UNIVERSITÄT LEIPZIG

Digital Text Forensics Search

Information Retrieval
WS 2017/2018

Betreurer: Martin Potthast
martin.potthast@uni-leipzig.de

Autor: David Drost
dd42cequ@studserv.uni-leipzig.de
Matrikelnummer:

Autor: Edward Kupfer
ek96foje@studserv.uni-leipzig.de
Matrikelnummer:

Autor: Hendrik Sawade
hs34byhe@studserv.uni-leipzig.de
Matrikelnummer: 3745956

Autor: Tobias Wenzel
tw54byka@studserv.uni-leipzig.de
Matrikelnummer: 3733301

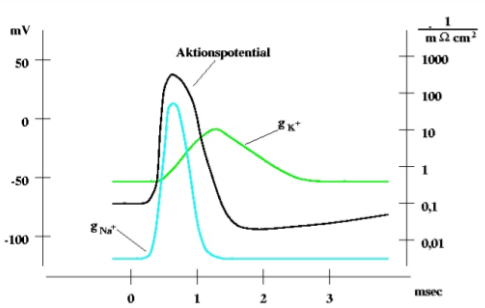
Abgabedatum: Leipzig, den 5. März 2018

Inhaltsverzeichnis

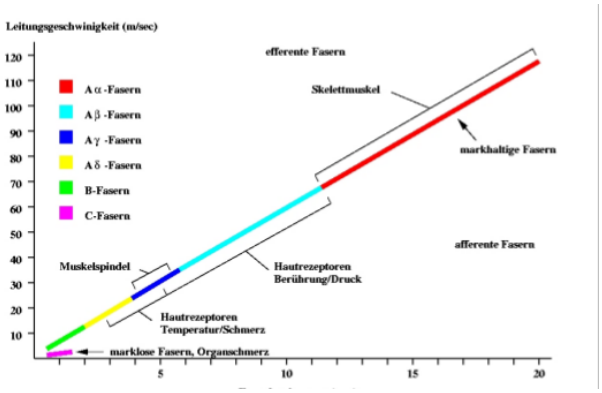
1 Einleitung

Hier schreibt man dann was. Und so . Offenbar sieht das nicht so schön aus.

kommentiere ihr Sachen.



(a) Pyramidenzelle



(b) Neocortex

Abbildung 1.1: Links: AP. Rechts: Leitungsgeschwindikeit von verschiedenen Fasern.

2 Vorverarbeitung

3 Indexierung

4 Backend

4.1 Auswahl eines Backend-Frameworks

Zuerst wurde für die Programmierung einer Webanwendung ein geeignetes Framework gesucht, um Programmier-Paradigmen umzusetzen und die Architektur besser zu abstrahieren. Hierfür wurden zahlreiche Frameworks untersucht, welche Dependency Injection (DI), Inversion of Control (IoC) und Aspect-Oriented Programming (AOP) unterstützen. Aufgrund der Auswahl der Programmiersprache Java für die Umsetzung der Anwendung schränkte sich die Anzahl der Frameworks ein. Die Recherche ergab folgende drei Frameworks:

Frameworks	Sprache	Eigenschaft
HiveMid	Java	DI, AOP-ähnliches Feature, IoC Container
Google Guice	Java	DI, AOP, IoC Container, Annotations, Generics, modular
Spring Boot	Java	DI, AOP, IoC Container, Annotations, Generics, modular

Tabelle 4.1: DI-Frameworks

HiveMind und Google Guice bieten gegenüber Spring Boot leichter verständliche Programmierungstechniken sowie einen prägnanteren und lesbareren Code. HiveMind fokussiert sich auf das Verbinden von Services. Seine Konfiguration erfolgt über eine XML-Datei oder eine eigene Definitions-Sprache. Hierdurch ist HiveMind ein kleiner und simpel gestalteter DI-Container. Darüber hinaus bietet HiveMind die Möglichkeit, mit AOP zu arbeiten. Google Guice hingegen unterstützt Features wie Annotations und Generics, die ab Java 1.5 zur Verfügung stehen. Sie helfen dabei, eine weitgehend aufgeräumte und einfache Konfiguration zu ermöglichen. Google Guice und Spring Boot bieten sehr ähnliche Ansätze und kommen mit vielen Anforderungen, die Unternehmenssoftware erfüllen müssen, zurecht. Google Guice ist durch die geringere Komplexität leichter zu verstehen und insgesamt kleiner als Spring Boot. Jedoch bietet die Modularität von Spring Boot

den größeren Vorteil: Die Module können, je nachdem welche der Entwickler benötigt, ohne viel Aufwand hinzugefügt werden. Schlussendlich fiel die Entscheidung auf das Spring Boot-Framework, da dessen Flexibilität und Modularität die Entwicklung von Anwendungen stark vereinfacht und daher die beste Wahl darstellt.

