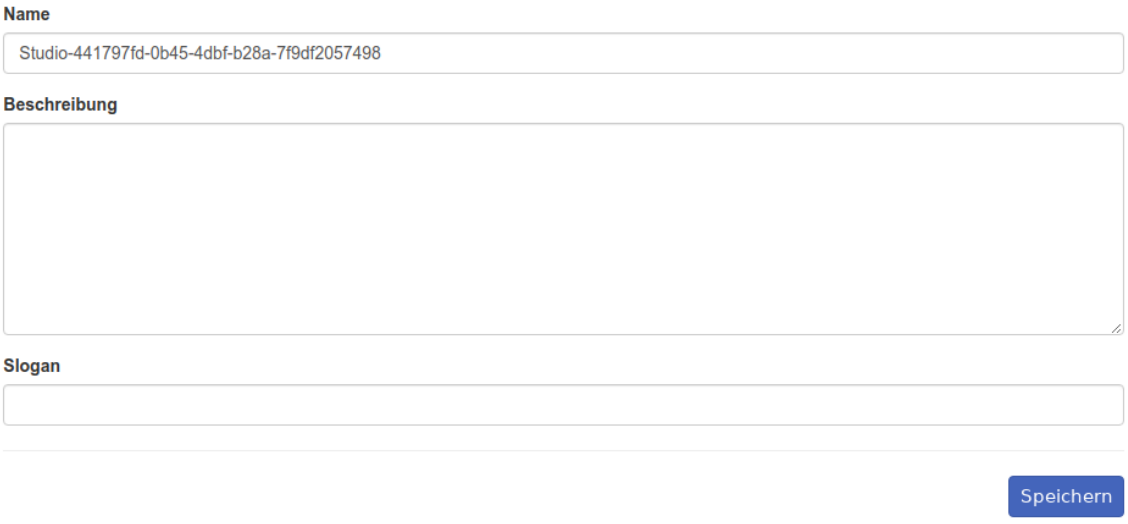

Abbildung 3: Formular für die Studiobearbeitung im Webbrowser

Der Beaneditor von Tapestry stellt eine Möglichkeit dar, die Werte eines Objekts über ein Formular zu manipulieren, ohne jedes Formularelement explizit in HTML-Code zu implementieren. Für die einzelnen Felder werden automatisch passende Eingabemöglichkeiten nach dem entsprechenden Datentyp (String, Integer, Enum), ausgewählt. Gegebenenfalls kann eine automatische Auswahl mit einem eigenen Element überschrieben werden, z.B. mit einer Textarea für einen längeren Text mit Zeilenumbrüchen. Die vom Nutzer erstellten Daten können dann in der Java-Klasse der Tapestry-Seite ausgewertet werden, welche im folgenden zu sehen ist.

```java
@RequiresAuthentication
public class StudioEditor {

	@Property
	@PageActivationContext(index = 0)
	private Long studioId;

	@Property
	private PStudio pStudio;

	@SetupRender
	void setup() {

		initStudio();
	}

	void onPrepareForSubmit() {

		initStudio();
	}

	Object onSuccessFromStudioForm() {

		final PStudio savedStudio = pStudioRepository.save(pStudio);
		alertManager.success("saved studio with id=" + studioId);
		return prls.createPageRenderLinkWithContext(
			AdminStudioEditor.class,savedStudio.getId());
	}

	private void initStudio() {

		if (studioId != null && studioId > 0) {
			pStudio = pStudioRepository.findById(studioId).get();
		} else {
			pStudio = new PStudio();
			pStudio.setName("Studio-" + pStudio.getUuid());
			pStudio.setImage("https://test.de/image.jpg");
		}
	}
}
```

Quelltext 12: Java-Klasse der Tapestry-Seite für die Studiobearbeitung

Die Studiodaten müssen jeweils zwei mal für die Webseite neu aufbereitet werden. Das ist einerseits beim Laden der Seite und weiterhin beim Übermitteln des Formulars. Daher bietet es sich an, das Ganze in eine separate Methode mit dem Namen „initStudio()“ auszulagern. Dort wird mit der vom Nutzer übermittelten studioId das vorhandene Studio zur Bearbeitung geladen bzw. ein neues Studio temporär angelegt. Nach der Bearbeitung über die UI-Elemente und dem Abschicken des Formulars wird das Studio über das StudioRepository in der Datenbank gespeichert. Ähnlich verhält sich das für die Serie, bei der jedoch nur die Java-Klasse gezeigt werden soll.

```
@RequiresAuthentication
public class ArticleGroupEditor {

	@Property
	@PageActivationContext(index = 0)
	private Long articleGroupId;

	@Property
	@PageActivationContext(index = 1)
	private Long studioId;

	@Property
	private PArticleGroup pArticleGroup;

	@SetupRender
	void setup() {

		initArticleGroup();
	}

	void onPrepareForSubmit() {

		initArticleGroup();
	}

	Object onSuccessFromArticleGroupForm() {

		final PArticleGroup savedArticleGroup = pArticleGroupRepository
			.save(pArticleGroup);
		alertManager.success("saved aGroup with id=" + articleGroupId);
		return prls.createPageRenderLinkWithContext(
			AdminArticleGroupEditor.class, savedArticleGroup.getId());
	}

	private void initArticleGroup() {

		if (articleGroupId != null && articleGroupId > 0) {
			pArticleGroup = pArticleGroupRepository
				.findById(articleGroupId).get();
		} else {
			if (studioId != null && studioId > 0) {
				final PStudio pStudio = pStudioRepository
					.findById(studioId).get();
				pArticleGroup = new PArticleGroup();
				pArticleGroup.setStudio(pStudio);
				pArticleGroup.setName("ArticleGroup-" +
				pArticleGroup.getUuid());
				pArticleGroup.setImage("https://test.de/image.jpg");
				pArticleGroup.setArticleAccessType(
					EArticleAccessType.BLOCKED);
			} else {
				throw new IllegalStateException(
					"invalid articleGroupId or studioId!");
			}
		}
	}
}
```

Quelltext 13: Java-Klasse der Tapestry-Seite für die Serienbearbeitung

Im Unterschied zum Studio wird hier bei der Serie eine Exception geworfen, wenn keine Studio-ID angegeben wurde und eine neue Serie angelegt werden soll. Jeder Serie muss ein Studio zugewiesen werden, um überhaupt angelegt werden zu können.

4.2 Erstellung eines neuen Beitrags

Sobald ein Studio mit einer Serie existiert, können in dieser Serie die Beiträge angelegt werden. Das Prinzip gleicht dem beim Anlegen einer Serie, wo der entsprechenden Seite eine Studio-ID übergeben wurde. In diesem Fall kann eine gültige ArticleGroup-ID (Serien-ID) übergeben werden. Alternativ reicht auch die Anmeldung eines entsprechenden Nutzers, dem die Serie zur Verwaltung zugeordnet wurde. Das wird ermöglicht durch die Verknüpfung eines Nutzers mit einer Serie in der Datenbank durch den entsprechenden Fremdschlüssel (FK_ARTICLEGROUP_ID) in der Nutzerkonto-Tabelle (USER_ACCOUNT), was im Kapitel über den Aufbau der Datenbankstruktur erläutert wurde. Hier soll zunächst wieder nur die Java-Klasse für die Beitragserstellung beschrieben werden. Im folgenden sind Ausschnitte aus dieser zu sehen.

```
1  @RequiresAuthentication
2  public class EditArticle {
3
4  ...
5
6          Object onSuccess() {
7
8                  pArticle.setDatePublished(System.currentTimeMillis());
9                  pArticle = articleAccessService.save(pArticle);
10                 alertManager.success("saved article with uuid=" + studioUuid);
11                 return prls.createPageRenderLinkWithContext(
12                         EditArticle.class, pArticle.getUuid());
13         }
14
15         public String getVideoSources() {
16
17         ...
18
19         }
20
21
22         public void setVideoSources(final String sources) {
23
24
25         ...
26
27         }
28
29         private void initArticle() {
30
31                 final Subject subject = securityService.getSubject();
32                 final String userUuid = Frei3SecurityUtils
33                         .getUuidFromSubject(subject);
34                 final PArticleGroup pArticleGroup = articleAccessService
```

```
35                          .getArticleGroupFromUserAccount(userUuid);
36
37                  if (articleUuid == null) {
38                          pArticle = new PArticle();
39                          pArticle.setArticleGroup(pArticleGroup);
40                          pArticle.setArticleVideos(new ArrayList<PArticleVideo>());
41                          pArticle.setHeading("Article-" + pArticle.getUuid());
42                          pArticle.setImage("https://test.de/image.jpg");
43                          pArticle.setDatePublished(System.currentTimeMillis());
44                          pArticle.setArticleAccessType(EArticleAccessType.BLOCKED);
45                  } else {
46                          pArticle = articleAccessService
47                                  .getArticleByUuid(articleUuid);
48                          if (pArticle == null || account.getArticleGroup()
49                                  .equals(pArticle.getArticleGroup()) == false) {
50                                  throw new IllegalAccessError("cannot edit article");
51                          }
52                  }
53          }
54  }
```

Quelltext 14: Java-Klasse der Tapestry-Seite für die Beitragsbearbeitung

Während der Initialisierung der Seite über „initArticle()“ wird wieder entweder der vorhandene Beitrag geladen bzw. temporär ein neuer erzeugt. Wie bereits erwähnt kann eine Serie (PArticleGroup) auch einem Nutzerkonto zugewiesen werden. Sollte dies der Fall sein, wird dem neuen Beitrag gleich bei der Initialisierung die richtige Serie zugewiesen. Für den Fall, dass bereits ein Beitrag unter der übergebenen articleUuid vorhanden ist, wird geprüft, ob dieser der gleichen Serie zugeordnet wurde. Sollte das nicht so sein, ist der Zugriff auf den Beitrag nicht erlaubt und eine entsprechende Exception wird geworfen.