4.2 Auswahl einer Datenbank

Als nächstes wurde für das spätere Speichern der Interaktion zwischen Backend, Suchergebnissen und User, genauer des User-Feedbacks eine geeignetes eingebettetes Datenbanksystem gesucht. Die Recherche ergab folgende Datenbanken:

Datenbank	Sprache	Eigenschaft
SQLite	C	SQL-92-Standard, Transaktionen, Unterabfragen (Subselects), Sichten (Views), Trigger und benutzerdefinierte Funktionen, direkte Integration in Anwendungen, In-Memory-Datenbank
Apache Cassandra	Java	Spaltenorientierte NoSQL-Datenbank, für sehr große strukturierte Datenbanken, hohe Skalierbarkeit und Ausfallsicherheit bei großen, verteilten Systemen
H2	Java	Schnell, Referenzielle Integrität, Transaktionen, Clustering, Datenkompression, Verschlüsselung und SSL, direkte Einbettung in Java-Anwendungen oder Betrieb als Server möglich, direkte Unterstützung in Spring Boot, In-Memory-Datenbank

Tabelle 4.2: Datenbanken

SQLite bietet einen leichten Einstieg in die Datenbanken. Dabei stellt SQLite den größten Teil des SQL-92-Standards zur Verfügung und kann Transaktionen, Unterabfragen und viele weitere Funktionen durchführen. Außerdem ist es eine In-Memory-Datenbank. Jedoch unterstützt Spring Boot diese Datenbank nicht von Haus aus und es müssten aufwendige Konfiguration vorgenommen werden.

Apache Cassandra ist eine spaltenorientierte NoSQL-Datenbank und ist für große strukturierte Daten, hohe Skalierbarkeit und Ausfallsicherheit ausgelegt. Für das vorliegende Projekt ist Apache Cassandra jedoch zu groß ausgelegt, da für das Backend mit geringeren Datenmengen gearbeitet werden soll.

H2 ist eine In-Memory-Datenbank, welche schnell ist und referenzielle Integrität, Transaktionen, Clustering sowie Datenkompression unterstützt. Außerdem kann Spring Boot mit dieser Datenbank ohne besondere Maßnahmen wie aufwändige Konfigurationen verwendet und in die vorliegende Anwendung integriert werden. Deshalb wurde entschlossen, H2 als Datenbank anzuwenden.

4.3 Erstellung des Backends

Nach der Auswahl der Backendtechnologien wurde die Grundarchitektur des Backends konzipiert und implementiert.

4.3.1 Kommunikation mit der Datenbank

Zunächst wurden Datenmodels wie `Query` oder `LoggingDocument` erstellt. Hieraus werden später die Tabellen der Datenbank generiert. Um mit der Datenbank kommunizieren zu können, werden Data Access Objects (DAO) als Kommunikationsschnittstellen erstellt. Ein DAO hat eine Anbindung zu den Spring-Boot Repositorys, welche in der Lage sind SQL-Query zu generieren und übermittelt diese an die Datenbank. Beispiele hierfür sind das Speichern und Abrufen von `LoggingDocument`-Daten, welche einen Teil des User-Feedbacks darstellen.

```
1 public interface LoggingDocDao extends JpaRepository<
    LoggingDocument, Long> {
2 LoggingDocument findByDocId(Long docId);
3 }
```

Im obigen Code-Ausschnitt wird mit Hilfe von Spring Boot die SQL-Qyery `findByDocId` aus dem `LoggingDocument` generiert. Dies findet über den Namen eines Interfaces statt. Die einzelnen Komponenten, welche implementiert wurden, kommunizieren nicht direkt über die DAOs mit der Datenbank, sondern über ein Interface. Dadurch ist eine lose Kopplung zwischen den Komponenten, DAOs und der Datenbank möglich. Damit ist die Datenbank ohne große Änderungen in den Implementierungen austauschbar. Folglich fehlen nur noch Änderungen in den Konfigurationen und eventuell in den DAOs.

```
1 public class LoggingDocServiceImpl implements ↩
    LoggingDocService {
2   public LoggingDocument findbyId(Long id) {
3   return loggingDocDao.findOne(id);
4   }}
```

Im vorliegenden Listing ist `LoggingDocService` als Beispiel für einen Service dargestellt. In der Implantation des Interfaces wird nun das DAO aufgerufen, beispielsweise die Methode `findbyId`.