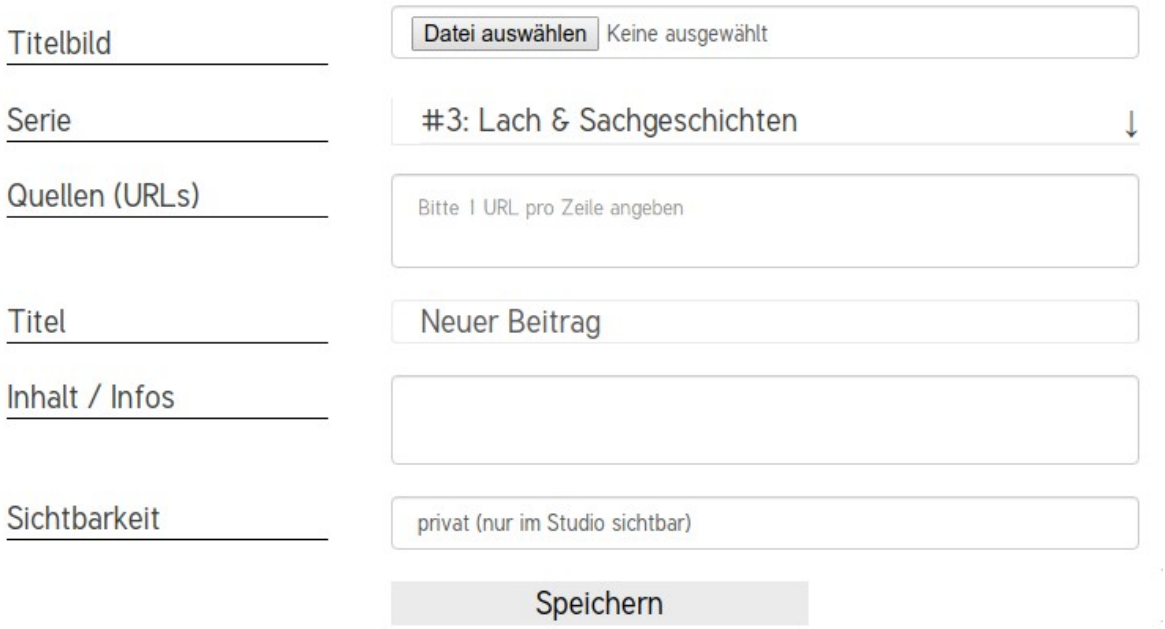

Abbildung 4: Formular für die Beitragsbearbeitung im Webbrowser

Sobald der Beitrag bearbeitet und das Formular abgesendet wurde (onSuccess), werden die Änderungen mit dem aktualisierten Veröffentlichungsdatum in der Datenbank gespeichert. Eine besondere Rolle wird hierbei den Videoquellen zuteil, denn für dieses Formularfeld existieren zwei separate Methoden für das Laden und anschließende Speichern nach der Bearbeitung der Videoquellen.

4.3 Angabe der Videoquellen

Bei der Bearbeitung des Beitrags werden die Videoquellen nur als eine Liste von URLs angezeigt und können ebenso bearbeitet werden. Nach dem Absenden des Formulars werden die einzelnen URLs analysiert und auf ihre Gültigkeit geprüft. Anschließend werden die gewonnenen Information in der Datenbank gespeichert. Das Laden der Videoquellen ist noch relativ einfach. In der Methode „getVideoSources()“ werden die Videoquellen eines Beitrags als Liste geladen. Jede Videoquelle beinhaltet die URL als Feld, sodass diese in einer Zeichenkette durch Zeilenumbrüche getrennt dem Formularfeld übergeben werden können.

```
1  public String getVideoSources() {
2
3          final List<PArticleVideo> videoList = pArticle.getArticleVideos();
4          String result = "";
5
6          for (final PArticleVideo video : videoList) {
7                  result += video.getVideoUrl() + "\n";
8          }
9
10         return result;
11 }
12
13 public void setVideoSources(final String sources) {
14
15         final Tuple2<PArticle, List<String>> tuple = articleAccessService
16                 .parseVideoSourcesAndWriteToDatabase(sources, pArticle);
17         pArticle = tuple.getA();
18
19         for (final String message : tuple.getB()) {
20                 alertManager.error(message);
21         }
22 }
```

Quelltext 15: Methoden zum Abrufen und Sichern der Videoquellen

Dagegen ist das Analysieren der Videoquellen in der Methode „setVideoSources()“ weitaus komplexer. Da das Formularelement ein einfaches Textfeld ist, sind dementsprechend auch beliebige Eingaben vom Nutzer möglich. Daher wird die Zeichenkette mit den URLs einer speziellen Methode in einem Service außerhalb der Tapestry-Seite übergeben und dort verarbeitet. Sollten Fehler bei der

Verarbeitung auftreten, werden diese von der Methode zurückgegeben, sodass diese anschließend dem Nutzer gemeldet werden können. Die Methode parseVideoSourcesAndWriteToDatabase(), welche im weiteren Verlauf noch näher erklärt wird, ist im groben folgendermaßen aufgebaut:

- Text in einzelne Zeilen (String-Array) zerlegen (eine URL pro Zeile)
- fehlende Videoquellen aus der Datenbank löschen
- neue Videoquellen auf Gültigkeit prüfen und in die Datenbank schreiben

Das Zerlegen des Textes mit den Videoquellen in ein Array aus den einzelnen Zeilen erfolgt über einen einfachen regulären Ausdruck unter Zuhilfenahme der String-Methode „split()“. Sollte kein Text zum Zerlegen vorhanden sein, wird ein leeres Array erzeugt. Danach wird eine Liste mit den vorhandenen Videoquellen des Beitrags geladen und geprüft, welche von den URLs noch in dem Array vorhanden sind. Alle nicht auffindbaren Videoquellen werden anschließend aus der Datenbank gelöscht.

```
1  @Override
2  public Tuple2<PArticle, List<String>> parseVideoSourcesAndWriteToDatabase(
3                  final String sources, final PArticle pArticle) {
4
5          final String[] lines;
6          if (sources == null) {
7                  lines = new String[0];
8          } else {
9                  lines = sources.split("\\s*\\r?\\n\\s*");
10         }
11
12         final List<PArticleVideo> videoList = pArticle.getArticleVideos();
13
14         // remove videos
15         videoList.removeIf(new Predicate<PArticleVideo>() {
16
17                 @Override
18                 public boolean test(PArticleVideo t) {
19
20                         for (final String line : lines) {
21                                 if (line.compareToIgnoreCase(t.getVideoUrl()) == 0) {
22                                         return false;
23                                 }
24                         }
25                         return true;
26                 }
27         });
28
29         pArticle.setArticleVideos(videoList);
30         final PArticle savedPArticle = pArticleRepository.save(pArticle);
```

```
31
32          final List<String> messages = new ArrayList<String>();
33          // add videos
34          for (final String line : lines) {
35                  if (UrlUtils.isValidURL(line)) {
36                          final EVideoProvider eVP = VideoUtils
37                                  .getVideoProviderFromUrl(line);
38                          if (eVP == null) {
39                                  messages.add("unknown videoprovider (" + line + ")");
40                                  continue;
41                          }
42                          final String vId = VideoUtils.getVideoIdFromUrl(line, eVP);
43                          if (vId == null) {
44                                  messages.add("videoid not found (" + line + ")");
45                                  continue;
46                          }
47
48                          boolean test = false;
49                          for (final PArticleVideo video : videoList) {
50                                  final String url = VideoUtils
51                                          .getVideoUrlFromProviderAndID(eVP, vId);
52                                  if (url.equalsIgnoreCase(video.getVideoUrl())) {
53                                          test = true;
54                                          break;
55                                  }
56                          }
57                          if (test == false) {
58                                  addVideoToArticle(pArticle, eVP, vId);
59                          }
60                  } else {
61                          messages.add("invalid URL (" + line + ")");
62                  }
63          }
64
65          return new Tuple2<PArticle, List<String>>(savedPArticle, messages);
66  }
```

Quelltext 16: Methode zum Parsen mehrerer Videoquellen aus einem String

Jetzt folgt die eigentliche Analyse der neuen Videoquellen. Jede URL wird zuerst auf erlaubte Videoanbieter wie YouTube oder Bitchute kontrolliert. Dadurch offenbart sich das URL-Schema des jeweiligen Anbieters und es kann anschließend die Video-ID aus der URL extrahiert werden. Wenn auch das erfolgreich war, erfolgt die Prüfung, ob die Videoquelle zu dem Beitrag bereits in der Datenbank auftaucht. Ist auch das der Fall, wird die Videoquelle hinzugefügt. Sämtliche Fehlschläge werden in eine Liste aufgenommen und als Rückgabe der aufrufenden Methode zusammen mit dem aktualisierten Beitrag übergeben.

5. Darstellung der veröffentlichten Mediendaten

5.1 Übersicht der Darstellungsmöglichkeiten

Neben der Verwaltung der Mediendaten, also dem Erstellen und Bearbeiten von Studios, Serien und Beiträgen, müssen diese dem potenziellen Nutzer auch zur Verfügung gestellt werden. Das erfolgt im einfachsten Fall über eine Suchanfrage, die über eine Suchleiste im oberen Bereich der Webseite zur Verfügung gestellt wird. Weiterhin werden die neuesten Beiträge gleich auf der Startseite angezeigt, sodass sie für die Nutzer als solche erkennbar sind. Die Sortierung erfolgt dementsprechend nach dem Veröffentlichungsdatum. Für angemeldete Nutzer gibt es eine Seite „Neues aus Abos“, wo nur die neusten Beiträge aus den abonnierten Serien des Nutzers angezeigt werden.

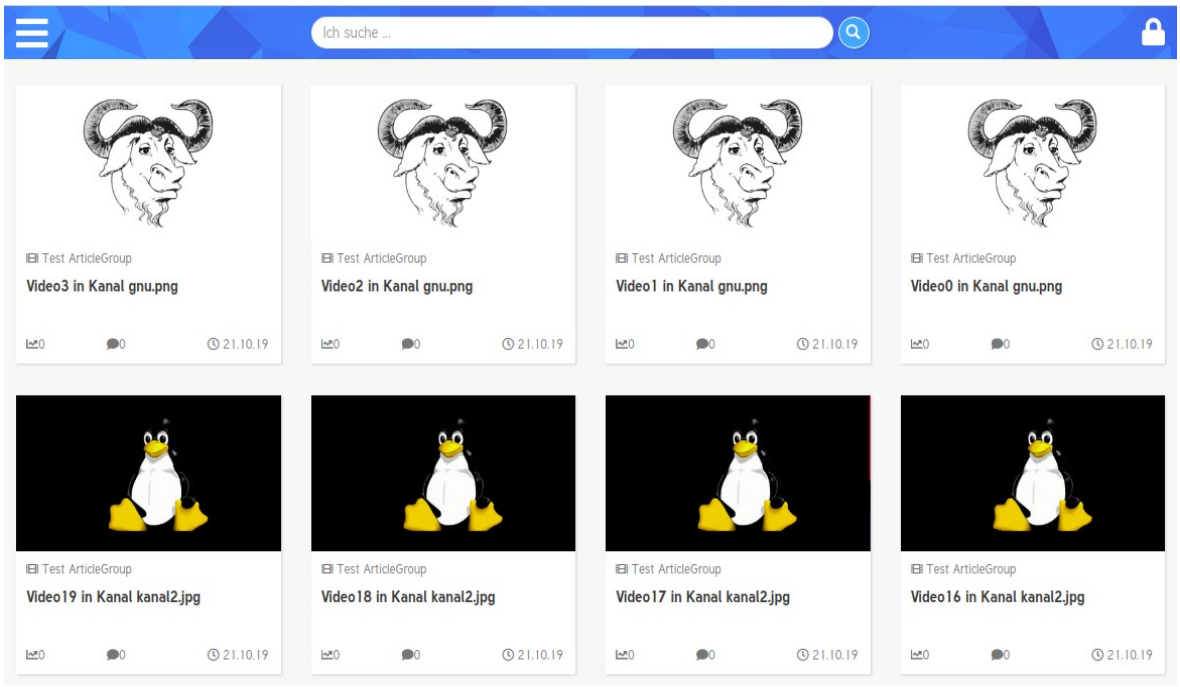

Abbildung 5: Startseite der Medienplattform mit Vorschaukarten von Beiträgen

Für das Anfragen der Daten kann entweder der gleiche Datenbankserver wie bei der Verwaltung der Mediendaten verwendet werden bzw. eine spezielle Suchmaschine, z.B. Elasticsearch, die mit den Einträgen aus der Datenbank gefüllt wird. Zur Anzeige der einzelnen Suchergebnisse oder den Beiträgen auf der Startseite wird eine individuelle Vorschaukarte verwendet, worin ein Teil der Daten wie Titel und Bild für die Beiträge, Serien oder Studios schon angezeigt werden.

5.2 Vorschaukarte für Mediendaten

Da auf der Startseite und auf den Suchseiten die aufgelisteten Daten stets im selben Format angezeigt werden, lohnt es sich hierfür jeweils eine eigene Tapestry-Komponente der Vorschaukarte für Beiträge, Serien und Studios zu implementieren, welche später über eine Schleife mehrfach aufgerufen werden kann. Da sich der Aufbau nahezu gleicht, soll hier die Vorschaukarte für die Beiträge als Beispiel reichen. Im folgenden ist eine vereinfachte Version der Komponente zu sehen.

```
1  <t:pagelink page="Article" context="article.uuid">
2
3          <img src="${article.imagesmall}" onerror="this.src = '${errorImage}';" />
4  ...
5          <i class="far fa-clock"></i>
6          ${duration}
7  ...
8          <i class="fas fa-film"></i>
9          ${article.articleGroupName}
10 ...
11         ${article.heading}
12 ...
13         <i class="fas fa-chart-line"></i>
14         ${article.views}
15 ...
16         <i class="far fa-clock"></i>
17         ${datePublished}
18
19 </t:pagelink>
```

Quelltext 17: Tapestry XML-Vorlage mit Pagelink für die Vorschaukarte

Gekapselt wird die Vorschaukarte von einer PageLink-Komponente, welche einen Link in HTML erzeugt, der auf die entsprechende Tapestry-Seite verweist, welche in dem Attribut „page“ übergeben wurde. Über „context“ wird die UUID des Beitrags übergeben, um später den richtigen Beitrag auf der Beitragsseite anzuzeigen. Die Bestandteile aller Vorschaukarten sind unter anderem ein Bild, Titel und das Veröffentlichungsdatum. Dem Bild (<img>-Tag) wird noch ein Fehlerbild übergeben, falls das Objekt keinen gültigen Link enthält. In der Java-Klasse wird dieses über ein „Asset“ geladen. Mithilfe der Notation ${} werden Java Objekte in Strings umgewandelt und direkt an der entsprechenden Stelle im HTML-Code eingefügt. Die Java-Klasse zu der Komponente ist folgendermaßen aufgebaut.

```
1  public class ArticleItemView {
2
3          @Property
4          @Inject
5          @Path("context:static/images/articleImageError.jpg")
6          private Asset errorImage;
7
8          @Parameter(required = true)
9          @Property
10         private PArticle article;
11
12         @Property
13         private String duration;
14
15         @SetupRender
16         void setup() {
17
18                 duration = "XX:XX";
19                 for (final PArticleVideo articleVideo: article.getArticleVideos()) {
20                         if (articleVideo.getVideoDuration() > 0) {
21                                 duration = DateTimeUtils.printDuration(
22                                         articleVideo.getVideoDuration());
23                                 break;
24                         }
25                 }
26         }
27
28         public String getDatePublished() {
29
30                 if (article.getDatePublished() != null) {
31                         final Long dateTime = article.getDatePublished();
32                         return DateTimeUtils.printDateShort(dateTime);
33                 }
34                 return "";
35         }
36 }
```

Quelltext 18: Java-Klasse der Tapestry-Komponente für die Vorschaukarte

Da jede Vorschaukarte einen anderen Beitrag darstellen soll, ist es hier sinnvoll den entsprechenden Beitrag als Parameter zu übergeben. Dies geschieht über die Annotation @Parameter. Über die zusätzlichen Option „required = true“ wird eingestellt, dass dieser Paramter der Komponente zwingend mitgeteilt werden muss. In der SetupRender Methode wird die Dauer einer Videoquelle von dem Beitrag geholt. Weil davon auszugehen ist, dass sämtliche Videoquellen das gleiche Video verlinken, muss somit auch die Länge identisch sein. Zusätzlich wird über eine weitere Methode das Veröffentlichungsdatum vom Unix-Format in eine für Menschen lesbare Form umgewandelt. In den nächsten Abschnitten soll nun diese Komponente bzw. die Entsprechungen für die Vorschaukarte der Serie und des Studios verwendet werden.