4.3.2 Controller

Im nächsten Schritt wurden sogenannte Controller erstellt. Diese bilden eine wichtige Schnittstelle für die Kommunikation mit dem Frontend und Backend. Controller reagieren auf HTTP-Requests, welche von dem Frontend oder anderen Clients gesendet werden. Die Aufgabe ist es, für bestimmte Ressource-URLs spezielle Ereignisse auszuführen. Ein Beispiel hierfür ist Auswertung der Suchanfrage der Search-Zeile im Frontend und das Rücksenden der Suchergebnisse.

```
1 @RequestMapping(method = RequestMethod.GET, path = "/")
2 public ModelAndView searchPage(
3 @RequestParam(defaultValue = "")
4 String query) {
5 ModelAndView modelAndView = new ModelAndView("search");
6 ...
7 List<ScoreDoc> list = querySearcher.search(query);
8 ...
9 modelAndView.addObject("searchResultPage", searchResultPage);
10 return modelAndView;}
```

Im obigen Code-Beispielabschnitt ist erkennbar, dass, beim Auslösen eines Request bei der Path-URL „/“ die Funktion `SearchPage` aufgerufen und eine Suche ausgeführt wird. Hierfür wird der Request-Parameter mit `query` ausgewertet. Die Suche erfolgt mithilfe der Komponente Lucene, welche bereits in Abschnitt 3 näher erläutert wurde. Im Anschluss werden die Suchergebnisse als `modelAndView`-Objekt dem Frontend übergeben.

4.3.3 Komponenten des Frontends

Nach dem Controller wurden die Komponenten, welche für das Frontend benötigt werden, konzipiert und im Anschluss implementiert.

Wurden die Suchen durchgeführt und die Lucene-Komponente die Suchergebnisse zurückgegeben, wird die gesamte Ergebnisliste gesplittet. Hierbei wird für die angeforderte Seite eine Subliste erstellt, in welcher nur die geforderten Suchergebnisse enthalten sind und die restlichen Ergebnisse verworfen werden. Hierdurch ist es nicht erforderlich, alle Ergebnisse zu transformieren. Dadurch arbeitet die Anwendung wesentlich schneller.

Als nächstes wurde ein Data Transfer Object (DTO) erstellt, um die relevanten Suchergebnisse, die in der Subliste enthalten sind, in das gewünschte Ausgabeformat zu überführen und in einer separaten Liste zu sammeln. Das DTO hat dabei unter anderem die Variablen Autor, Titel, Snippet oder den Redirect-Link, welcher auf die zugehörige PDF zeigt. Der Link hierfür wird mit der Methode `createLink` erzeugt. Zu diesem Zweck werden aus der docId, Query und dem Host ein Link erstellt. Als Beispiel wird folgender Link generiert:

http://localhost:8080/pdf/?docId=1&query=xyz.

Der Parameter `docId` ist dabei eine Id, welche die PDF zu dem Suchergebnis angibt und der Parameter `query` dient dem User-Feedback. Beim späteren Klick auf das Suchergebnis wird zum einen die dazugehörige PDF angezeigt und zum anderen gleichzeitig in der Datenbank das Dokument, welches mit dem einer bestimmten Query gefunden wurde, gespeichert. Damit ist es möglich, Rückschlüsse auf die Wichtigkeit des Dokuments zu ziehen. Der eben beschriebene Vorgang erfolgt mit der Methode `mapDocumentListToSearchResults`. Nach dem Erstellen der Liste der transformierten Suchergebnisse, wird sie zum Objekt `searchResultPage` hinzugefügt und um weitere Angaben ergänzt. Das ist im folgenden Code-Abschnitt ausschnittsweise zu sehen.

```
List<ScoreDoc> split = pager.split(list, currentPage);
searchResultList = querySearcher.↩
    mapDocumentListToSearchResults(split,query);
searchResultPage.setTotalResults(list.size());
searchResultPage.setResultsOnPage(searchResultList);
searchResultPage.setPage(currentPage);
```

Hinzu zum Beispiel kommt die Gesamtanzahl an Suchergebnissen oder auf welcher Page man sich befindet.

Die `searchResultPage` wird nun der Spring Boot Thymeleaf-Komponente übergeben und die `search.html`-Page erstellt. Hierfür wurde ein `search.html`-Template erstellt, in welchem Anweisungen zum Umgang mit den übergebenden Daten gegeben werden. Thymeleaf befolgt diese Anweisungen und wandelt sie in entsprechende HTML-Komponenten um, damit im Anschluss der der Umwandlung ein Webbrowser die Page anzeigen kann. Ein Beispiel der Anweisungen für Thymeleaf wird im folgenden Code-Abschnitt aufgezeigt.

```
<div th:each="result : ${searchResultPage.resultsOnPage}">
  <h3 class="card-title">
    <a th:href="${result.webUrl.href}"
      th:text="${result.title}">
    </a>
  </h3>
</div>
```

Das `searchResultPage`-Objekt wird aufgerufen. Daraufhin wird in einer Schleife die Liste der DTO-Objekte, die im `searchResultPage`-Objekt enthalten sind, mit Titel und `webURL` als HTML h3-Tag, welche einen Link darstellt, erstellt. Durch die Schleife wird somit für jedes Element ein eigenes HTML-Element generiert. Die Gestaltung der Oberfläche und deren Elemente wurde in separaten CSS- und Javascript-Dateien vorgenommen. Nach der Erstellung der `search.html`-Page wird die fertige Page über den Request zurückgegeben und der Webbrowser zeigt sie an.

5 Evaluation und Fazit