5.3 Neueste Beiträge auf der Startseite

Der entscheidende Teil in der tml-Datei der Tapestry-Seite für die Auflistung der Beiträge ist relativ kurz. Hier wird eine Schleifen-Komponente von Tapestry verwendet, die einer „for...in“-Schleife gleicht. Dem Attribut „source“ wird eine Liste von Objekten, in diesem Fall der neuesten Beiträge, übergeben. Die einzelnen Elemente der Liste läd Tapestry in das Objekt, das bei dem Attribut „value“ angegeben wurde. So können die einzelnen Beiträge der zuvor beschriebenen Komponente für die Vorschaukarte übergeben werden.

```
1  <t:if test="articles">
2          <t:loop t:source="articles" t:value="loopArticle">
3                  <t:articleitemview article="loopArticle" />
4          </t:loop>
5  </t:if>
```

Quelltext 19: Einbindung der Vorschaukartenkomponente auf der Startseite

In der zugehörigen Java-Klasse wird die Liste mit den Beiträgen und die Laufvariable für einen einzelnen Beitrag angelegt.

```
1  public class Index {
2
3          @Property
4          private List<PArticle> articles;
5
6          @Property
7          private PArticle loopArticle;
8
9          @SetupRender
10         void setup() {
11
12                 final Sort sort = new Sort(Direction.DESC, "datePublished");
13                 final Pageable pageable = PageRequest.of(0, 32, sort);
14                 articles = pArticleRepository.findAll(pageable);
15         }
16 }
```

Quelltext 20: Java-Klasse der Tapestry-Seite für die Startseite

Über die SetupRender-Methode wird die Beitragsliste initialisiert. Das passiert durch eine Datenbankanfrage mithilfe des ArticleRepository, dem mittels eines Pageable-Objekts die nötigen Information hierfür übergeben werden. In diesem Fall werden aus der Datenbank die ersten 32 Beiträge sortiert nach dem Veröffentlichungsdatum abgerufen.

5.4 Suche nach Studios, Serien und Beiträgen

5.4.1 Allgemeiner Hinweis

Jede Plattform, auf der eine unbestimmte Menge an verschiedensten Inhalten bereitstellt wird, sollte über eine Suchfunktion verfügen, um dem potenziellen Nutzer die entsprechend gewünschten Inhalte leichter zugänglich zu machen. Hierzu empfiehlt es sich eine spezielle Suchmaschine, wie Elasticsearch, zu verwenden, welche die vorhandene Datenbank entlastet. Es ist ein Cronjob notwendig, der die in der Datenbank vorhandenen Daten in zyklischen Abständen in die Suchmaschine einpflegt bzw. aktualisiert. Da die Anfragen an die Suchmaschine wie bei der Datenbank mittels eines Repositories vereinfacht wären und sich so nahezu gleichen, wird in dieser Abhandlung die Suche nach Studios, Serien oder Beiträgen direkt in der vorhandenen Datenbank durchgeführt.

5.4.2 Suchanfrage verarbeiten und Ergebnis anzeigen

Das Formular für die Suche besteht lediglich aus zwei Komponenten. Das ist erstens das Eingabefeld für den Suchbegriff und zweitens die Schaltfläche, um das Formular mit dem eingegebenen Suchbegriff abzusenden.

```
<t:form t:id="searchForm" role="search" t:autofocus="false">
        <div class="input-group">
                <t:textfield t:id="searchValue" placeholder="Ich suche ..." />
                <div class="input-group-btn">
                        <button type="submit">
                                <i class="glyphicon glyphicon-search"></i>
                        </button>
                </div>
        </div>
</t:form>
<t:if test="articles">
        <t:loop source="articles" value="loopArticle">
                <t:articleitemview article="loopArticle" />
        </t:loop>
        <p:else>
                <p>Es konnten keine Beiträge gefunden werden.</p>
        </p:else>
</t:if>
```

Quelltext 21: Tapestry XML-Vorlage für die Suchseite eines Beitrags

Für die Anzeige der Suchergebnisse kommt wieder die „for...in“-Schleife von Tapestry (<t:loop>) zum Einsatz, in der die gefundenen Beiträge (articles) einzeln der Vorschaukarten-Komponente mittels einer Schleifenvariable (loopArticle) übergeben werden. Da es bei einer Suchanfrage auch mal vorkommen kann, dass keine Ergebnisse zurückkommen, wird für diesen Fall dem Nutzer eine spezielle Meldung angezeigt. Um eine solche bedingte Auswertung zu ermöglichen, gibt es in Tapestry hierfür eine if-Komponente, in der die Schleife ausgeführt werden kann, mit einem else-Zweig, der die Meldung des Nutzers ausgibt. Zurück zum Formular, das in der folgenden Java-Klasse ausgewertet wird.

```
1  public class SearchHits {
2
3          @Property
4          private List<PArticle> articles;
5
6          @Property
7          private PArticle loopArticle;
8
9          @Property
10         @PageActivationContext(index = 0)
11         private String searchValue;
12
13         @SetupRender
14         void setup() {
15
16                 final Sort sort = new Sort(Direction.DESC, "datePublished");
17                 final Pageable pageable = PageRequest.of(0, 32, sort);
18                 articles = pArticleRepository
19                         .findAllByHeadingContainsOrIntroContains(
20                                 searchValue, searchValue, pageable);
21         }
22
23         Object onSuccessFromSearchForm() {
24
25                 return prls.createPageRenderLinkWithContext(searchValue);
26         }
27 }
```

Quelltext 22: Java-Klasse für die Suchseite eines Beitrags

Die Methode onSuccessFromSearchForm() wird von Tapestry aufgerufen, nachdem das Formular vom Nutzer abgesendet worden ist. In der Methode wird ganz einfach ein Link erzeugt, der die gesamte Seite neu Rendern lässt, sobald dieser als Rückgabewert von der Methode zurückgegeben wird. Der im Eingabefeld des Suchformulars eingetragene und übermittelte Suchbegriff wird dem Link angefügt, sodass dieser sich auch beim Neuladen der Seite immer noch in dem String „searchValue“ befindet. In der setup()-Methode wird nun das Repository für die Beiträge genutzt, um aus der Datenbank die Einträge zu holen, welche den Suchbegriff beinhalten. Das gleiche Prinzip wird so nicht nur für die Beitragssuche, sondern auch für die Serien- und Studiosuche angewendet.

5.4.3 Autovervollständigung bei der Eingabe

Für die Autovervollständigung des Suchbegriffes in dem Eingabefeld des Formulares existiert in Tapestry ein spezielles Mixin. Mixins erweitern in Tapestry

eine Komponente um ein oder mehrere Funktionalitäten. Mixins, in diesem Fall jenes für die Autocomplete-Funktionalität, werden wie folgt über ein Attribut der Komponente übergeben.

```
<t:textfield t:id="searchValue" placeholder="Ich suche ..." t:mixins="autocomplete“ />
```

Quelltext 23: Tapestry-Textfield wird um ein Autocomplete-Mixin erweitert

Zur Steuerung der Autocomplete-Funktionalität ist nur eine weitere Methode in der zuvor beschriebenen Java-Klasse nötig. Dieser wird als Parameter die bisherige Eingabe in das Textfeld übergeben und erwartet als Rückgabe eine Liste von Strings als mögliche Eingaben in das Textfeld. Die Methode wird bei jeder Änderung des Textfeldes erneut aufgerufen und sollte daher eher weniger rechenintensiv aufgebaut sein.

```
1  List<String> onProvideCompletionsFromSearchValue(final String input) {
2
3          final List<String> searchTerms = new ArrayList<String>();
4
5          final List<Article> articles = pArticleRepository
6                  .findAllByHeadingContainsOrIntroContains(input, input);
7
8          final Pattern pattern = Pattern.compile(
9                  "[^\\W]*" + Pattern.quote(input.toLowerCase()) + "[^\\W]*");
10
11         for (final Article article : articles) {
12                 final Matcher matcher = pattern
13                         .matcher(article.getHeading().toLowerCase());
14                 while (matcher.find()) {
15                         if (searchTerms.contains(matcher.group()) == false) {
16                                 searchTerms.add(matcher.group());
17                         }
18                 }
19                 final Matcher matcher = pattern
20                         .matcher(article.getIntro().toLowerCase());
21                 while (matcher.find()) {
22                         if (searchTerms.contains(matcher.group()) == false) {
23                                 searchTerms.add(matcher.group());
24                         }
25                 }
26         }
27
28         return searchTerms;
29 }
```

Quelltext 24: Event, welches vom Autocomplete-Mixin aufgerufen wird

In diesem Fall wird versucht eine Liste mit möglichen Suchbegriffen zusammenzustellen, die dem Nutzer angezeigt werden. Dazu werden erst einmal alle Datenbankeinträge geholt, welche auf die bisherige Eingabe zutreffen. Mittels regulärem Ausdruck werden diese Einträge durchsucht, aber so, dass die bisherige Eingabe auf vollständige Begriffe erweitert wird. Die vollständigen Begriffe werden als Strings einer neuen Liste hinzugefügt und anschließend von der Methode zurückgegeben.

5.5 Anzeigen eines Beitrags und Videoquelle

5.5.1 Aufbau der Seite

Zuletzt soll ein vom Nutzer gefundener Beitrag natürlich auch angesehen werden können. Hierfür wird eine extra Tapestry-Seite benötigt, auf der die Daten zu dem jeweiligen Beitrag angezeigt werden. Der Nutzer soll außerdem die Möglichkeit haben eine der Videoquellen, welche noch gültig ist, abzuspielen. Da es sich bei den Videoquellen lediglich um Links zu anderen Videoplattformen handelt, wird als Container-Komponente ein „iframe“ verwendet. Die meisten Videoportale inkl. Youtube, Vimeo und Dailymotion bieten ein sogenanntes „Embedding“ an, also das Einbinden von Videos auf fremden Webseiten. Somit wird das Abspielen eines Videos ein Kindespiel, da der Player von dem Videoportal gleich mitgeliefert wird und sich der Größe des Iframes anpasst. Sollte es vorkommen, das keine gültige Videoquelle mehr hinterlegt ist, wird in dem Fall einfach das Titelbild des Beitrags eingeblendet. Sämtliche sonstigen Funktionen der Seite funktionieren weiterhin und der Nutzer weiß hierbei, das das Video trotz mehrerer Quelle gesperrt wurde bzw. keine Videoquelle hinterlegt ist. Es kann dann dementsprechend gehandelt werden und eine neue Videoquelle zu dem Beitrag hinterlegt werden.

```
<t:if test="videoExisting">
        <iframe frameborder="0" width="640" height="360" src="${getVideoUrl()}" />
</t:if>
<t:if test="videoExisting" negate="true">
        <img src="${article.image}" onerror="this.src = '${errorImage}';" />
</t:if>

...

<i class="far fa-clock"></i>
<span>${datePublished}</span>
<i class="fas fa-chart-line"></i>
${article.views}

...

<h2>${article.heading}</h2>
<i class="fas fa-film margin-right-5"></i>
${article.articleGroupName}
${article.studioname}

...

<t:outputraw value="${article.intro}" />
```

Quelltext 25: Tapestry XML-Vorlage für die Anzeigeseite eines Beitrags

Ansonsten werden die restlichen Daten zu dem Beitrag angezeigt, wie Veröffentlichungsdatum, Aufrufzahl, Titel, Serien- und Studioname sowie die Beschreibung (INTRO) des Beitrags. Diese wird hierbei einer OutputRaw-Komponente von Tapestry übergeben, da die Beschreibung auch Formatierungen in HTML enthalten kann und diese so mitgerendert werden. Die Auswahl, welche URL dem Iframe übergeben wird, wird in der Java-Klasse getroffen.

```
@SetupRender
void setup() {

        article = articleAccessService.getArticleByUuid(articleUuid);

        if (articleVideos != null && articleVideos.isEmpty() == false) {
                for (int i = 0; i < articleVideos.size(); i++) {
                        final PArticleVideo pArticleVideo = articleVideos.get(i);
                        if (pArticleVideo .isValid()) {
                                videoUrl = pArticleVideo.getVideoUrl();
                                videoIndex = i;
                                break;
                        }
                }
        }
}

public String getDatePublished() {

        if (article.getDatePublished() != null) {
                final Long dateTime = article.getDatePublished();
                return DateTimeUtils.printDateTime(dateTime);
        }
        return "";
}

```

```
27  public boolean isVideoExisting() {
28
29          return videoIndex >= 0;
30  }
```

Quelltext 26: Java-Klasse für die Anzeigeseite eines Beitrags

In der SetupRender-Methode wird, wie gewohnt, der Beitrag aus der Datenbank geladen. Weiterhin wird überprüft, ob Videoquellen für den Beitrag zur Verfügung stehen. Ist das der Fall, wird über diese iteriert, bis eine als gültig markierte Videoquelle gefunden wurde. Entsprechend wird dann der Index und die URL der Videoquelle in eine Tapestry-Property geschrieben, um während des Rendervorgangs vom Iframe abgerufen werden zu können. Schließlich existieren noch zwei weitere Methoden. Eine, um das Veröffentlichungsdatum wieder in eine für Menschen lesbare Form umzuwandeln, und eine weitere, um anhand des soeben gesetzten Videoindex zu prüfen, ob der Iframe mit dem Video oder bei Nichtvorhandensein das Titelbild des Beitrags angezeigt wird.

5.5.2 Auswahl der nächsten Videoquelle

Da eine als gültig markierte Videoquelle nicht immer eineindeutig gültig sein muss, besteht für den Nutzer noch die Möglichkeit, manuell die Videoquelle zu wechseln. Hierfür wird eine weitere Komponente in der tml-Datei benötigt.

```
<t:eventlink t:event="changeArticleVideo" t:context="videoIndex">
```

Quelltext 27: Eventlink in Tapestry zum Wechsel der Videoquelle

Ein EventLink ruft in Tapestry eine spezielle EventHandler-Methode in der Java-Klasse der Tapestry-Seite auf.

```
1  void onChangeArticleVideo(final int index) {
2
3          article = articleAccessService.getArticleByUuid(articleUuid);
4
5          if (articleVideos.size() > (index + 1)) {
6                  videoUrl = article.getArticleVideos().get(index + 1).getVideoUrl();
7                  videoIndex = index + 1;
8          } else {
9                  videoUrl = article.getArticleVideos().get(0).getVideoUrl();
10                 videoIndex = 0;
11         }
12
13         ajaxResponseRenderer.addRender(iframezone);
14 }
```

Quelltext 28: Event, welches vom Tapestry-Eventlink aufgerufen wird

In diesem Fall soll nur der Iframe eine andere Videoquelle laden. Damit im nachhinein nicht die ganze Seite neu geladen werden muss, gibt es mit dem AjaxResponseRenderer von Tapestry die Möglichkeit, nur einen kleinen Bereich der Seite neu zu rendern. Dazu muss sich der gewünschte Bereich jedoch in einer Tapestry-Zone befinden. Im Falle des Iframes würde das wie folgt aussehen.

```
1  <t:zone t:id="middlezone">
2          <iframe frameborder="0" width="640" height="360" src="${getVideoUrl()}" />
3  </t:zone>
```

Quelltext 29: Tapestry-Zone um einen Iframe neu zu rendern

Durch den Klick eines Nutzers auf den EventLink wird also die Methode onChangeArticleVideo per HTTP-POST aufgerufen und die nächste Video-URL gesetzt. Anschließend wird nur der Iframe neu gerendert und der Nutzer erhält die nächste Videoquelle.

6. Automatisierung der Medienplattform

6.1 Automatisierung mittels Cronjobs

Neben den von Nutzern der Medienplattform getriggerten Aktionen existieren auch solche, die in regelmäßigen Abständen automatisch ausgeführt werden müssen. Dazu zählen unter anderem das Angleichen des Inhaltes einer Serie an den Inhalt eines Kanals auf YouTube, also dem Spiegeln der Videos eines YouTube-Kanals in eine Serie als Beiträge, oder das regelmäßige Prüfen der Videoquellen auf ihre Gültigkeit. Diese Prozesse werden gerne auch Cronjobs genannt („cron“ kommt aus dem Griechischen von „chronos“ und steht für „die Zeit“) und werden durch einfache Methoden ohne Parameter und Rückgabewert definiert, die im Spring-Framework mit der Annotation @Scheduled versehen werden (Spring nutzt hierzu das Framework Quartz). Der Annotation wird dann das Attribut „cron“ mitgegeben, eine Zeichekette, die letzlich den Zeitpunkt bestimmt, wann die Methode ausgeführt wird.

```
1  @Component
2  public class TestScheduler {
3
4          @Scheduled(cron = "15 0/2 * * * *")
5          public void testMethod1() { ... }
6
7          @Scheduled(cron = "50 15 0/3 * * *")
8          public void testMethod2() { ... }
9  }
```

Quelltext 30: Beispiel für die Erstellung eines Cronjobs in Spring

Der Aufbau der Zeichenkette folgt einem vorgegebenen Schema, in dem nacheinander Sekunde, Minute, Stunde, Tag im Monat, Monat, Wochentag und Jahr übergeben werden. Diese werden bestimmt durch Zahlen und/oder spezielle Zeichen wie in der Zeichenkette zu sehen ist. Hier wird testMethod1() bei 15 Sekunden, alle zwei Minuten, jede Stunde usw. ausgeführt. Die Methode

testMethod2() wird hingegen alle drei Stunden, bei Minute 15 und Sekunde 50, ausgeführt. Eine ausführlichere Beschreibung der Zeitbestimmung kann auf den Webseiten zum Spring-Framework nachgelesen werden.

6.2 Spiegeln eines YouTube-Kanals

6.2.1 Übersicht

Neben der Erstellung von Beiträgen in einer Serie durch den Nutzer selbst gibt es die Möglichkeit, einen YouTube-Kanal automatisch spiegeln zu lassen. Das heißt, dass sämtliche Videos des Kanals als Beiträge auf eine Serie der Medienplattform übertragen werden. Für die Implementierung kann die YouTube-API verwendet werden, wofür sogar eine eigene Client-Bibliothek für Java bereitstellt wird. Grob gegliedert unterteilt sich der Algorithmus in folgende vier Bestandteile:

- **Serien auf YT-Kanal-ID prüfen:**
  Iteration über sämtliche Serien einer Datenbank inkl. Prüfung, ob eine Kanal-ID für YouTube (CHANNEL_ID_YOUTUBE) hinterlegt ist.

- **Playlist von Youtube anfordern:**
  Nach neuestem Datum sortierte Playlist-Einträge des YT-Kanals anfordern und in Liste speichern, bis ein bereits gespiegelter Eintrag gefunden wurde.

- **Videoinformation anfordern:**
  Iteration über die Liste, beginnend vom letzten Element sowie für jedes Element das YT-Video anfordern.

- **Konvertieren und Speichern:**
  Playlist-Item und Videoinformation in Beitrag konvertieren und der Datenbank hinzufügen.

Der ganze Vorgang wird asynchron ausgeführt und kann weiterhin nur gestartet werden, wenn ein bereits begonnener Durchlauf abgeschlossen ist, um eine Inkonsistenz der Daten zu vermeiden, d.h. der aufrufende Cronjob wird stündlich gestartet, macht jedoch nichts, sollte der letzte Cronjob noch ausgeführt werden.

6.2.2 Serien auf YT-Kanal-ID prüfen

Die Prüfung der Kanal-ID erfolgt gleich in der Methode runYoutubeCrawler(), welche als Cronjob ausgeführt wird. Hier erfolgt zugleich auch die Prüfung, ob der Prozess der Spiegelung bereits läuft. Wenn das der Fall ist, wird lediglich eine Fehlermeldung ausgegeben. Kernstück der Methode ist die Initialisierung des YoutubeService mithilfe der Client-Bibliothek sowie Anfrage und Überprüfung aller Serien auf eine gültige Kanal-ID. Nur wenn eine gültige Kanal-ID gefunden wurde und der entsprechende Youtube-Kanal existiert, wird der eigentliche Algorithmus für die Spiegelung eines Youtube-Kanals auf eine Serie gestartet.

```
1  @Scheduled(cron = "30 30 0/1 * * *")
2  public void runYoutubeCrawler() {
3
4          if (isRunning == false) {
5                  isRunning = true;
6
7                  final YouTube youTube = getYoutubeService();
8
9                  final List<PArticleGroup> articleGroups =
10                         pArticleGroupRepository.findAll();
11                 for (final PArticleGroup group : articleGroups) {
12                         if (StringUtils.isBlank(group.getChannelIdYoutube())) {
13                                 continue;
14                         }
15
16                         try {
17                                 final Channel channel = getYoutubeChannel(
18                                         youTube, group.getChannelIdYoutube());
19
20                                 if (channel != null) {
21                                         runYoutubeCrawlerForArticleGroup(group,
22                                                 youTube, channel);
23                                 } else {
24                                         LOGGER.error("cannot get youtube channel");
25                                 }
26                         } catch (final IOException ioe) {
27                                 LOGGER.error(ioe.getMessage());
28                         }
29                 }
30
31                 isRunning = false;
```

```
32          } else {
33                  LOGGER.error("yt crawler is already running");
34          }
35  }
```

Quelltext 31: Cronjob zur Spiegelung von Youtube-Kanälen

Da während des weiteren Verlaufes sehr viele Netzwerkanfragen vorkommen und somit auch IOExceptions auftreten können, wird die Ausführung des Spiegelprozesses für einen YT-Kanal innerhalb eines try-catch-Blocks ausgeführt. So kann im Fehlerfall gleich mit der nächsten Serie fortgefahren werden und für den fehlerhaften Prozess wird eine entsprechende Fehlermeldung ausgegeben.

6.2.3 Playlist von Youtube anfordern

Wenn ein gültiger YouTube-Kanal gefunden worden ist, werden als nächstes die Uploads auf dem Kanal über eine PlayList angefordert. Da YouTube pro API-Request maximal 50 Objekte zurückgibt, muss die API über eine Schleife mehrmals angefragt werden. Dazu wird der Anfrage bei jedem Schleifendurchlauf ein PageToken mitgegeben.

```
 1  public void runYoutubeCrawlerForArticleGroup(final PArticleGroup group,
 2          final YouTube youTube, final Channel channel) throws IOException {
 3
 4          final List<PArticle> pArticleList = articleAccessService
 5                  .getArticlesByArticleGroup(group);
 6          final ArrayList<PlaylistItem> newDataList = new ArrayList<PlaylistItem>();
 7
 8          String nextToken = "";
 9          final YouTube.PlaylistItems.List playlistItemRequest =
10                  generateChannelUploadsRequest(youTube, channel);
11          do {
12                  playlistItemRequest.setPageToken(nextToken);
13                  final PlaylistItemListResponse playlistItemResult =
14                          playlistItemRequest.execute();
15
16                  final List<PlaylistItem> temp = playlistItemResult.getItems();
17
18                  for (final PlaylistItem item : temp) {
19                          if (isYTPlayListItemInArticleList(item, pArticleList)) {
20                                  addNewArticlesToDB(group, youTube, newDataList);
21                                  return;
22                          }
```

```
23                          newDataList.add(item);
24                  }
25                  nextToken = playlistItemResult.getNextPageToken();
26          } while (nextToken != null);
27
28          addNewArticlesToDB(group, youTube, newDataList);
29  }
```

Quelltext 32: Methode zur Spiegelung eines einzelnen Youtube-Kanals

Die einzelnen PlayListItems werden nun in einer weiteren Schleife in eine neue Liste übertragen. Das geschieht solange, bis ein bereits in der Datenbank vorhandener Beitrag gefunden wurde, der dem PlayListItem entspricht. Wenn das der Fall ist bzw. wenn beide Schleifen vollständig durchlaufen wurden, also demnach noch keine Beiträge einem PlayListItem entsprechen, dann müssen die in der Liste übertragenen PlayListItems als Beiträge in die Datenbank übertragen werden. Die PlayList ist bereits von YouTube nach den neuesten Beiträgen sortiert. Daher befinden sich in der neuen Liste ausschließlich die noch nicht in der Datenbank vorhandenen Einträge.

6.2.4 Videoinformation anfordern

Um bei einem Schleifenabbruch, z.B. durch eine IOException keinen Eintrag auszulassen, wird durch die Liste von hinten nach vorne iteriert. Ein Abbruch, z.B. durch einen Netzwerkfehler, würde bewirken, dass beim nächsten Start des Cronjobs die gleiche PlayList wieder bis zum bereits vorhandenen Datenbankeintrag geladen werden würde. Es ist somit eine lückenlose Spiegelung möglich.

```
1   private void addNewArticlesToDB(final PArticleGroup pArticleGroup,
2           final YouTube youTube, final List<PlaylistItem> list) throws IOException {
3
4           int localVideoLoadCount = 0;
5           final ListIterator<PlaylistItem> iterator = list.listIterator(list.size());
6           while (iterator.hasPrevious()) {
7                   final PlaylistItem item = iterator.previous();
8
```

```
 9                  if (localVideoLoadCount >= MAX_VIDEO_LOADCOUNT_PER_AGROUP) {
10                          return;
11                  }
12
13                  final Video video = getYoutubeVideoFromPlaylistItem(youTube, item);
14                  localVideoLoadCount++;
15
16                  createArticleFromYoutubeInformation(pArticleGroup, video, item);
17          }
18  }
```

Quelltext 33: Methode für den Abruf der Videodaten

Vor der Übertragung in die Datenbank muss jedoch noch die Videoinformation zum PlayListItem von der Youtube-API angefragt werden. Um die Netzwerkauslastung zu drosseln, kann an dieser Stelle der Algorithmus auch abgebrochen werden, wenn z.B. eine bestimmte Anzahl von Videos angefragt wurden. Dieses Prinzip schont gleichzeitig das erlaubte Limit bei den YouTube-Anfragen, welches sogar kostenpflichtig sein kann. Schließlich wird über die Methode createArticleFromYoutubeInformation() die Information von dem Video und dem PlayListItem in einen Beitrag für die Medienplattform konvertiert und in die Datenbank gespeichert.

6.2.5 Konvertieren und Speichern

Der Vorgang für die Konvertierung der Information und dem Anlegen des Beitrags in der Datenbank ist sehr simpel. Allerdings ist das Format für die einzelnen Felder des Beitrags sehr unterschiedlich im Vergleich zu dem Format der YouTube-Information, wodurch viele einzelne Konvertierungsschritte notwendig sind, welche, obwohl einfach, trotzdem die Methode sehr umfangreich gestalten. Bei den von der YouTube-API erhaltenen Information handelt es sich eigentlich um JSON-Objekte, welche jedoch von der Client-Bibliothek in Java-Objekte konvertiert werden. Hierzu zählen unter anderem Snippet, ContentDetails und Status. Da diese Objekte nicht zwangsläufig vorhanden sein müssen, muss vor dem Zugriff auf NULL geprüft werden, um einer NullPointerException vorzubeugen. Sollte dies so sein, wird einfach eine Fehlermeldung ausgegeben

und mit dem nächsten Beitrag fortgefahren. In diesem Fall kann eine Konvertierung nicht stattfinden und der Beitrag muss ausgelassen werden, obwohl hierdurch die Spiegelung nicht vollständig ist. Der fehlende Beitrag kann dann leider nur per Hand nachgetragen werden.

```
1  public void createArticleFromYoutubeInformation(final PArticleGroup group,
2          final Video video, final PlaylistItem item) throws IOException {
3
4          if (item.getSnippet() == null || item.getContentDetails() == null
5          || item.getContentDetails().getVideoId() == null
6          || video.getContentDetails() == null)  {
7                  LOGGER.error("missing important informations");
8          }
9
10         final PArticle newArticle = new PArticle();
11         newArticle.setArticleGroup(group);
12
13         if (item.getStatus() != null && item.getStatus().getPrivacyStatus() != null
14         && item.getStatus().getPrivacyStatus().equalsIgnoreCase("public")) {
15                 newArticle.setArticleAccessType(EArticleAccessType.PUBLIC);
16         } else {
17                 newArticle.setArticleAccessType(EArticleAccessType.BLOCKED);
18         }
19         final String dateStr = item.getSnippet().getPublishedAt().toStringRfc3339();
20         final long dateTime = DateTime.parseRfc3339(dateStr).getValue();
21         newArticle.setDatePublished(dateTime);
22         newArticle.setDateCreation(dateTime);
23         newArticle.setHeading(item.getSnippet().getTitle());
24         newArticle.setIntro(item.getSnippet().getDescription());
25
26         final int views = video.getStatistics().getViewCount().intValue();
27         newArticle.setViews(views);
28         newArticle.setUpdateCache(true);
29
30         copyImageFromThumbnails(newArticle, item.getSnippet().getThumbnails());
31
32         final PArticle articleSaved = articleAccessService.save(newArticle, null);
33
34         final String videoId = item.getContentDetails().getVideoId();
35         final String durStr = video.getContentDetails().getDuration();
36         final int duration = (int) Duration.parse(durStr).getSeconds();
37         articleAccessService.addVideoToArticle(articleSaved,
38                 EVideoProvider.YOUTUBE, videoId, duration, views, dateTime);
39 }
```

Quelltext 34: Methode zur Konvertierung eines Youtube-Videos in einen Beitrag

Wenn nun sämtliche Information vorhanden ist, kann die Konvertierung beginnen. Zuallererst wird ein neuer Beitrag angelegt und der richtigen Serie zugewiesen. Dann erfolgt Schritt für Schritt die Umwandlung und Zuweisung der Daten wie Zugriffstyp, Veröffentlichungsdatum, Titel und Beschreibung. Anschließend wird noch ein Thumbnail als Titelbild kopiert und der Beitrag in der Datenbank gespeichert. Als letzten Schritt wird eine Videoquelle, welche auf das YouTube-Video verlinkt, erzeugt und auch in der Datenbank gespeichert.

6.3 Videoquellen auf Gültigkeit prüfen

Um eine Zensursicherheit durch Redundanz zu gewährleisten, also bei Löschung eines Videos in einer Videoquelle sogleich die nächste Videoquelle anzuzeigen, benötigt es einen Cronjob, welcher zyklisch die Videoquellen auf ihre Gültigkeit kontrolliert. Das kann je nach Videoanbieter über einen API-Zugriff oder über das Parsen des HTML-Codes der Videoseite geschehen. In jedem Fall wird für die unterschiedlichen Videoanbieter jeweils eine eigene Methode benötigt, um die benötigten Daten abzufragen. Daher lohnt sich hierfür die Implementierung eines Interface mit einer Methode, welche immer mit der Video-ID aufgerufen werden kann und dann entsprechende Information über das Video als Rückgabe liefert.

```
1  public abstract interface IVideoProviderService {
2
3          Tuple2<Integer, Boolean> getVideoInfo(final String videoId);
4  }
```

Quelltext 35: Interface zum Abruf von Information diverser Videoanbieter

Im folgenden soll nun als Beispiel die Implementierung des Interface für den Videoanbieter BitChute gezeigt werden, welcher anders als YouTube keine API bereitstellt, um Information anzufragen.

```
1  public class BitchuteService implements IVideoProviderService {
2
3          private static final String ID = "{VIDEO_ID}";
4
5          private static final String REQUEST =
6                  "https://www.bitchute.com/video/{VIDEO_ID}/counts";
7
8          @Override
9          public Tuple2<Integer, Boolean> getVideoInfo(final String videoId) {
10
11                 final String requestUrl = StringUtils.replace(REQUEST, ID, videoId);
12
13                 final String content = Frei3UrlUtils.downloadFromUrl(requestUrl);
14                 if (StringUtils.isBlank(content)) {
15                         return null;
```

```
16                  }
17
18                  try {
19                          final JSONObject json = new JSONObject(content);
20                          return new Tuple3<Integer, Boolean>(
21                                  json.getInt("view_count"), Boolean.TRUE);
22                  } catch (final JSONException e) {
23                          return new Tuple3<Integer, Boolean>(null, Boolean.FALSE);
24                  }
25          }
26  }
```

Quelltext 36: Beispiel Implementierung des Videoanbieter Interface für Bitchute

Bei dem Videoanbieter BitChute ist es möglich der Video-Url ein „counts“ anzufügen, wodurch man ein JSON-Object erhält, welches BitChute selbst für den Abruf mit JavaScript benötigt, um die unterschiedlichen Zähler wie z.B. die aktuellen Aufrufzahlen des Videos auf der Webseite zu aktualisieren. Dieser Wert wird auch hier angefragt und lässt darauf schließen, ob das Video noch vorhanden ist oder aus irgendeinem Grund gelöscht wurde. Dieses Prinzip stellt nur eine von vielen Möglichkeiten dar, die Gültigkeit eines Videos zu prüfen und variiert wie bereits erwähnt bei allen Videoanbietern. Außerdem ist nicht garantiert, das die Methode in Zukunft immer funktionieren wird, da der Videoanbieter seine eigene Webseite ständig verändern kann. In dem Fall muss nach einer neuen Möglichkeit gesucht werden, die benötigten Information abzufragen und die Methode angepasst werden. Identisch bleibt jedoch das Interface mit der abstrakten Methode, die wie folgt in dem Cronjob verwendet wird.

```
1   @Scheduled(cron = "0 0 0 * * *")
2   public void updateVideoData() {
3
4           final List<PArticleVideo> videoList = pArticleVideoRepository.findAll();
5           for (final PArticleVideo pArticleVideo : videoList) {
6                   final Tuple2<Integer, Boolean> data;
7                   switch (pArticleVideo.getVideoProvider()) {
8                   case DAILYMOTION:
9                           data = dailymotion.getVideoInfo(pArticleVideo.getVideoId());
10                          break;
11                  case YOUTUBE:
12                          data = youtube.getVideoInfo(pArticleVideo.getVideoId());
13                          break;
14                  case BITCHUTE:
15                          data = bitchute.getVideoInfo(pArticleVideo.getVideoId());
16                          break;
17                  default:
18                          data = null;
19                  }
20
21                  if (data == null) {
```

```
22                          LOGGER.error("cannot get video info");
23                          continue;
24                  }
25
26                  boolean updated = false;
27                  if (data.getA() != null) {
28                          pArticleVideo.setViews(data.getA().intValue());
29                          updated = true;
30                  }
31                  if (data.getB() != null) {
32                          pArticleVideo.setValid(data.getB().booleanValue());
33                          updated = true;
34                  }
35                  if (updated == false) {
36                          continue;
37                  }
38
39                  pArticleVideo.setDateApiUpdate(System.currentTimeMillis());
40                  pArticleVideoRepository.save(pArticleVideo);
41          }
42  }
```

Quelltext 37: Cronjob für die Aktualisierung der Videoinformation

Die Methode besteht aus einer Hauptschleife, in der jede in der Datenbank vorhandene Videoquelle abgearbeitet wird. Bei jedem Durchlauf wird die Video-ID an eine des Videoanbieters entsprechende Implementierung der Methode in dem zuvor beschriebenen Interface übergeben. Die von getVideoId() in „data“ zurückgegebene Videoinformation wird anschließend ausgewertet und die aktualisierte Videoquelle anschließend in der Datenbank gespeichert. Insgesamt bietet das Verfahren eine schlanke Möglichkeit die Videoquellen auf Gültigkeit zu prüfen. Allerdings sollte auf die Häufigkeit der Ausführung des Cronjobs geachtet werden, wenn in den getVideoId()-Methoden keine offizielle API des Zielservers benutzt wird. Der Zielserver könnte den Zugriff sperren, wenn z.B. zu viele Anfragen von der selben IP auf den Server registriert wurden.

7. Schlussbemerkungen

7.1 Fazit

Wie in den einzelnen Kapiteln aufgezeigt wurde, ist es unter der Nutzung von modernen Frameworks wie Spring und Tapestry keine große Schwierigkeit mehr, eine solide Medienplattform zu implementieren. Durch die strikte Trennung der Programmlogik durch Spring-Services von der Datenhaltung mittels JPA-Repository und schließlich der Anzeige im Webbrowser mittels Tapestry-Seiten wurde auf elegante Weise das MVC-Prinzip (Model View Controller) umgesetzt. Da die Datenhaltung so abstrakt wie möglich gehalten wurde und die Implementierung durch Hibernate automatisch vollzogen wird, ist der Einsatz von verschiedensten Datenbanksystemen möglich. Tapestry stellt eine breite Palette an Komponenten bereit was die grafische Darstellung der Medienplattform enorm einfach macht und gleichzeitig ein hohes Maß an Sicherheit gegenüber Serverattacken im Zusammenspiel mit der Datenhaltung durch JPA ermöglicht.

Weiterhin wurde dargestellt, wie mit einfachen Mitteln eine solide Zensursicherheit erreicht werden kann. Die Auslagerung der eigentlichen Videoinhalte auf mehreren fremden Videoplattformen in redundanter Weise bietet eine solide Sicherheit für die eigentlichen Inhalte. Durch die Aufgliederung der internen Datenhaltung in Studios, Serien, Beiträgen und Videoquellen wird dem Nutzer eine einfach zu verstehende Struktur an die Hand gegeben. Zusätzlich wird durch die automatische Prüfung der Videoinhalte mittels Cronjob eine zeitnahe Erkennung ungültiger Videoquellen ermöglicht, sodass einer weiteren Sperrung auf anderen Seiten durch weitere Vervielfältigung des entsprechenden Videoinhaltes rechtzeitig entgegen gewirkt werden kann.

7.2 Ausblick

Grundsätzlich bietet das in dieser Abhandlung beschriebene Projekt schon einen soliden Umfang, um gleichzeitig eine einfach strukturierte und dennoch innovative Medienplattform aufzusetzen, die für den Nutzer einfach zu verstehen und zu bedienen ist sowie gleichzeitig ein gutes Maß an Datensicherheit bietet. Dennoch können diese Punkte im zukünftigen Ausbau des Projektes noch weiter optimiert sowie durch zusätzliche Funktionalität erweitert werden.

Dazu zählt einerseits die Umsetzung einer höheren Skalierbarkeit, um dem steigenden Nutzeraufkommen gerecht zu werden. Durch den Einsatz weiterer Datenbanksysteme wie z.B. Cassandra, welche für den Einsatz für BigData optimiert sind, sowie speziellen Suchmaschinen, wie Elasticsearch, kann die Verfügbarkeit für Millionen von Nutzern problemlos gewährleistet werden.

Weiterhin kann eine noch bessere Zensursicherheit über verschiedene Möglichkeiten erreicht werden. Dazu zählt neben dem Auslagern der Videoinhalte auf unterschiedliche Videoportale z.B. das eigene Hosting der Videoinhalte auf verschiedenen Servern, welche gleichmäßig auf der Welt verteilt agieren. Zudem könnte dem Nutzer die Möglichkeit geboten werden, eigene Video-Server mit einer speziell implementierten Software aufzusetzen, die wiederum die Links für die Einbettung auf der hier implementierten Medienplattform liefert.

Schließlich sollte noch erwähnt werden, dass eine hundertprozentige Zensursicherheit mit einer zentral betriebenen Software nicht erreicht werden kann. Daher gehört zu den zukünftigen Aufgaben auch eine zunehmende Dezentralisierung der Serverstruktur, was auch die Verteilung auf unterschiedliche Betreiber beinhaltet. Es darf sozusagen niemanden geben, der auf sämtlichen Systemen über die vollen Administratorrechte verfügt.

Mit diesen abschließenden Worten soll die Arbeit beendet werden, auch wenn selbst diese nicht gänzlich allen Möglichkeiten entsprechen, die in diesem Zusammenhang umgesetzt werden könnten.

Stichwortverzeichnis

Abkürzungsverzeichnis

API	**A**plication **P**rogramming **I**nterface
DSGVO	**D**aten**s**chutz-**G**rund**v**er**o**rdnung
FK	**F**oreign **K**ey (Fremdschlüssel)
HTML	**H**yper**t**ext **M**arkup **L**anguage
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
ID	**Id**entifier (**Id**entifikation)
IP	**I**nternet **P**rotocol
JPA	**J**ava **P**ersistence **A**PI
JSON	**J**ava**S**cript **O**bject **N**otation
MVC	**M**odel **V**iew **C**ontroller
PK	**P**rimary **K**ey (Primärschlüssel)
POJO	**P**lain **O**ld **J**ava **O**bject
SQL	**S**tructured **Q**uery **L**anguage
TML	**T**apestry **M**arkup **L**anguage
UUID	**U**niversally **U**nique **Id**entifier
URL	**U**niform **R**esource **L**ocator
XML	E**x**tensible **M**arkup **L**anguage
YT	**Y**ou**T**ube

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Glossar

Begriff	Definition / Erklärung
Annotation	Annotationen dienen der Strukturierung des Quelltextes, indem die Erzeugung von Programmtexten und Hilfsdateien zum Teil automatisiert werden.
Cronjob	wiederkehrende Aufgabe, welche mithilfe des Cron-Daemon gestartet wird. Dieser dient der zeitbasierten Ausführung von Prozessen in Unix und unixartigen Betriebssystemen wie Linux, BSD oder macOS.
Dailymotion	Ein Videoportal des gleichnamigen französischen Unternehmens.
Dependency Injection	Ein in der objektorientierten Programmierung bezeichnetes Entwurfsmuster, welches die Abhängigkeiten eines Objekts erst zur Laufzeit regelt.
Flags	Statusindikatoren, welche als Hilfsmittel zur Kennzeichnung bestimmter Zustände benutzt werden können
Framework	Ein Programmiergerüst in der Softwaretechnik, das in der objektorientierten sowie bei der komponentenbasierten Entwicklung verwendet wird.
Iframe	Ein Iframe, oder Inlineframe ist ein HTML-Element, das der Strukturierung von Webseiten dient.
OpenSource	Software, deren Quelltext öffentlich und von Dritten eingesehen, geändert und genutzt werden kann.
Redundanz	Information, die in einer Informationsquelle mehrfach vorhanden ist.
Rendern	Rendern ist die Erstellung einer Grafik aus Rohdaten z. B. die Visualisierung der Webseite durch den HTML-Code.
Streaming	Datenübertragungsverfahren, bei dem die Daten bereits während der Übertragung angesehen oder angehört werden können.
Vimeo	Ein Videoportal des US-amerikanischen Unternehmens Vimeo LLC.
YouTube	Ein Videoportal des US-amerikanischen Unternehmens YouTube, LLC.

Literaturverzeichnis

[zu Kap. 1.3] Wikipedia: „Videoportal“ unter
https://de.wikipedia.org/wiki/Videoportal

[zu Kap. 2.1] Stephan Augusten: „Was ist das Spring Framework?“ unter
https://www.dev-insider.de/was-ist-das-spring-framework-a-829846/

[zu Kap. 2.2] Frank W. Rahn: „Spring an einem einfachen Beispiel (Tutorial)“ unter
https://www.frank-rahn.de/spring-einem-einfachem-beispiel/

[zu Kap. 3.4] Oliver Gierke, Thomas Darimont, Christoph Strobl: „Spring Data JPA - Reference Documentation“ unter
https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html

[zu Kap. 3.4] Baeldung: „Spring Data JPA @Query“ unter
https://www.baeldung.com/spring-data-jpa-query

[zu Kap. 6.2] Google LLC: „YouTube Data API“ unter
https://developers.google.com/youtube/v3

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit selbstständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der von mir angegebenen Literatur und Hilfsmittel angefertigt habe.

Schackenthal, den 06.11.2019 Rudolf Niehoff

Einverständniserklärung

Ich erkläre mein Einverständnis zu einer Veröffentlichung der vorliegenden Arbeit im Internet.

Ja Nein

Schackenthal, den 06.11.2019 Rudolf Niehoff